

# Programação III

## Paradigmas de Programação Avançados

Salvador Abreu, Universidade de Évora, 2022/23

# Programa em Lógica

Programas revisitados:

- um procedimento é uma coleção de cláusulas.
- uma cláusula é uma conjunção de literais.

Cláusula:

- Cabeça
  - Literal
  - Funtor principal: indica o **predicado**.
- Corpo
  - Um goal
  - Conjunção de literais (o “and” é a vírgula: “,”)
  - Pode levar parêntesis
  - Pode usar ~~disjunção (“;”)~~ e negação por falha (“\+”)

# Unificação

A variável lógica é uma das formas de termos. Recordemos que um termo pode ser:

- uma constante (átomo)
- um termo composto (functor principal + N sub-termos)
- uma variável (livre), dita variável lógica

A **unificação** de dois termos indica se se consegue que sejam idênticos, aplicando uma substituição às variáveis em ambos os termos.

Dizemos que os termos **A** e **B** são unificáveis se existir uma substituição  $\sigma$  em que **A** $\sigma$  fique idêntico a **B** $\sigma$ .

Uma substituição  $\sigma$  é sempre da forma  $\{ \mathbf{V1/S1}, \mathbf{V2/S2}, \dots \}$  em que **Vi** é uma variável e **Si** é o termo pelo qual ela será substituída.

# Denotação dum programa em Lógica

O conjunto dos literais que são verdade.

Normalmente infinito!

Não é explicitamente representado.

Entendimento acerca do que é: designa a **semântica** (dita *declarativa*) dum programa, ou seja, as coisas que é possível deduzir a partir do programa.

# Semântica Operacional

Não dá para calcular o conjunto inteiro de consequências dum programa, usando uma estratégia ascendente (***bottom up***), em geral (*embora nas linguagens ASP seja outra história...*)

Dá sim para determinar se um objetivo (***goal***) é consequência do programa, usando uma estratégia descendente (***top down***).

# Semântica Operacional

Assim, para demonstrar um goal que é uma conjunção de literais  $A_1, A_2, \dots, A_n$ , (dita **resolvente**) o interpretador Prolog começa por seleccionar um literal (o primeiro), digamos  $A$ . Depois, escolhe uma cláusula que unifique (por uma substituição  $\sigma$ ) com  $A$ , i.e. uma da forma:

$$A' :- B_1, B_2, \dots, B_k.$$

Em que a unificação com  $\sigma$  resulta na identidade  $A\sigma = A'\sigma$ .  $\sigma$  será o Unificador Mais Geral de  $A$  e  $A'$ .

O interpretador irá substituir  $A\sigma$  por  $(B_1\sigma, B_2\sigma, \dots, B_k\sigma)$ , recomeçando o processo **até a resolvente estar vazia**.

Pode haver mais duma cláusula  $A'$  aplicável: o interpretador irá registar *todas as alternativas* e *explorá-las sequencialmente*, sempre que a alternativa anterior tiver falhado. Chama-se a isto *retrocesso* ou **backtracking**.

# Semântica operacional da falha

A falha (**failure**) ocorre quando um goal da resolvente não pode ser demonstrado.

Acontece quando não há cláusula que unifique.

A estratégia do interpretador é desfazer todas as substituições até à mais recente **que tenha alternativas**, e retomar a execução escolhendo a próxima cláusula correspondente.

# Procedimento de prova (um **interpretador Prolog**)

proc **demonstrar** (**G**: lista de literais, **G0**) =

se **G** = [] então

// o goal inicial **G0** foi demonstrado

senão, seja **G** = [**A1**, **A2**, ..., **An**]

para todas as cláusulas (**A'** :- **B1**, **B2**, ..., **Bk**)

**σ** ← mgu (**A1**, **A'**)

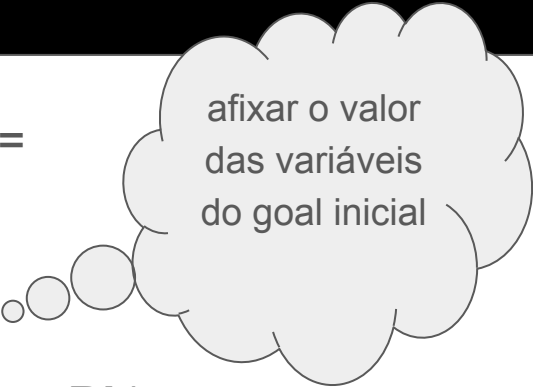
se **σ** ≠ falha então

**demonstrar** ([**B1σ**, **B2σ**, ..., **Bkσ**, **A2σ**, ..., **Anσ**], **G0σ**)

es

arap

es



afixar o valor  
das variáveis  
do goal inicial



# Execução dum programa Prolog

É feita pela invocação do procedimento de prova “demonstra”.

Chama-se à árvore de chamadas recursivas de demonstra uma ***árvore de pesquisa***.

A estratégia:

- Não é completa: pode haver sequências infinitas (recursão infinita)
- Depende da ordem das cláusulas e da ordem dos literais dentro duma cláusula.

# O cut

O **cut** (ou corte) denota-se “!”

Não tem significado lógico (nem verdade nem falso).

No procedimento “demonstra” o cut é sempre verdade (sucede), mas tem um *efeito secundário*: **elimina as alternativas à cláusula que continha o cut.**

# Cut: exemplo (sem cut)

Se tivermos

```
p(X,Y) :- q(X), r(X, Y).  
p(b(X),a(Y)) :- s(X), r(X,Y).  
p(z,z).
```

```
q(a). q(b).  
s(c).
```

```
r(a,a0). r(a,a1).  
r(b,b0). r(b,b1). r(b,b2).  
r(c,c0). r(c,c1).
```

```
| ?- p(X,Y).
```

```
X = a  
Y = a0 ? a
```

```
X = a  
Y = a1
```

```
X = b  
Y = b0
```

```
X = b  
Y = b1
```

```
X = b  
Y = b2
```

```
X = b(c)  
Y = a(c0)
```

```
X = b(c)  
Y = a(c1)
```

```
X = z  
Y = z
```

```
yes  
| ?-
```

# Cut: exemplo (cut 1)

Se tivermos

```
p(X,Y) :- q(X), r(X, Y), !.  
p(b(X),a(Y)) :- s(X), r(X,Y), !.  
p(z,z).
```

```
q(a). q(b).  
s(c).
```

```
r(a,a0). r(a,a1).  
r(b,b0). r(b,b1). r(b,b2).  
r(c,c0). r(c,c1).
```

```
| ?- p(X,Y).
```

```
X = a  
Y = a0
```

```
yes  
| ?- p(b,Y).
```

```
Y = b0
```

```
yes  
| ?- p(b(X),Y).
```

```
X = c  
Y = a(c0)
```

```
yes  
| ?-
```

# Cut: exemplo (cut 2)

Se tivermos

```
p(X,Y) :- q(X), !, r(X, Y).  
p(b(X),a(Y)) :- s(X), !, r(X,Y).  
p(z,z).
```

```
q(a). q(b).  
s(c).
```

```
r(a,a0). r(a,a1).  
r(b,b0). r(b,b1). r(b,b2).  
r(c,c0). r(c,c1).
```

```
| ?- p(X,Y).
```

```
X = a  
Y = a0 ? ;
```

```
X = a  
Y = a1
```

```
yes  
| ?- p(b(X),Y).
```

```
X = c  
Y = a(c0) ? ;
```

```
X = c  
Y = a(c1)
```

```
yes  
| ?-
```

# Alguns predicados built-in (de sistema)

**call/1: call(*G*)** corresponde a interpretar o termo *G* como um novo goal, e tentar interpretá-lo, i.e. **call(*G*)** é verdade se *G* também for.

**call/2, call/3, ... call/*N*: call(*G*, *A*), call(*G*, *A*, *B*), ...** como anteriormente, mas acrescenta os argumentos (*A*, *B*, ...) ao goal *G*. Por exemplo: **call(liga(*X*), *A*, *B*)** é como se fosse chamado **liga(*X*, *A*, *B*)**.

**=/2: igualdade (unicidade): *X*=*Y*** sucede se *X* e *Y* **forem unificáveis**. É como se estivesse definido assim:

***X*=*X*.**

Nota: **=** é um operador binário, portanto ***A*=*B*** é simplesmente notação infixa para **'='(*A*, *B*)**.

**\+/1: negação por falha**, i.e. **\+ *G*** sucede se *G* falhar e vice-versa.

# Cut: usos

Serve para nos “comprometermos” com o que acabou de ser provado, i.e. o goal que vem imediatamente antes (ou a escolha da cláusula caso esteja no início).

Padrão de cláusula:

**CABEÇA :- CONDIÇÃO, !, CORPO.**

“CONDIÇÃO” é frequentemente chamada “guarda”. Ex:

```
comenta(N) :- N<8, !, write(chumbaste).  
comenta(N) :- N>16, !, write(parabens).  
comenta(N) :- N>10, !, write(passaste).  
comenta(N) :- !, write(vais_a_oral).
```

Nota: a última cláusula é como se tivesse sido escrita:

```
comenta(N) :- true, !, write(vais_a_oral).
```

# Cut nos built-ins

O cut é essencial para definir muitos built-ins, especialmente os extra-lógicos.

```
\+ G :- call(G), !, fail.
```

```
\+ _.
```

```
var(V) :- \+ \+ X=a, \+ \+ X=b.
```

```
nonvar(T) :- var(T), !, fail.
```

```
nonvar(_).
```

```
memberchk(X, [X|_]) :- !.
```

```
memberchk(X, [_|L]) :- memberchk(X, L).
```



# Built-ins

Ver o manual de referência do GNU Prolog

- Manual completo: [http://gprolog.org/manual/html\\_node/](http://gprolog.org/manual/html_node/)
- Só built-ins: [http://gprolog.org/manual/html\\_node/gprolog024.html](http://gprolog.org/manual/html_node/gprolog024.html)

No caso do SWI Prolog temos uma situação semelhante.