



# Sintaxe e Semântica de Programas Prolog



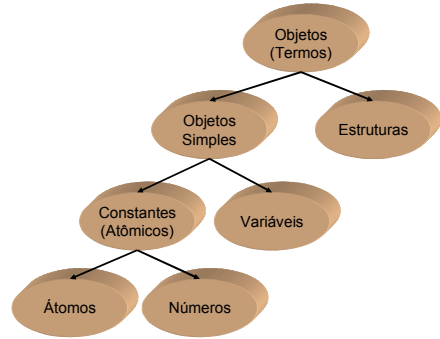
Inteligência Artificial

- Nesta aula será vista a sintaxe e semântica de conceitos básicos em Prolog e introduz objetos de dados estruturados
- Os tópicos abordados são:
  - Objetos simples (átomos, números, variáveis)
  - Objetos estruturados
  - Unificação como operação fundamental em objetos
  - Operadores
  - Significado declarativo e procedural

José Augusto Baranauskas  
Departamento de Física e Matemática – FFLRP-USP

E-mail: [augusto@usp.br](mailto:augusto@usp.br)  
URL: <http://dfm.fmrp.usp.br/~augusto>

# Objetos em Prolog



# Átomos

- São cadeias compostas pelos seguintes caracteres:
  - letras maiúsculas: A, B, ..., Z
  - letras minúsculas: a, b, ..., z
  - dígitos: 0, 1, ..., 9
  - caracteres especiais: + - \* / < > = : . & \_ ~
- Podem ser construídos de três maneiras:
  - cadeias de letras, dígitos e o caractere '\_', começando com uma letra minúscula: anna, nil, x25, x\_25, x\_25AB, x\_, x\_y, tem\_filhos, tem\_um\_filho
  - cadeias de caracteres especiais: <-->, =====, ..., ..., ::=
  - cadeias de caracteres entre apóstrofes: 'Abraão', 'América\_do\_Sul', 'América\_Latina'

# Números

- Números usados em Prolog incluem números inteiros e números reais

Operadores Aritméticos	
adição	+
subtração	-
multiplicação	*
divisão	/
divisão inteira	//
resto divisão inteira	mod
potência	**
atribuição	is

Operadores Relacionais	
X > Y	X é maior do que Y
X < Y	X é menor do que Y
X >= Y	X é maior ou igual a Y
X <= Y	X é menor ou igual a Y
X =:= Y	X é igual a Y
X = Y	X unifica com Y
X \= Y	X é diferente de Y

# Números

- O operador = tenta unificar apenas
  - ?- X = 1 + 2.
  - x = 1 + 2
- O operador is força a avaliação aritmética
  - ?- X is 1 + 2.
  - x = 3
- Se a variável à esquerda do operador is já estiver instanciada, Prolog apenas compara o valor da variável com o resultado da expressão à direita de is
  - ?- X = 3, X is 1 + 2.
  - x = 3
  - ?- X = 5, X is 1 + 2.
  - no

# Variáveis

- São cadeias de letras, dígitos e caracteres '\_', sempre começando com letra maiúscula ou com o caractere '\_'
  - X, Resultado, Objeto3, Lista\_Alunos, ListaCompras, \_x25, \_32
- O escopo de uma variável é dentro de uma mesma regra ou dentro de uma pergunta
- Isto significa que se a variável X ocorre em duas regras/perguntas, então são duas variáveis distintas
- Mas a ocorrência de X dentro de uma mesma regra/pergunta significa a mesma variável

## Variáveis

- Uma variável pode estar:
  - Instanciada: quando a variável já referencia (está unificada a) algum objeto
  - Livre ou não-instanciada: quando a variável não referencia (não está unificada a) um objeto, ou seja, quando o objeto a que ela referencia ainda não é conhecido
- Uma vez instanciada, somente Prolog pode torná-la não-instanciada através de seu mecanismo de inferência (nunca o programador)

7

## Variável Anônima

- Quando uma variável aparece em uma única cláusula, não é necessário utilizar um nome para ela
- Utiliza-se a variável **anônima**, que é escrita com um simples caracter '\_'. Por exemplo
  - `temfilho(X) :- progenitor(X,Y).`
- Para definir **temfilho**, não é necessário o nome do filho(a)
- Assim, é o lugar ideal para a variável anônima:
  - `temfilho(X) :- progenitor(X,_).`

8

## Variável Anônima

- Cada vez que um *underscore* '\_' aparece em uma cláusula, ele representa uma nova variável anônima
- Por exemplo
  - `alguém_tem_filho :- progenitor(_,_).`equivale à:
  - `alguém_tem_filho :- progenitor(X,Y).`que é bem diferente de:
  - `alguém_tem_filho :- progenitor(X,X).`
- Quando utilizada em uma pergunta, seu valor não é mostrado. Por exemplo, se queremos saber quem tem filhos mas sem mostrar os nomes dos filhos, podemos perguntar:
  - `?- progenitor(X,_).`

9

## Estruturas

- Objetos estruturados (ou simplesmente estruturas) são objetos de dados que têm vários componentes
- Cada componente, por sua vez, pode ser uma estrutura
- Por exemplo, uma data pode ser vista como uma estrutura com três componentes: dia, mês, ano
- Mesmo possuindo vários componentes, estruturas são tratadas como simples objetos

10

## Estruturas

- De forma a combinar componentes em um simples objeto, deve-se escolher um **functor**
- Um functor para o exemplo da data seria **data**
- Então a data de 4 de maio de 2003 pode ser escrita como:
  - `data(4,maio,2003)`

11

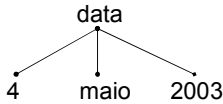
## Estruturas

- Qualquer dia em maio pode ser representado pela estrutura:
  - `data(Dia,maio,2003)`
- Note que **Dia** é uma variável que pode ser instanciada a qualquer objeto em qualquer momento durante a execução
- Sintaticamente, todos objetos de dados em Prolog são termos
- Por exemplo, são termos:
  - `maio`
  - `data(4,maio,2003)`

12

## Estruturas

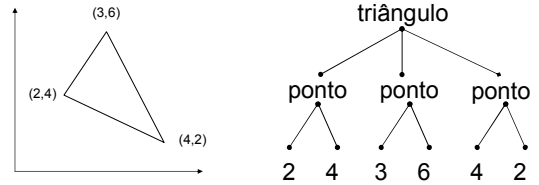
- ❑ Todos os objetos estruturados podem ser representados como árvores
- ❑ A raiz da árvore é o functor e os filhos da raiz são os componentes
- ❑ Para a estrutura `data(4,maio,2003)`:



13

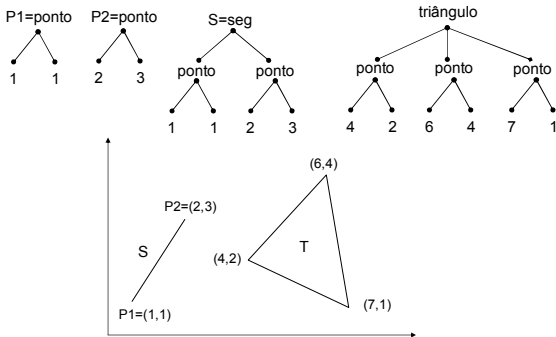
## Estruturas

- ❑ Por exemplo, o triângulo pode ser representado como
  - `triângulo(ponto(2,4),ponto(3,6),ponto(4,2))`



14

## Estruturas



15

## Predicados para Verificação dos Tipos de Termos

Predicado	É verdadeiro se:
<code>var(X)</code>	X é uma variável não instanciada
<code>nonvar(X)</code>	X não é uma variável ou X é uma variável instanciada
<code>atom(X)</code>	X é um átomo
<code>integer(X)</code>	X é um inteiro
<code>float(X)</code>	X é um número real
<code>atomic(X)</code>	X é uma constante (átomo ou número)
<code>compound(X)</code>	X é uma estrutura

18

## Predicados para Verificação dos Tipos de Termos

```

?- var(Z), Z = 2.
Z = 2
?- Z = 2, var(Z).
no
?- integer(Z), Z = 2.
no
?- Z = 2, integer(Z), nonvar(Z).
Z = 2
?- atom(3.14).
no
?- atomic(3.14).
yes
?- atom(==>).
yes
?- atom(p(1)).
no
?- compound(2+X).
yes
  
```

19

## Unificação de Termos

- ❑ Dois termos unificam (*matching*) se:
  - Eles são idênticos ou
  - As variáveis em ambos os termos podem ser instanciadas a objetos de maneira que após a substituição das variáveis por esses objetos os termos se tornam idênticos
- ❑ Por exemplo, há unificação entre os termos
  - `data(D,M,2003)` e `data(D1,maio,A)`
  - instanciando `D = D1`, `M = maio`, `A = 2003`

20

## Unificação de Termos

```
?- data(D,M,2003) = data(D1,maio,A),
   data(D,M,2003) = data(15,maio,A1).
D = 15
M = maio
D1 = 15
A = 2003
A1 = 2003

?- triângulo = triângulo, ponto(1,1) = X,
   A = ponto(4,Y), ponto(2,3) = ponto(2,Z).
X = ponto(1,1)
A = ponto(4,_G652)
Y = _G652
Z = 3
```

21

## Unificação de Termos

- Por outro lado, não há unificação entre os termos
  - `data(D,M,2003)` e `data(D1,M1,1948)`
  - `data(X,Y,Z)` e `ponto(X,Y,Z)`
- A unificação é um processo que toma dois termos e verifica se eles unificam
  - Se os termos não unificam, o processo falha (e as variáveis não se tornam instanciadas)
  - Se os termos unificam, o processo tem sucesso e também instancia as variáveis em ambos os termos para os valores que os tornam idênticos

22

## Unificação de Termos

- As regras que regem se dois termos **S** e **T** unificam são:
  - se **S** e **T** são constantes, então **S** e **T** unificam somente se são o mesmo objeto
  - se **S** for uma variável e **T** for qualquer termo, então unificam e **S** é instanciado para **T**
  - se **S** e **T** são estruturas, elas unificam somente se
    - ❖ **S** e **T** têm o mesmo functor principal e
    - ❖ todos seus componentes correspondentes unificam

23

## Comparação de Termos

Operadores Relacionais	
$X = Y$	X unifica com Y que é verdadeiro quando dois termos são o mesmo. Entretanto, se um dos termos é uma variável, o operador = causa a instanciação da variável porque o operador causa unificação
$X \neq Y$	X não unifica com Y que é o complemento de $X=Y$
$X == Y$	X é literalmente igual a Y (igualdade literal), que é verdadeiro se os termos X e Y são idênticos, ou seja, eles têm a mesma estrutura e todos os componentes correspondentes são os mesmos, incluindo o nome das variáveis
$X \neq Y$	X não é literalmente igual a Y que é o complemento de $X==Y$
$X @< Y$	X precede Y
$X @> Y$	Y precede X
$X @=< Y$	X precede ou é igual a Y
$X @>= Y$	Y precede ou é igual a X

24

## Comparação de Termos

```
?- f(a,b) == f(a,b).
yes
?- f(a,b) == f(a,X).
no
?- f(a,X) == f(a,Y).
no
?- X == X.
yes
?- X == Y.
no
?- X \== Y.
yes
?- X \= Y.
no
?- g(X,f(a,Y)) == g(X,f(a,Y)).
yes
```

25

## Precedência de Termos

- A precedência entre termos simples é determinado para ordem alfabética ou numérica
- Variável livres @< números @< átomos @< estruturas
- Uma estrutura precede outra se o functor da primeira tem menor aridade que o da segunda
  - Se duas estruturas têm mesma aridade, a primeira precede a segunda se seu functor é menor que o da outra
  - Se duas estruturas têm mesma aridade e funtores iguais, então a precedência é definida (da esquerda para a direita) pelos funtores dos seus componentes

26



## Exemplo: Macaco & Banana

- Quais os movimentos permitidos que alteram o mundo de um estado para outro?
  - Pegar a banana
  - Subir na caixa
  - Empurrar a caixa
  - Caminhar no chão da sala
- Nem todos os movimentos são possíveis em cada estado do mundo
  - 'pegar a banana' somente é possível se o macaco está acima da caixa diretamente abaixo da banana e o macaco ainda não tem a banana
- Vamos formalizar em Prolog usando a relação move
  - `move(Estado1, Movimento, Estado2)` onde Estado1 é o estado antes do movimento, Movimento é o movimento executado e Estado2 é o estado após o movimento

33

## Exemplo: Macaco & Banana

- O movimento 'pegar a banana' com sua pré-condição no estado antes do movimento pode ser definido por:
 

```
move(estado(no_centro, acima_caixa, no_centro, não_tem),
    pegar_banana,
    estado(no_centro, acima_caixa, no_centro, tem) ).
```

  - Este fato diz que após o movimento o macaco tem a banana e ele permanece acima da caixa no meio da sala
- Vamos expressar o fato que o macaco no chão pode caminhar de qualquer posição horizontal Pos1 para qualquer posição Pos2
 

```
move(estado(Pos1, no_chão, Caixa, Banana),
    caminhar(Pos1, Pos2),
    estado(Pos2, no_chão, Caixa, Banana) ).
```
- De maneira similar, os movimentos 'empurrar' e 'subir' podem ser especificados

34

## Exemplo: Macaco & Banana

- A pergunta principal que nosso programa deve responder é: O macaco consegue, a partir de um estado inicial **Estado**, pegar a banana?
- Isto pode ser formulado usando o predicado `consegue/1` que pode ser formulado baseado em duas observações:
  - Para qualquer estado no qual o macaco já tem a banana, o predicado `consegue/1` certamente deve ser verdadeiro; nenhum movimento é necessário
    - ♦ `consegue(estado( _, _, _, tem) ).`
  - Nos demais casos, um ou mais movimentos são necessários; o macaco pode obter a banana em qualquer estado Estado1 se há algum movimento de Estado1 para algum estado Estado2 tal que o macaco consegue pegar a banana no Estado2 (em zero ou mais movimentos)
    - ♦ `consegue(Estado1) :-`  
`move(Estado1, Movimento, Estado2),`  
`consegue(Estado2) .`

35

## Exemplo: Macaco & Banana

```
move(estado(no_centro, acima_caixa, no_centro, não_tem), % antes de mover
    pegar_banana, % pega banana
    estado(no_centro, acima_caixa, no_centro, tem) ). % depois de mover
move(estado(P, no_chão, P, Banana),
    subir, % subir na caixa
    estado(P, acima_caixa, P, Banana) ).
move(estado(P1, no_chão, P1, Banana),
    empurrar(P1, P2), % empurrar caixa de P1 para P2
    estado(P2, no_chão, P2, Banana) ).
move(estado(P1, no_chão, Caixa, Banana),
    caminhar(P1, P2), % caminhar de P1 para P2
    estado(P2, no_chão, Caixa, Banana) ).

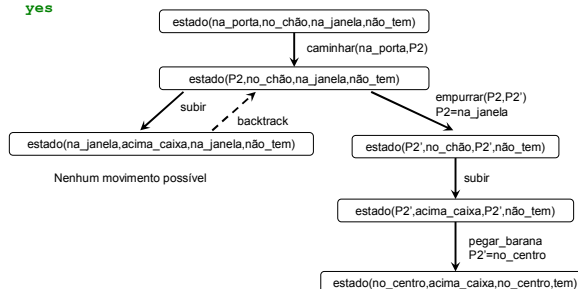
consegue(estado( _, _, _, tem) ). % macaco já tem banana

consegue(Estado1) :- % movimentar e tentar conseguir
    move(Estado1, Movimento, Estado2), % a banana
    consegue(Estado2) .
```

36

## Exemplo: Macaco & Banana

```
?- consegue(estado(na_porta, no_chão, na_janela, não_tem) ).
yes
```



37

## Listas

- Lista é uma das estruturas mais simples em Prolog, muito comum em programação não numérica
- Ela é uma seqüência ordenada de elementos
- Uma lista pode ter qualquer comprimento
- Por exemplo uma lista de elementos tais como **ana, ténis, pedro** pode ser escrita em Prolog como:
  - `[ana, ténis, pedro]`

38

## Listas

- O uso de colchetes é apenas uma melhoria da notação, pois internamente listas são representadas como árvores, assim como todos objetos estruturados em Prolog
- Para entender a representação Prolog de listas, é necessário considerar dois casos
  - A lista é vazia, escrita como `[]` em Prolog
  - Uma lista (não vazia) consiste:
    - ❖ no primeiro item, chamado **cabeça** (*head*) da lista
    - ❖ na parte restante da lista, chamada **cauda** (*tail*)

39

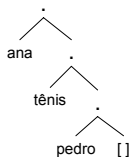
## Listas

- No exemplo `[ana, tênis, pedro]`
  - **ana** é a Cabeça da lista
  - **[tênis, pedro]** é a Cauda da lista
- A cabeça de uma lista pode ser qualquer objeto (inclusive uma lista); a cauda tem que ser uma lista
- A Cabeça e a Cauda são então combinadas em uma estrutura pelo functor especial `.`
  - `.(Cabeça, Cauda)`
- Como a Cauda é uma lista, ela é vazia ou ela tem sua própria cabeça e sua cauda

40

## Listas

- Assim, para representar listas de qualquer comprimento, nenhum princípio adicional é necessário
- O exemplo `[ana, tênis, pedro]` é representando como o termo:
  - `.(ana, .(tênis, .(pedro, [])))`
- O programador pode escolher ambas notações



41

## Listas

```

?- Lista1 = [a,b,c],
   Lista2 = .(a,.(b,.(c,[]))).
Lista1 = [a, b, c]
Lista2 = [a, b, c]

?- Hobbies1 = .(tênis, .(música,[])),
   Hobbies2 = [esqui, comida],
   L = [ana,Hobbies1,pedro,Hobbies2].
Hobbies1 = [tênis,música]
Hobbies2 = [esqui,comida]
L = [ana, [tênis,música], pedro, [esqui,comida]]
    
```

42

## Listas

- Em geral, é comum tratar a cauda como um objeto simples
- Por exemplo, `L = [a,b,c]` pode ser escrito como
  - Cauda = `[b,c]`
  - `L = .(a,Cauda)`
- Para expressar isso, Prolog fornece uma notação alternativa, a **barra vertical**, que separa a cabeça da cauda
  - `L = [a | Cauda]`
- A notação é geral por permitir qualquer número de elementos seja seguido por `|` e o restante da lista:
  - `[a,b,c] = [a | [b,c]] = [a,b | [c]] = [a,b,c | []]`

43

## Unificação em Listas

Lista1	Lista2	Lista1 = Lista2
<code>[mesa]</code>	<code>[X Y]</code>	X=mesa Y=[]
<code>[a,b,c,d]</code>	<code>[X,Y Z]</code>	X=a Y=b Z=[c,d]
<code>[[ana,Y] Z]</code>	<code>[[X,foi],[ao,cinema]]</code>	X=ana Y=foi Z=[[ao,cinema]]
<code>[ano,bissexto]</code>	<code>[X,Y Z]</code>	X=ano Y=bissexto Z=[]
<code>[ano,bissexto]</code>	<code>[X,Y,Z]</code>	Não unifica

44

## Operações em Listas

- ❑ Frequentemente, é necessário realizar operações em listas, por exemplo, buscar um elemento que faz parte de uma lista
- ❑ Para isso, a recursão é o recurso mais amplamente empregado
- ❑ Para verificar se um nome está na lista, é preciso verificar se ele está na cabeça ou se ele está na cauda da lista
- ❑ Se o final da lista for atingido, o nome não está na lista

45

## Predicado de Pertinência

- ❑ Inicialmente, é necessário definir o nome do predicado que verifica se um elemento **X** pertence ou não a uma lista **Y**, por exemplo, **pertence(X,Y)**
- ❑ A primeira condição especifica que um elemento **X** pertence à lista se ele está na cabeça dela. Isto é indicado como:
  - **pertence(X,[X|Z]).**
- ❑ A segunda condição especifica que um elemento **X** pertence à lista se ele pertencer à sua cauda. Isto pode ser indicado como:
  - **pertence(X,[W|Z]) :- pertence(X,Z).**

46

## Predicado de Pertinência

- ❑ Sempre que um programa recursivo é definido, deve-se procurar pelas condições limites (ou condições de parada) e pelo caso(s) recursivo(s):
  - **pertence(Elemento, [Elemento|Cauda]).**
  - **pertence(Elemento, [Cabeça|Cauda]) :- pertence(Elemento, Cauda).**
- ❑ Após a definição do programa, é possível interrogá-lo. Por exemplo,  
**?- pertence(a, [a,b,c]).**  
**yes**

47

## Predicado de Pertinência

- ❑ **?- pertence(d, [a,b,c]).**  
**no**
- ❑ **?- pertence(X, [a,b,c]).**  
**X = a ;**  
**X = b ;**  
**X = c ;**  
**no**
- ❑ Entretanto, se as perguntas forem:
  - **?- pertence(a, X).**
  - **?- pertence(X, Y).**
- ❑ deve-se observar que cada uma delas tem infinitas respostas, pois existem infinitas listas que validam essas perguntas para o programa **pertence/2**

48

## Modo de Chamada de Predicados

- ❑ Para documentar como um predicado deve ser chamado, utiliza-se a notação (como comentário no programa):
  - **+** o argumento é de entrada (deve estar instanciado)
  - **-** o argumento é de saída (não deve estar instanciado)
  - **?** o argumento é de entrada e saída (pode ou não estar instanciado)
- ❑ O predicado **pertence/2** documentado com o modo de chamada é:

```
% pertence(?Elemento, +Lista)
pertence(E, [E|_]).
pertence(E, [_|Cauda]) :-
    pertence(E, Cauda).
```

49

## Exercícios

- ❑ Definir predicado **último** que encontra o último elemento de uma lista
  - O último elemento de uma lista que tenha somente um elemento é o próprio elemento
  - O último elemento de uma lista que tenha mais de um elemento é o último elemento da cauda
- ❑ Concatenar duas listas, formando uma terceira
  - Se o primeiro argumento é a lista vazia, então o segundo e terceiro argumentos devem ser o mesmo
  - Se o primeiro argumento é a lista não-vazia, então ela tem uma cabeça e uma cauda da forma **[X|L1]**; concatenar **[X|L1]** com uma segunda lista **L2** resulta na lista **[X|L3]**, onde **L3** é a concatenação de **L1** e **L2**
- ❑ Defina uma nova versão do predicado **último** utilizando a concatenação de listas

50



## Último Elemento de uma Lista

- ❑ O último elemento de uma lista que tenha somente um elemento é o próprio elemento `ultimo(Elemento, [Elemento])`.
- ❑ O último elemento de uma lista que tenha mais de um elemento é o último elemento da cauda `ultimo(Elemento, [Cabeca|Cauda]) :- ultimo(Elemento,Cauda)`.
- ❑ Programa completo:

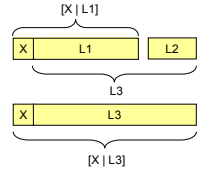
```
% ultimo(?Elemento, +Lista)
ultimo(Elemento, [Elemento]).
ultimo(Elemento, [Cabeca|Cauda]) :-
    ultimo(Elemento,Cauda).
```

51

## Concatenar Listas

- ❑ Se o primeiro argumento é a lista vazia, então o segundo e terceiro argumentos devem ser o mesmo `concatenar([],L,L)`.
- ❑ Se o primeiro argumento é a lista não-vazia, então ela tem uma cabeça e uma cauda da forma `[X|L1]`; concatenar `[X|L1]` com uma segunda lista `L2` resulta na lista `[X|L3]`, onde `L3` é a concatenação de `L1` e `L2` `concatenar([X|L1],L2,[X|L3]) :- concatenar(L1,L2,L3)`.
- ❑ Programa completo:

```
% concatenar(?/+L1, ?/?L2, +/?L)
concatenar([],L,L).
concatenar([X|L1],L2,[X|L3]) :-
    concatenar(L1,L2,L3).
```



52

## Exemplo: Macaco & Banana

```
move(estado(no_centro,acima_caixa,no_centro,não_tem), % antes de mover
    pegar_banana, % pega banana
    estado(no_centro,acima_caixa,no_centro,tem)) . % depois de mover
move(estado(P,no_chão,P,Banana),
    subir, % subir na caixa
    estado(P,acima_caixa,P,Banana)) .
move(estado(P1,no_chão,P1,Banana),
    empurrar(P1,P2), % empurrar caixa de P1 para P2
    estado(P2,no_chão,P2,Banana)) .
move(estado(P1,no_chão,Caixa,Banana),
    caminhar(P1,P2), % caminhar de P1 para P2
    estado(P2,no_chão,Caixa,Banana)) .

consegue(estado(_,_,tem), []). % macaco já tem banana
consegue(Estado1,[Movimento|Resto]) :-
    move(Estado1,Movimento,Estado2), % a banana
    consegue(Estado2,Resto).

?- consegue(estado(na_porta,no_chão,na_janela,não_tem),X).
X = [caminhar(na_porta,na_janela), empurrar(na_janela, no_centro), subir,
    pegar_banana]
```

53

## Operadores

- ❑ Em várias situações é conveniente escrever funtores como **operadores**
- ❑ Esta é uma forma sintática que facilita a leitura de estruturas
- ❑ Por exemplo, numa expressão aritmética como  $2*a+b*c$ 
  - `+` e `*` são operadores
  - `2`, `a`, `b` são argumentos
- ❑ Esta expressão aritmética na sintaxe normal de estruturas (termos) é:
  - `+(*(2,a),*(b,c))`
- ❑ Entretanto, é importante lembrar que os operadores não “realizam” nenhuma aritmética: o  $3+4$  não é a mesma coisa que 7
  - O termo  $3+4$  é representado como `+(3,4)`, que é uma estrutura
  - Assim, escrever  $3+4$  é equivalente a escrever `+(3,4)`

54

## Operadores

- ❑ De forma que Prolog entenda apropriadamente expressões tais como  $a+b*c$ , Prolog deve conhecer que `*` tem precedência sobre `+`
- ❑ Assim, a precedência de operadores decide qual é a correta interpretação de expressões
- ❑ Prolog permite a declaração de novos operadores, por exemplo, podemos definir os átomos **tem** e **suporta** como operadores e escrever no programa fatos tais como:
  - `pedro tem informação.`
  - `chão suporta mesa.`
- ❑ Estes fatos são equivalentes à:
  - `tem(pedro,informação).`
  - `suporta(chão,mesa).`

55

## Operadores

- ❑ A definição de um operador deve aparecer através de uma diretiva antes da expressão contendo o operador
- ❑ No exemplo anterior, o operador **tem** pode ser definido através da diretiva
  - `:- op(600,xfx,tem).`
  - Isso informa Prolog que desejamos usar **tem** como operador, cuja precedência é 600 e seu tipo é `'xfx'`, que é um tipo de operador infixo
  - A forma `'xfx'` sugere que o operador, denotado pela letra `'f'` está entre dois argumentos, denotados por `'x'`
- ❑ Novamente, a definição de operadores não especifica nenhuma operação ou ação e são usados apenas para combinar objetos em estruturas

56

# Operadores

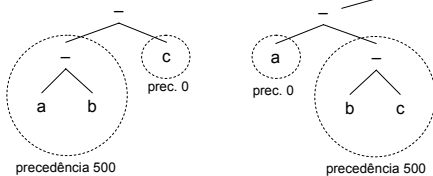
- Os nomes dos operadores devem ser átomos
- A precedência varia em algum intervalo, dependendo da implementação Prolog (1-1200)
- Há 3 grupos tipos de operadores, indicados pelos especificadores tal como `xfx`:
  - (1) Operadores infixos de três tipos: `xfx`, `xfy`, `yfx`
  - (2) Operadores pré-fixos de dois tipos: `fx`, `fy`
  - (3) Operadores pós-fixos de dois tipos: `xf`, `yf`

# Operadores

- Os especificadores são escolhidos para refletir a estrutura da expressão, onde:
  - `f` representa o operador
  - `x` representa um argumento cuja precedência é estritamente menor que a do operador
  - `y` representa um argumento cuja precedência é menor ou igual à do operador
- A precedência de um argumento colocado entre parênteses ou de um objeto não estruturado é zero
- Se um argumento é uma estrutura, então sua precedência é igual à precedência de seu functor principal

# Operadores

- Por exemplo `a-b-c` é entendido como `(a-b)-c` e não como `a-(b-c)`
  - Assumindo que `-` tenha precedência 500
  - O operador `-` tem que ser definido como `yfx`



Interpretação inválida porque a precedência de b-c não é menor do que a precedência de -

# Operadores

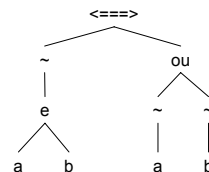
- Exemplo de alguns operadores pré-definidos
  - `-op (1200, xfx, [-->, :-]) .`
  - `-op (1200, fx, [:-, ?-]) .`
  - `-op (1100, xfy, [';', '|']) .`
  - `-op (1050, xfy, ->) .`
  - `-op (1000, xfy, ',') .`
  - `-op (954, xfy, \) .`
  - `-op (900, fy, ['\+', not]) .`
  - `-op (700, xfx, [is, <, =, =. ., @=, =:=, <=, =\=, >, >=, @<, @=<, @>, \=, \==]) .`
  - `-op (600, xfy, :) .`
  - `-op (500, yfx, [+ , -]) .`
  - `-op (500, fx, [+ , -]) .`
  - `-op (400, yfx, [* , / , // , mod]) .`
  - `-op (200, xfx, **) .`
  - `-op (200, xfy, ^) .`

# Exemplo

- Assumindo as diretrizes
  - `-op (800, xfx, <====>) .`
  - `-op (700, xfy, ou) .`
  - `-op (600, xfy, e) .`
  - `-op (500, fy, ~) .`
- Como é interpretada a expressão `~(a e b) <====> ~a ou ~b`

# Exemplo

- `~(a e b) <====> ~a ou ~b` é interpretado como `<====>( ~(e(a, b)), ou( ~(a), ~(b))`



## Predicados para (De)composição Termos

- **atom\_chars(A,L)**: converte um átomo A em uma lista L composta pelos caracteres (e vice-versa)
  - `?- atom_chars(laranja,X).`
  - `X = [l,a,r,a,n,j,a]`
  - `?- atom_chars(Z,[m,a,c,a,c,o]).`
  - `Z = macaco`
- **number\_chars(A,L)**: Similar ao predicado atom\_chars, mas para números
  - `?- number_chars(123.5,X).`
  - `X = ['1', '2', '3', '.', '5']`
  - `?- number_chars(X,['1', '2', '3']).`
  - `Z = 123`
- Algumas implementações Prolog fornecem o predicado **name(A,L)** que tem a mesma funcionalidade que atom\_chars(A,L) e number\_chars(A,L) combinados

63

## Predicados para (De)composição Termos

- **Termo =.. L**: é verdadeiro se L é uma lista contendo o functor principal de Termo, seguido pelos seus argumentos
  - `=..` é lido como **univ**
- **functor(Termo,F,N)**: é verdadeiro se Termo é uma estrutura com functor F e aridade N
- **arg(N,Termo,A)**: é verdadeiro se o N-ésimo argumento do Termo é A

64

## Predicados para (De)composição Termos

```
?- f(a,b) =.. L.  
L = [f,a,b]  
?- T =.. [retângulo,3,5].  
T = retângulo(3,5)  
?- Z =.. [p,X,f(X,Y)].  
Z = p(X,f(X,Y))  
?- (a+b) =.. [F,X,Y].  
F = +, X = a, Y = b  
?- [a,b,c,d] =.. L.  
L = [.,a,[b,c,d]]  
?- [a,b,c,d] =.. [X|Y].  
X = ., Y = [a,[b,c,d]]  
?- X =.. [a,b,c,d].  
X = a(b,c,d)  
?- X =.. [concatenar,[a,b],[c],[a,b,c]].  
X = concatenar([a,b],[c],[a,b,c])
```

65

## Predicados para (De)composição Termos

```
?- functor(t(f(x),x,t),F,Aridade).  
F = t, Aridade = 3  
?- functor(a+b,F,N).  
F = +, N = 2  
?- functor(f(a,b,g(Z)),F,N).  
Z = _G309, F = f, N = 3  
?- functor([a,b,c],F,N).  
F = ., N = 2  
?- functor(laranja,F,N).  
F = laranja, N = 0  
?- functor([a,b,c],',',3).  
no  
?- functor([a,b,c],a,Z).  
no
```

66

## Predicados para (De)composição Termos

```
?- arg(1,t(f(x),y,t),Arg).  
Arg = f(x)  
?- arg(2,t(f(x),y,t),Arg).  
Arg = y  
?- arg(1,a+(b+c),Arg).  
Arg = a  
?- arg(2,[a,b,c],X).  
X = [b,c]  
?- arg(1,a+(b+c),b).  
no  
?- functor(D,data,3), arg(1,D,29),  
arg(2,D,janeiro), arg(3,D,2004).  
D = data(29,janeiro,2004)
```

67

## Predicados para (De)composição Termos

- Termos construídos com o predicado `=..`, podem ser usados como metas
- A vantagem é que o programa pode modificar-se a si próprio durante a execução, gerando e executando metas de formas que não foram necessariamente previstas quando o programa foi escrito
- O seguinte fragmento ilustra essa idéia
  - obter(Functor),  
calcular(ListaArgumentos),  
Meta =.. [Functor | ListaArgumentos],  
Meta.
  - obter/1 e calcular/1 são predicados definidos pelo usuário para obter os componentes da meta a ser construída; a meta é então construída por `=..` e disparada sua execução simplesmente através de seu nome, Meta

68

## Predicados para (De)composição Termos

- Algumas implementações Prolog exigem que todas as metas sejam átomos ou estruturas com um átomo como functor principal
- Assim, uma variável, mesmo que instanciada, não é sintaticamente aceita como uma meta
- O problema é solucionado através do predicado **call/1**, assim o fragmento anterior torna-se:
  - `obter(Functor),`
  - `calcular(ListaArgumentos),`
  - `Meta =.. [Functor | ListaArgumentos],`
  - `call(Meta).`

69

## Semântica de Programas Prolog

- Considere a cláusula onde P, Q e R são termos:
  - `P :- Q, R.`
- Significado Declarativo:
  - P é verdadeiro se Q e R são verdadeiros
  - P segue (conseqüência) de Q e R
  - De (a partir de) Q e R segue P
- Significado Procedural:
  - Para resolver o problema P, resolva primeiro sub-problema Q e então o sub-problema R
  - Para satisfazer P, primeiro satisfaça Q e então R
- A diferença entre os significados declarativo e procedural é que o último não apenas define as relações lógicas entre a cabeça da cláusula e as condições no corpo, mas também a **ordem** na qual as condições são executadas

70

## Semântica de Programas Prolog

- O significado declarativo determina quando uma dada condição é verdadeira e, se for, para quais valores das variáveis ela é verdadeira
- O significado procedural determina como Prolog responde perguntas

71

## Significado Declarativo

- Em geral, uma meta em Prolog é uma lista de condições separadas por vírgulas que é verdadeira se todas as condições na lista são verdadeiras para uma determinada instanciação de variáveis
- A vírgula denota conjunção (**e**): todas as condições devem ser verdadeiras:
  - `X, Y` neste exemplo `X e Y` devem ser ambos verdadeiros para `X, Y` ser verdadeiro
- O ponto-e-vírgula denota disjunção (**ou**): qualquer uma das condições em uma disjunção tem que ser verdadeira
  - `X;Y` neste exemplo basta que `X` (ou `Y`) seja verdadeiro para `X;Y` ser verdadeiro
- O operador **!** denota a negação (**não**): é verdadeiro se o que está sendo negado não puder ser provado por Prolog
  - `!X` é verdadeiro se `X` falha
  - Na sintaxe de Edinburgh o operador **!** é denotado por **not**
- O predicado **true/0** sempre é verdadeiro
- O predicado **fail/0** sempre falha

72

## Significado Declarativo

- A cláusula
  - `P :- Q, R.`
- é lida como: P é verdade se Q é verdade ou R é verdade. É equivalente às duas cláusulas:
  - `P :- Q.`
  - `P :- R.`
- A conjunção (,) tem precedência sobre a disjunção (;), assim a cláusula:
  - `P :- Q, R; S, T, U.`
- é interpretada como:
  - `P :- (Q, R); (S, T, U).`
- e tem o mesmo significado que:
  - `P :- Q, R.`
  - `P :- S, T, U.`

73

## Significado Procedural

```
grande(urso).           % Cláusula 1
grande(elefante).       % Cláusula 2
pequeno(gato).          % Cláusula 3
marrom(urso).           % Cláusula 4
preto(gato).            % Cláusula 5
cinza(elefante).        % Cláusula 6
escuro(Z) :-            % Cláusula 7
    preto(Z).
escuro(Z) :-            % Cláusula 8
    marrom(Z).

?- escuro(X), grande(X).
```

74

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 1)  
Pergunta inicial:  
escuro(X), grande(X).

75

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 2)  
Procure de cima para baixo por  
uma cláusula cuja cabeça  
unifique com a primeira  
condição da pergunta  
escuro(X)

76

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 2)  
Procure de cima para baixo por  
uma cláusula cuja cabeça  
unifique com a primeira  
condição da pergunta  
escuro(X).  
Cláusula 7 encontrada  
escuro(Z) :- preto(Z).  
Troque a primeira condição pelo  
corpo instanciado da cláusula  
7, obtendo a nova lista de  
condições:  
preto(X), grande(X).

77

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 3)  
Nova meta: preto(X), grande(X).  
Procure por uma cláusula que  
unifique com preto(X)  
Cláusula 5 encontrada  
preto(gato)  
Esta cláusula não tem corpo,  
assim a lista de condições  
depois de instanciada toma-  
se:  
grande(gato).  
uma vez que já se provou  
preto(gato)  
X instancia com gato

78

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 4)  
Nova meta: grande(gato).  
Procure por uma cláusula que  
unifique com grande(gato).  
Nenhuma cláusula é encontrada.  
Volte (backtrack) para o Passo 3,  
desfazendo a instanciadação  
X=gato  
Novamente a meta é  
preto(X), grande(X).

79

## Significado Procedural

```
grande(urso).      % Cláusula 1
grande(elefante).  % Cláusula 2
pequeno(gato).    % Cláusula 3
marrom(urso).     % Cláusula 4
preto(gato).      % Cláusula 5
cinza(elefante).  % Cláusula 6
escuro(Z) :-      % Cláusula 7
  preto(Z).
escuro(Z) :-      % Cláusula 8
  marrom(Z).
```

?- escuro(X), grande(X).

(Passo 4)  
meta: preto(X), grande(X).  
Continue procurando após a  
Cláusula 5. Nenhuma cláusula é  
encontrada. Volte (backtrack)  
ao Passo 2 e continue  
procurando após a cláusula 7.  
Cláusula 8 encontrada:  
escuro(Z) :- marrom(Z).  
Troque a primeira condição pelo  
corpo instanciado da cláusula  
8, obtendo a nova lista de  
condições:  
marrom(X), grande(X).

80

## Significado Procedural

```
grande(urso).           % Cláusula 1
grande(elefante).      % Cláusula 2
pequeno(gato).         % Cláusula 3
marrom(urso).          % Cláusula 4
preto(gato).           % Cláusula 5
cinza(elefante).      % Cláusula 6
escuro(Z) :-           % Cláusula 7
    preto(Z).
escuro(Z) :-           % Cláusula 8
    marrom(Z).

?- escuro(X), grande(X).
```

(Passo 5)  
meta: marrom(X), grande(X).

Procure por uma cláusula que unifique com marrom(X)  
Cláusula 4 encontrada  
marrom(urso)  
Esta cláusula não tem corpo, assim a lista de condições depois de instanciada torna-se: grande(urso).  
uma vez que já se provou marrom(urso)  
X instancia com urso

81

## Significado Procedural

```
grande(urso).           % Cláusula 1
grande(elefante).      % Cláusula 2
pequeno(gato).         % Cláusula 3
marrom(urso).          % Cláusula 4
preto(gato).           % Cláusula 5
cinza(elefante).      % Cláusula 6
escuro(Z) :-           % Cláusula 7
    preto(Z).
escuro(Z) :-           % Cláusula 8
    marrom(Z).

?- escuro(X), grande(X).
```

(Passo 6)  
meta: grande(urso).

Procure por uma cláusula que unifique com grande(urso)  
Cláusula 1 encontrada  
grande(urso)  
Esta cláusula não tem corpo, assim a lista de condições torna-se vazia.  
Isto indica um término com sucesso e a instancição correspondente é  
X = urso

82

## Ordem das Cláusulas

- ❑ No exemplo do “Macaco & Banana”, as cláusulas sobre a relação **move** foram ordenadas como: pegar a banana, subir na caixa, empurrar a caixa e caminhar
- ❑ Estas cláusulas dizem que pegar é possível, subir é possível, etc
- ❑ De acordo com o significado procedural de Prolog, a ordem das cláusulas indica que o macaco prefere pegar a subir, subir a empurrar, etc.
- ❑ Esta ordem, na realidade, ajuda o macaco a resolver o problema
- ❑ Todavia, o que aconteceria se a ordem fosse diferente? Por exemplo, vamos assumir que a cláusula sobre ‘caminhar’ apareça em primeiro lugar

83

## Macaco & Banana (Original)

```
move(estado(no_centro,acima_caixa,no_centro,não_tem), % antes de mover
    pegar_banana, % pega banana
    estado(no_centro,acima_caixa,no_centro,tem) ). % depois de mover
move(estado(P,no_chão,P,Banana),
    subir, % subir na caixa
    estado(P,acima_caixa,P,Banana) ).
move(estado(P1,no_chão,P1,Banana),
    empurrar(P1,P2), % empurrar caixa de P1 para P2
    estado(P2,no_chão,P2,Banana) ).
move(estado(P1,no_chão,Caixa,Banana),
    caminhar(P1,P2), % caminhar de P1 para P2
    estado(P2,no_chão,Caixa,Banana) ).

consegue(estado(_,_,_ ,tem) ). % macaco já tem banana

consegue(Estado1) :- % movimentar e tentar conseguir
    move(Estado1,Movimento,Estado2), % a banana
    consegue(Estado2).
```

84

## Macaco & Banana (Ordem Alterada)

```
move(estado(P1,no_chão,Caixa,Banana),
    caminhar(P1,P2), % caminhar de P1 para P2
    estado(P2,no_chão,Caixa,Banana) ).
move(estado(no_centro,acima_caixa,no_centro,não_tem), % antes de mover
    pegar_banana, % pega banana
    estado(no_centro,acima_caixa,no_centro,tem) ). % depois de mover
move(estado(P,no_chão,P,Banana),
    subir, % subir na caixa
    estado(P,acima_caixa,P,Banana) ).
move(estado(P1,no_chão,P1,Banana),
    empurrar(P1,P2), % empurrar caixa de P1 para P2
    estado(P2,no_chão,P2,Banana) ).

consegue(estado(_,_,_ ,tem) ). % 1a cláusula de consegue/1

consegue(Estado1) :- % 2a cláusula de consegue/1
    move(Estado1,Movimento,Estado2),
    consegue(Estado2).
```

85

## Ordem das Cláusulas

- ❑ Vamos analisar a execução da versão com ordem alterada da pergunta ?- consegue(estado(na\_porta,no\_chão,na\_janela,não\_tem)).
- ❑ Que produz a seguinte execução (variáveis renomeadas apropriadamente):
  - (1) consegue(estado(na\_porta,no\_chão,na\_janela,não\_tem))
    - ↳ A segunda cláusula de consegue/1 é aplicada
  - (2) move(estado(na\_porta,no\_chão,na\_janela,não\_tem),M',S2'), consegue(S2')
    - ↳ Através do movimento **caminhar(na\_porta,P2')** temos:
      - (3) consegue(estado(P2',no\_chão,na\_janela,não\_tem))
        - ↳ A segunda cláusula de consegue/1 é aplicada novamente
      - (4) move(estado(P2',no\_chão,na\_janela,não\_tem),M'',S2''), consegue(S2'')
        - ↳ Agora ocorre a diferença: a primeira cláusula cuja cabeça unifica com a primeira meta acima é **caminhar** (e não **subir** como antes). A instancição é S2'' = estado(P2'',no\_chão,na\_janela,não\_tem), portanto a lista de metas se torna
    - (5) consegue(estado(P2'',no\_chão,na\_janela,não\_tem))
      - ↳ Novamente, a segunda cláusula de consegue/1 é aplicada
    - (6) move(estado(P2'',no\_chão,na\_janela,não\_tem),M''',S2'''), consegue(S2''')
      - ↳ Novamente, **caminhar** é tentado primeiro, produzindo
    - (7) consegue(estado(P2''',no\_chão,na\_janela,não\_tem))
- ❑ As metas (3), (5) e (7) são as mesmas, exceto por uma variável; e o sucesso de uma meta não depende do nome particular das variáveis envolvidas
- ❑ Assim a partir da meta (3) a execução não mostra progresso algum

86

## Ordem das Cláusulas

- ❑ Nota-se que a segunda cláusula de `consegue/1` e cláusula `caminhar de move/1` são usadas repetidamente
- ❑ O macaco caminha sem nunca tentar usar a caixa
- ❑ Como não há progresso, isso procede infinitamente: Prolog não percebe que não há sentido em continuar utilizando esta linha de raciocínio
- ❑ Este exemplo mostra Prolog tentando resolver um problema de modo que a solução nunca é encontrada, embora exista uma solução
- ❑ O programa está declarativamente correto, mas proceduralmente incorreto no sentido que ele não é capaz de produzir uma resposta à pergunta

87

## Pontos Importantes

- ❑ Objetos simples em Prolog são átomos, números e variáveis
- ❑ Objetos estruturados, ou estruturas, são utilizados para representar objetos que possuem vários componentes
- ❑ Estruturas são construídas através de funtores; cada functor é definido por seu nome e aridade
- ❑ O escopo léxico de uma variável é uma cláusula; assim o mesmo nome de variável em duas cláusulas significam duas variáveis diferentes

88

## Pontos Importantes

- ❑ Estruturas podem ser vistas como árvores; Prolog pode ser vista como uma linguagem para processamento de árvores
- ❑ A operação de unificação toma dois termos e tenta torná-los idênticos através da instanciação das variáveis em ambos os termos
- ❑ A ordem das cláusulas pode afetar a eficiência do programa; uma ordem indevida pode até mesmo causar chamadas recursivas infinitas

89

Slides baseados em:

Bratko, I.;  
*Prolog Programming for Artificial Intelligence*,  
3rd Edition, Pearson Education, 2001.

Clocksin, W.F.; Mellish, C.S.;  
*Programming in Prolog*,  
5th Edition, Springer-Verlag, 2003.

Programas Prolog para o  
Processamento de Listas e Aplicações,  
Monard, M.C & Nicoletti, M.C., ICMC-USP, 1993

Material elaborado por  
José Augusto Baranauskas  
2004

90