Sintaxe e Semântica de Programas Prolog

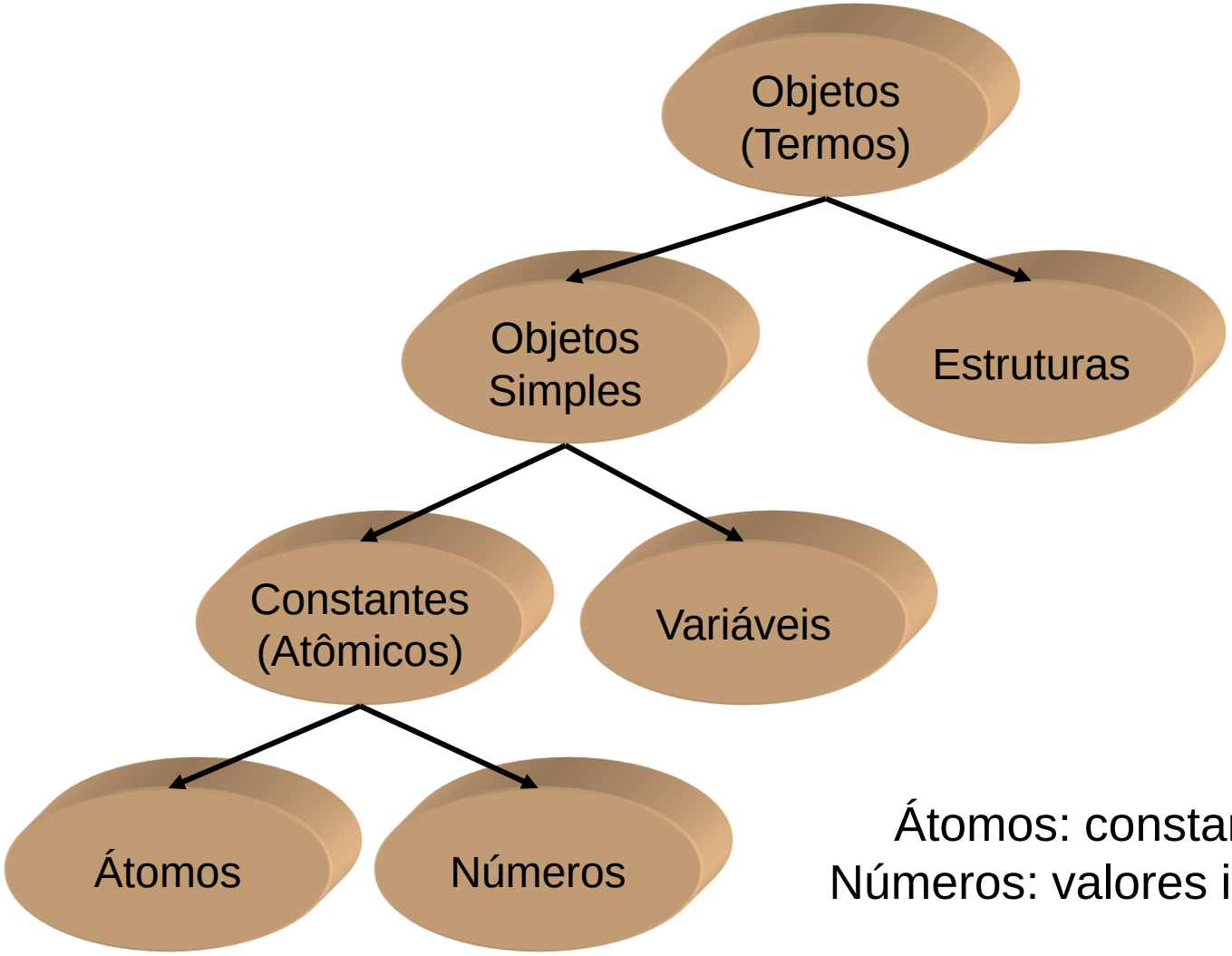
Inteligência Artificial

- Esta aula trata da 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 – FFCLRP-USP
Sala 228 – Bloco P2 – Fone (16) 602-4439

E-mail: augusto@ffclrp.usp.br
URL: http://www.fmrp.usp.br/augusto

Objetos em Prolog



Átomos: constantes textuais
Números: valores inteiros ou reais

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 apóstrofos: 'Abraão', 'América_do_Sul', 'América Latina'

Números

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

- Unificação: X pode ser idêntico a Y? Se sim, um referencia o outro

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`**
  - `?- X is +(1,2).`
  - **`X = 3`**

Números

- 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

- Cadeias de letras, dígitos e caracteres ‘_’ que determinam nomes de variável devem sempre começar com letra maiúscula ou com o caractere ‘_’, pois a letra minúscula inicia átomo
  - X, Resultado, Objeto3, Lista_Alunos, ListaCompras, _x25, _32

- Escopo de uma variável: dentro de uma mesma regra ou dentro de uma pergunta (vide exemplo do *slide* anterior)

Variáveis

- 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 tornar uma variável não-instanciada, através de seu mecanismo de inferência – ou seja, o programador não “libera” variável

Variável Anônima

- Quando uma variável aparece em uma única cláusula, é desnecessário usar um nome para ela
- Usa então uma variável **anônima**, que é escrita com um simples caracter '_'
- Por exemplo
  - tem_filhos(X) :- progenitor(X,Y).
- Para definir **tem_filhos**, o nome dos filhos(as) é desnecessário
- Logo, é exemplo ideal para a variável anônima:
  - tem_filhos(X) :- progenitor(X,_).

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 uma variável anônima é usada em uma pergunta, seu valor não é mostrado; logo, por exemplo, se queremos saber quem tem filhos, mas sem mostrar os nomes dos filhos, podemos perguntar:
  - **?-** progenitor(X,_).

Estruturas

- Objetos estruturados (ou simplesmente estruturas) são objetos de dados que têm vários componentes, de modo similar a um *struct* em C
- 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
- Além disso, cada componente de estrutura pode ser uma estrutura (“estrutura de estruturas”)

Estruturas

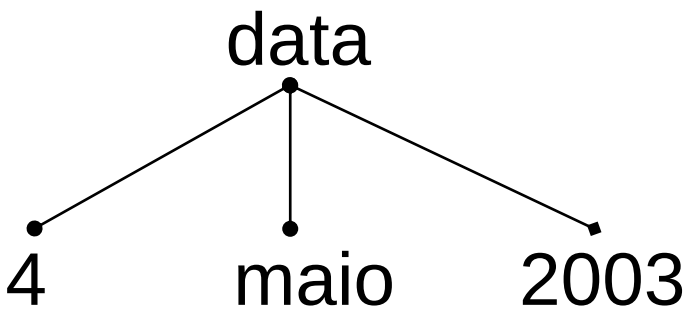
- De modo a combinar componentes em um simples objeto estrutura, deve-se escolher um **functor**
- Um functor para o exemplo da data seria **data**
- Então a data de 4 de maio de 2003 pode ser escrita como:
  - data(4,maio,2003)

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)

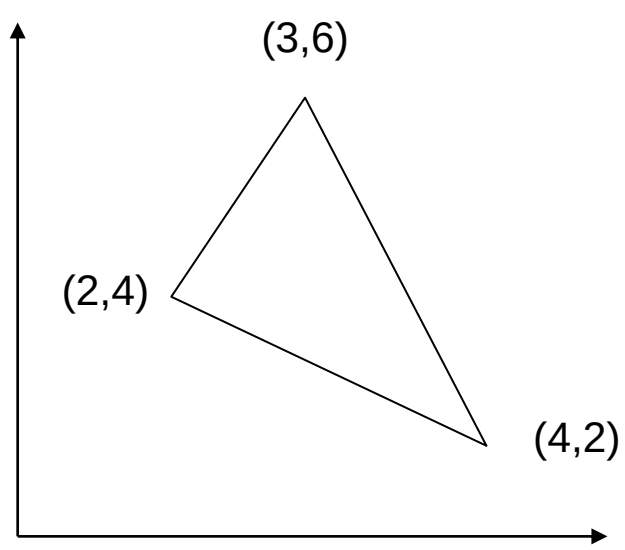
Estruturas

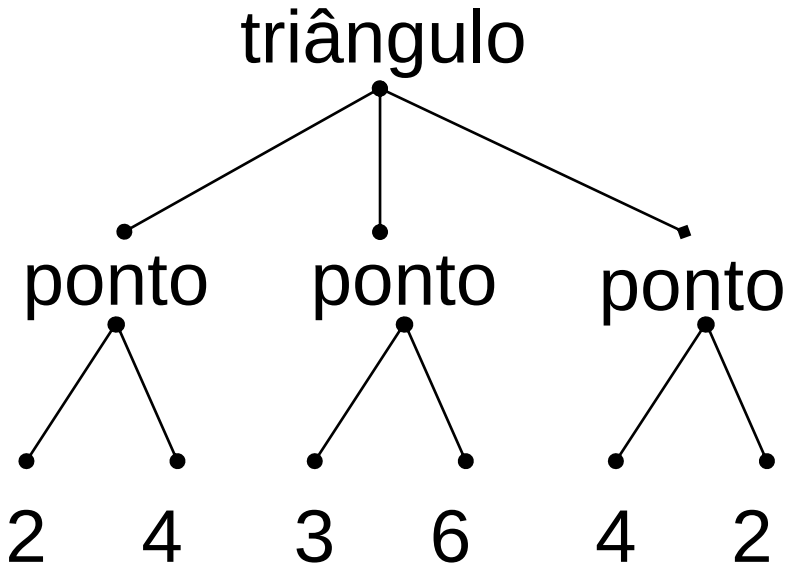
- Todos os objetos estruturados podem ser representados como árvores
- A raiz da árvore é o **functor** e os filhos da raiz são os componentes
- Para a estrutura data(4,maio,2003):



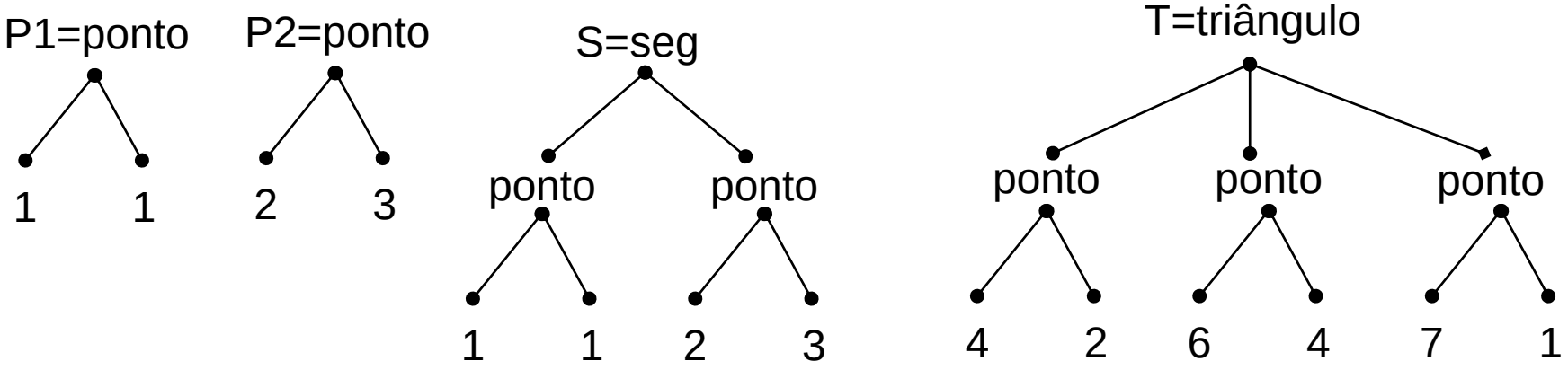
Estruturas

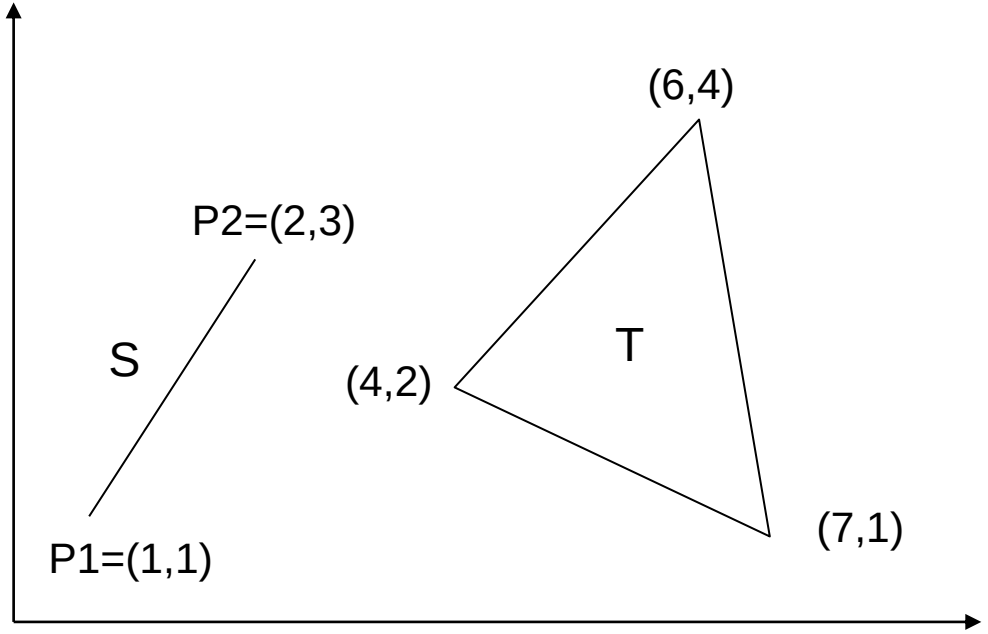
- Por exemplo, o triângulo pode ser representado como
  - triângulo(ponto(2,4),ponto(3,6),ponto(4,2))





Estruturas


P1=ponto
1
1
P2=ponto
2
3
S=seg
ponto
ponto
1
1
2
3
T=triângulo
ponto
ponto
ponto
4
2
6
4
7
1


(6,4)
P2=(2,3)
S
T
(4,2)
(7,1)
P1=(1,1)

Predicados para Verificação dos Tipos de Termos

Predicado	**É verdadeiro se:**
var(X)	X é uma variável livre (não-instanciada)
nonvar(X)	X não é uma variável ou X é uma variável instanciada
integer(X)	X é um inteiro
float(X)	X é um número real equivalente ao *double* do C
atomic(X)	X é uma constante (átomo ou número)
atom(X)	X é um átomo
compound(X)	X é uma estrutura

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
```

Unificação de Termos

- Dois termos unificam (*match*) 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

Unificação de Termos

```
?- data(D,M,2003) = data(D1,maio,A),
   data(D,M,2003) = data(15,maio,A1).
D = D1, D1 = 15,
M = maio,
A = A1, 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, Y),
Z = 3.
```

```
X = ponto(1,1)
A = ponto(4,_G652)
Y = _G652
Z = 3
```

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

Unificação de Termos

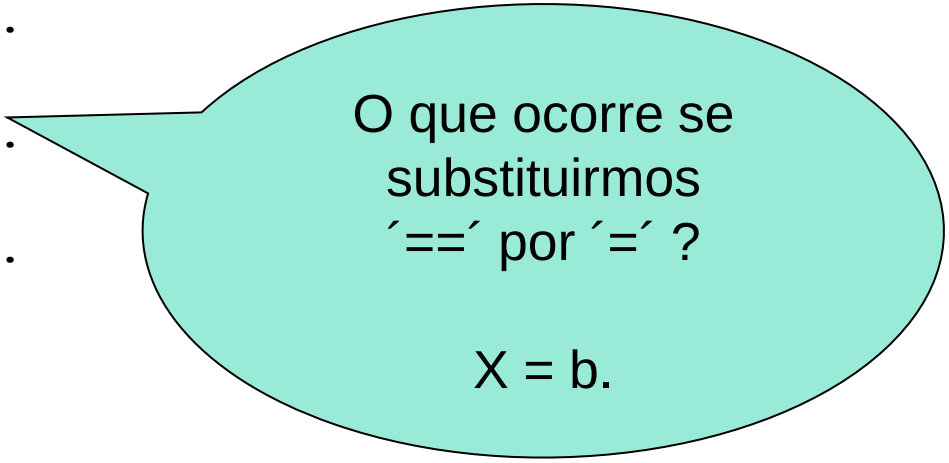
- As regras que regem se dois termos **S** e **T** unificam são:
  1. se **S** e **T** são constantes, então **S** e **T** unificam somente se são o mesmo objeto
  2. se **S** for uma variável e **T** for qualquer termo, então unificam e **S** é instanciado para **T**
  3. se **S** e **T** são estruturas,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

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 \= Y	X não unifica com Y, sendo 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 \== Y	X não é literalmente igual a Y, sendo o complemento de X==Y
X @< Y	X precede Y, *i.e.*, X é ordenado antes de Y
X @> Y	Y precede X
X @=< Y	X precede ou é igual a Y
X @>= Y	Y precede ou é igual a X

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
```



Operador ==

- Teste se dois termos são idênticos

```
?- a == a.
yes

?- a == b.
no

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

- Comparando == e =

```
?- X==Y.
no

?- X=Y.
X = _2808
Y = _2808
yes
```

Operador ==

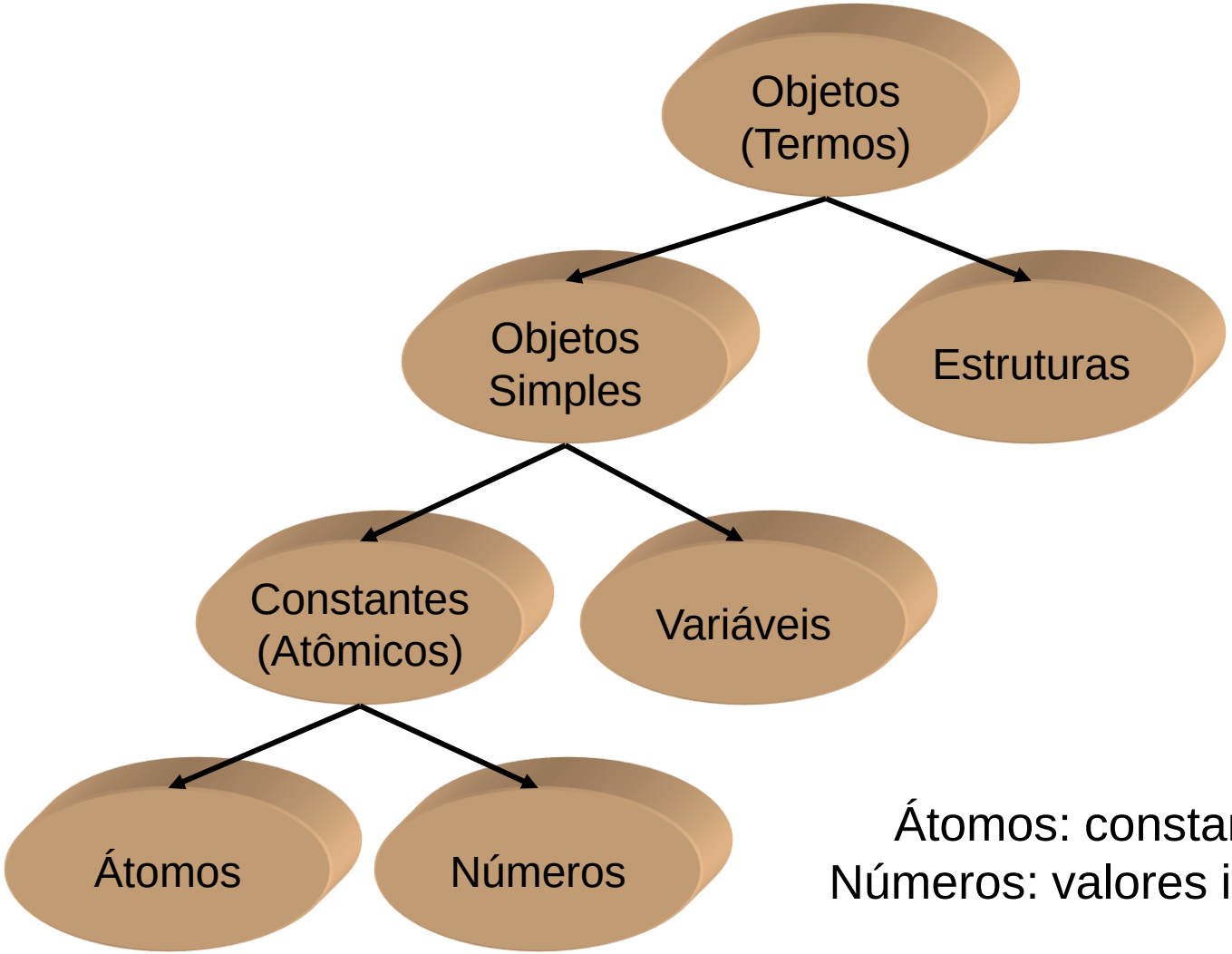
- Comparando == e =

```
?- a=X, a==X.
X = a
yes
```

- Comparando == e =

```
?- X=Y, X==Y.
X = _4500
Y = _4500
yes
```

Objetos em Prolog



Átomos: constantes textuais
Números: valores inteiros ou reais

Precedência de Termos

- A precedência entre termos simples é determinada para ordem alfabética ou numérica:

  variáveis 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 functores iguais, então a precedência é definida (da esquerda para a direita) pelos functores dos seus componentes

Precedência de Termos

variáveis livres
@< números
@< átomos
@< estruturas

```
?- X @< 10.
yes
?- X @< isaque.
yes
?- X @< f(X,Y).
yes
?- 10 @< sara.
yes
?- 10 @< f(X,Y).
yes
?- isaque @< sara.
yes
```

```
?- g(X) @< f(X,Y).
yes
?- f(Z,b) @< f(a,A).
yes
?- 12 @< 13.
yes
?- 12.5 @< 20.
yes
?- g(X,f(a,Y)) @<
   g(X,f(b,Y)).
yes
```

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

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, baseando-se em *backward chaining* – a partir de uma lista de metas, processa-se da conclusão para as premissas dessas metas para encontrar dados que as satisfaçam

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 (dialeto Prolog) o operador **\+** é denotado por **not**
  - O predicado **true/0** sempre é verdadeiro
  - O predicado **fail/0** sempre falha

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.

- Como ocorre em outras linguagens, 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.

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).
```

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).`

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).`

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 na pergunta inicial pelo corpo instanciado da cláusula 7, obtendo a nova lista de condições:

`preto(X), grande(X).`

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 (i.e., é um fato), assim a lista de condições depois de instanciada torna-se:

`grande(gato).`

uma vez que já se provou `preto(gato)`

`X` instancia com gato

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 instanciação `X=gato`

Novamente a meta é `preto(X), grande(X).`

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 por cláusula que unifique com `preto(X)` 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).`

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

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 instanciação correspondente é
`X = urso`

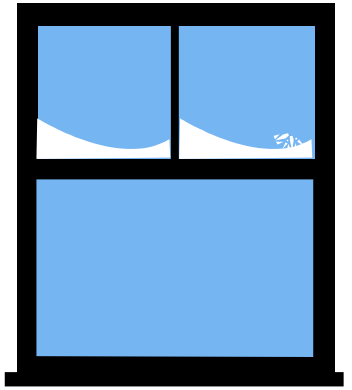
Significado Procedural

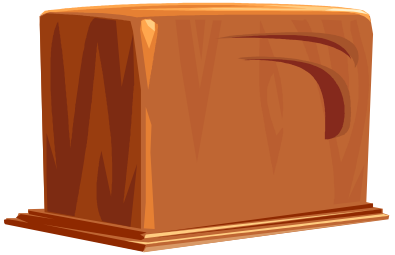
```
grande(urso).          % Cláusula 1
grande(elefante).      % Cláusula 2
pequeno(gato).         % Cláusula 3
marrom(urso).          % Cláusula 4
preto(gato)            % Cláusula 5
cinza(elefante).       % Cláusula 6
escuro(Z) :-           % Cláusula 7
  preto(Z).
escuro(Z) :-           % Cláusula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

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

(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 instanciação correspondente é
`X = urso`

Exemplo: Macaco & Banana





Exemplo: Macaco & Banana

- Um macaco encontra-se próximo à porta de uma sala
- No meio da sala há uma banana pendurada no teto
- O macaco tem fome e quer comer a banana mas ela está a uma altura fora de seu alcance
- Perto da janela da sala encontra-se uma caixa que o macaco pode utilizar para alcançar a banana

Exemplo: Macaco & Banana

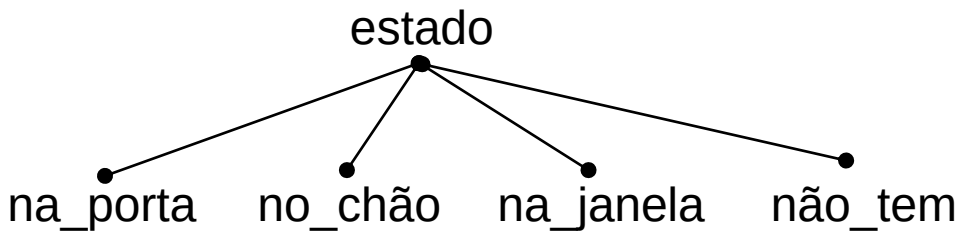
- O macaco pode realizar as seguintes ações:
  - caminhar no chão da sala;
  - subir na caixa (se estiver ao lado da caixa);
  - empurrar a caixa pelo chão da sala (se estiver ao lado da caixa);
  - pegar a banana (se estiver parado sobre a caixa diretamente embaixo da banana).
- É conveniente combinar essas 4 partes de informação em uma estrutura, cujo functor será **estado**

Exemplo: Macaco & Banana

- Possíveis valores para os argumentos da estrutura **estado**
  - 1º argumento (posição horizontal do macaco): na_porta, no_centro, na_janela
  - 2º argumento (posição vertical do macaco): no_chão, acima_caixa
  - 3º argumento (posição da caixa): na_porta, no_centro, na_janela
  - 4º argumento (macaco tem ou não tem banana): tem, não_tem

Exemplo: Macaco & Banana

- O estado inicial é determinado pela posição dos objetos:



- O estado final é qualquer estado no qual o último componente da estrutura é o átomo **tem**
  - `estado(_,_,_,tem)`

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

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

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 construído a partir de 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)

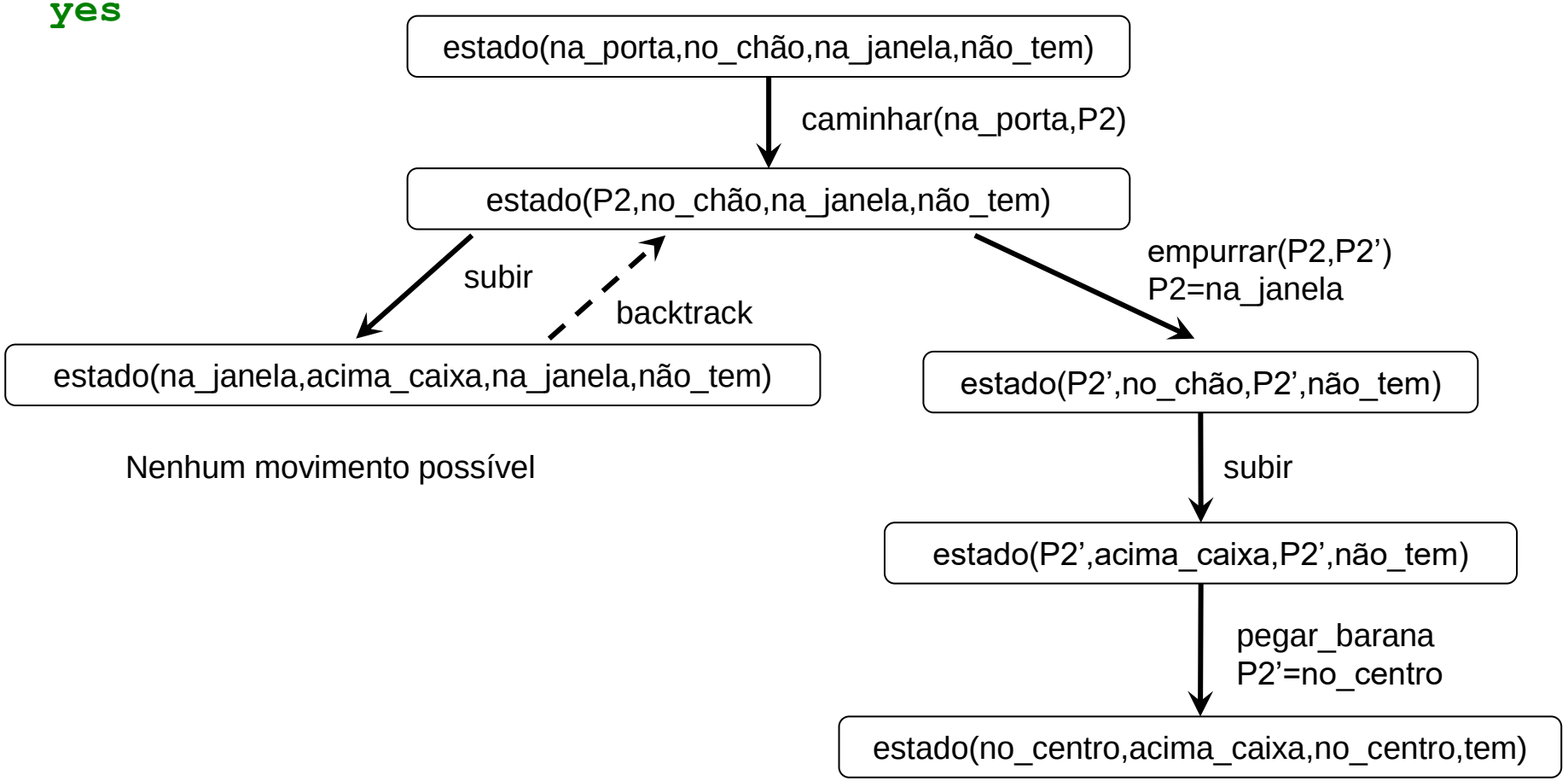
```
consegue(Estado1) :-
    move(Estado1,Movimento,Estado2),
    consegue(Estado2).
```

Exemplo: Macaco & Banana

```
move(estado(no_centro,acima_caixa,no_centro,não_tem),    % estado antes de mover
     pegar_banana,                                       % pega banana
     estado(no_centro,acima_caixa,no_centro,tem) ).      % estado 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).
```

Exemplo: Macaco & Banana

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



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, entre outros
- De acordo com o significado procedural de Prolog, a ordem das cláusulas indica que o macaco prefere pegar a subir, subir a empurrar, assim por diante
- 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

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).
```

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).
```

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

Adaptado por Huei Diana Lee e Newton Spolaôr

http://www.cse.unsw.edu.au/~billw/prologdict.html#firstF
http://www.swi-prolog.org/FAQ/floats.html