

Programação em Lógica

Prof. A. G. Silva

01 de setembro de 2016

- Referem-se a todas as construções sintáticas da linguagem
- Um termo pode ser:
 - ▶ Constante
 - ▶ Variável
 - ▶ Estrutura
- **Constantes** podem ser átomos ou números (em LISP, números são átomos)

Átomos e números

- Um átomo indica um objeto ou uma relação
 - ▶ Nomes de objetos como `maria`, `livro`, etc.
 - ▶ Nomes de predicados são sempre atômicos
 - ▶ Os grupos de caracteres `?-` (usado em perguntas) e `:-` (usado em regras)
 - ▶ Átomos de comprimento igual a um são os caracteres (podem ser lidos e impressos)
- Em relação a números, Prolog acompanha a outras linguagens, permitindo inteiros positivos e negativos, números em ponto flutuante. Exemplos válidos:
0, 1, -17, 2.35, -0.27653, 10e10, 6.02e-23

Variáveis

- Nomes cujo primeiro caracter é uma letra maiúscula; ou o sinal de sublinhado (“_”) para variáveis anônimas
- Variáveis com mesmo nome em uma mesma cláusula são as mesmas e ganham um único valor
- Variáveis anônimas são diferentes das outras nos seguintes aspectos:
 - 1 Cada ocorrência delas indica uma variável diferente, ainda que na mesma cláusula
 - 2 Ao serem usadas em uma pergunta, seus valores não são impressos nas respostas
- Variáveis anônimas são usadas para unificar com qualquer termo não interessando com qual valor serão instanciadas

Estruturas

- São termos mais complexos formados por um functor, seguido de componentes, separadas por vírgula entre parênteses. Exemplo:
`livro(incidente_em_antares, verissimo).`

Fatos de um banco de dados em Prolog são estruturas seguidas por um ponto final.

- Estruturas podem ser aninhadas. Exemplo:
`livro(incidente_em_antares, autor(eric, verissimo)).`
- Podem ser argumentos de fatos no banco de dados:
`pertence(pedro, livro(incidente_em_antares, verissimo)).`
- O número de componentes de um functor é a sua aridade.

Operadores

- Certas estruturas na forma infixa (em vez de pré-fixa): functor escrito como operador
- Propriedades a especificar: posição (pré, in ou pós-fixa), precedência (um número, quanto menor, maior a prioridade nos cálculos) e associatividade (esquerda ou direita)
- Operadores aritméticos $+$, $-$, $*$, $/$ geralmente infixos. Unário de negação pré-fixado.
- Precedência de $*$ e $/$ é maior que de $+$ e $-$
- A associatividade dos aritméticos é esquerda:
 $8/2/2$ é $(8/2)/2$ (e não $8/(2/2)$)

Importante

- Em Prolog, $2 + 3$ é simplesmente um fato
- Para cálculos, é necessário usar o predicado `is`
- Em Prolog, igualdade significa unificação. Existe o predicado `=` infixado, mas, em geral, pode ser substituído pelo uso de variáveis de mesmo nome. Se não existisse, poderia ser definido pelo fato:

$X = X.$

Como se pode definir o predicado abaixo sem usar igualdade?

`pai (ful ano, ci cl ano).`

`pai (bel trano, ci cl ano).`

`pai (pedro, joaqui m).`

`irmaos(X, Y) :- pai(X, PX), pai(Y, PY), PX = PY.`

Aritmética

- Predicados de comparação são infixos e comparam números ou variáveis instanciadas a números:

$=$ igual

\neq diferente

$<$ menor

$>$ maior

\leq menor ou igual

\geq maior ou igual

$=$ unifica (ex.: $X=Y$ significa que X unifica com Y)

- Predicados pré-definidos não pode ser redefinidos, nem podem ter fatos ou regras adicionados a eles

Exemplo de comparadores

- Banco de dados dos príncipes de Gales nos séculos 9 e 10, e os anos que reinaram:

rei nou(rhodi , 844, 878).

rei nou(anarawd, 878, 916).

rei nou(hywel _dda, 916, 950).

rei nou(l ago_ap_i euaf, 979, 965).

rei nou(hywal _ap_i euaf, 979, 965).

rei nou(cadwal l on, 985, 986).

rei nou(maredudd, 986, 999).

Quem foi o príncipe em um dado ano:

princi pe(X, Y) :- rei nou(X, A, B), Y >= A, Y =< B.

Operador is

- Infixo. Em seu lado direito deve vir uma expressão aritmética com apenas números ou variáveis instanciadas com números. Do lado esquerdo pode ser uma variável não instanciada (que passa a ser instanciada com o valor)
- Pode ser usado também como operador de igualdade numérica (lado esquerdo e direito iguais)
- Único que tem poder de calcular resultados de operações aritméticas

Operador is

- Exemplo (população em milhões de pessoas, área em milhões de quilômetros quadrados):

`pop(eua, 203).`

`pop(índia, 548).`

`pop(china, 800).`

`pop(brasil, 108).`

`area(eua, 8).`

`area(índia, 3).`

`area(china, 10).`

`area(brasil, 8).`

Densidade populacional:

`dens(X, Y) :- pop(X, P), area(X, A), Y is P/A.`

Operadores aritméticos

- Mais utilizados:

+	soma
-	subtração
*	multiplicação
/	divisão
//	divisão inteira
mod	resto da divisão
**	potenciação

- Outros operadores: <http://www.swi-prolog.org/man/arith.html>

Exercícios

- 1 Considere o seguinte banco de dados:

```
soma(5).  
soma(2).  
soma(2 + X).  
soma(X + Y).
```

e a meta

```
soma(2 + 3)
```

Com quais fatos esta meta unifica? Quais as instanciações de variáveis em cada caso?

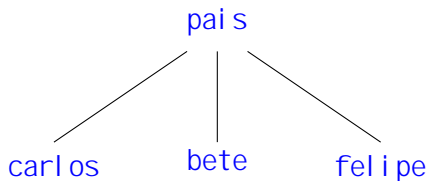
- 2 Quais são os resultados das perguntas abaixo?

```
?- X is 2 + 3.  
?- X is Y + Z.  
?- 6 is 2 * 4.  
?- X = 5, Y is X // 2.  
?- Y is X // 2, X = 5.
```

Estruturas de dados

- Prolog tem suporte versátil para representar estruturas de dados
- As estruturas podem ser desenhadas como árvores, onde o funtor é a raiz e os componentes, seus filhos:

`pais(carlos, bete, felipe)`



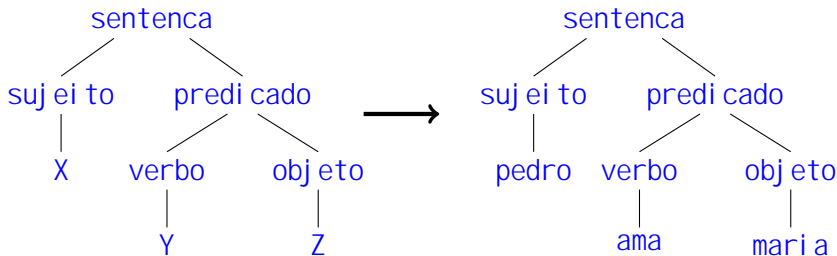
- Frases da língua portuguesa podem ter suas sintaxes representadas por estruturas em Prolog.

Estruturas de dados

- Exemplo de sentença simples com sujeito e predicado (\neq do Prolog):

`sentenca(sujeito(X), predicado(verbo(Y), objeto(Z)))`

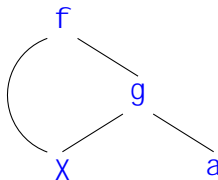
Exemplo: sentença “Pedro ama Maria”, instanciando as variáveis da estrutura com palavras da sentença



Estruturas de dados

- Forma pictórica para mostrar ocorrências de uma mesma variável:

$f(X, g(X, a))$



- Neste caso, tem-se um grafo orientado acíclico

Listas

- Em Prolog, uma lista é ou:
vazia
 - ▶ uma lista vazia
[]
 - ▶ ou uma estrutura com dois componentes: a

Listas

- Prolog (como LISP) tem maneira alternativa de denotar listas
- Basta colocar os elementos separados por vírgulas entre colchetes:
[a, b, c]
- Qualquer termo pode ser componente de uma lista, como variáveis ou outras listas:

[o, homem, [gosta, de, pescar]]

[a, V1, b, [X, Y]]

Listas

- Listas são processadas, dividindo-as em cabeça e cauda, exceto quando vazia (como o car e o cdr em LISP). Alguns exemplos:

Lista	Cabeça	Cauda
[a, b, c]	a	[b, c]
[a]	a	[]
[]	<i>nao tem</i>	<i>nao tem</i>
[[o, gato], sentou]	[o, gato]	[sentou]
[o, [gato, sentou]]	o	[[gato, sentou]]
[o, [gato, sentou], ali]	o	[[gato, sentou], ali]
[X + Y, x + y]	X + Y	[x + y]

Listas

- Notação mais intuitiva para indicar $.(X, Y)$ como decomposição de uma lista em cabeça e cauda:

$[X|Y]$

- Exemplos

$p([1, 2, 3]).$

$p([o, gato, sentou, [no, capacho]]).$

$?- p([X|Y]).$

$X = 1, Y = [2, 3];$

$X = o, Y = [gato, sentou, [no, capacho]];$

no

$?- p([_, _, _, [_|X]]).$

$X = [capacho]$

Exercício

- Decida se as unificações abaixo ocorrem, e quais são as instanciações de variáveis em cada caso positivo.

$[X, Y, Z] = [\text{pedro}, \text{adora}, \text{peixe}]$.

$[\text{gato}] = [X|Y]$.

$[X, Y|Z] = [\text{maria}, \text{aprecia}, \text{vinho}]$.

$[[a, X]|Z] = [[X, \text{lebre}], [\text{veio}, \text{aqui}]]$.

$[[\text{lebre}, X]|Z] = [[X, \text{lebre}], [\text{veio}, \text{aqui}]]$.

$[\text{anos}|T] = [\text{anos}, \text{dourados}]$.

$[\text{vale}, \text{tudo}] = [\text{tudo}, X]$.

$[\text{cavalo}|Q] = [P|\text{branco}]$.

$[] = [X|Y]$.

Recursão

- Suponha a verificação de um determinada cor em uma lista:

[azul , verde, vermel ho, amarel ho]

Procedimento em Prolog:

- ▶ Verificar se está na cabeça. Se não, procurar na cauda
- ▶ Verificar então a cabeça da cauda... e assim por diante
- ▶ A falha (cor ausente) ocorre ao tomar uma lista vazia

Recursão

- Para implementar em Prolog, é preciso definir uma relação entre objetos e listas onde eles aparecem
- `member(X, Y)` é verdadeiro se o termo `X` é um elemento da lista `Y`
- Duas condições a verificar:
 - ▶ É um fato que `X` é um elemento da lista `Y`, se `X` for igual à cabeça de `Y`
`member(X, Y) :- Y = [X|_].`
ou
`member(X, [X|_]).`
 - ▶ `X` é membro de `Y`, se `X` é membro da cauda de `Y` (recursão):
`member(X, Y) :- Y = [_|Z], member(X, Z).`
ou
`member(X, [_|Y]) :- member(X, Y).`

Recursão

- O uso da variável anônima significa o não interesse sobre a cauda (no primeiro caso) ou sobre a cabeça (no segundo caso). Exemplos:

```
?- member(d, [a, b, c, d, e, f, g]).
```

```
true
```

```
?- member(2, [3, a, d, 4]).
```

```
false
```

- Para definir predicado recursivo é preciso das condições de parada e do caso recursivo
- No caso de `member`, há duas condições de parada:
 - ▶ Primeira cláusula: se o primeiro argumento unificar com a cabeça do segundo argumento
 - ▶ Segunda cláusula: se o segundo argumento é a lista vazia, que não unifica com nenhuma das cláusulas e faz o predicado falhar

Recursão

- Predicados podem funcionar bem em chamadas com constantes, mas falhar com variáveis. O exemplo:

```
lista([A|B]) :- lista(B).  
lista([ ]).
```

é a própria definição de lista, funcionando bem com constantes:

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

```
true
```

```
?- lista([ ]).
```

```
true
```

```
?- lista(f(1, 2, 3)).
```

```
false
```

mas Prolog entrará em *loop* em (inverter a ordem das cláusulas resolve o problema):

```
?- lista(X).
```

Concatenação de listas

- Predicado `append` para concatenar duas listas.

```
append([ ], L, L).  
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

- Exemplos:

```
?- append([al fa, beta], [gama, del ta], X).  
X = [al fa, beta, gama, del ta]
```

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

Concatenação de listas

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

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

- A primeira cláusula é a condição de parada. Lista vazia concatenada com qualquer lista resulta na própria lista. Para a segunda condição:
 - 1 O primeiro elemento da primeira lista será também o primeiro elemento da terceira lista
 - 2 Concatenando a cauda da primeira lista com a segunda lista resulta na cauda da terceira lista
 - 3 Deve-se usar o próprio `append` para obter a concatenação de ?? acima
 - 4 Ao ir aplicando a segunda cláusula, o primeiro argumento vai reduzindo, até ser a lista vazia, e finalizar a recursão

Acumuladores

- Cálculos podem depender do que já foi encontrado até o momento
- A técnica de acumuladores consiste em utilizar um ou mais argumentos do predicado para representar “a resposta até o momento” durante este percurso
- Estes argumentos recebem o nome de acumuladores
- Prolog possui um predicado pré-definido `length` para o cálculo do comprimento de uma lista. Segue uma definição própria, **sem acumuladores**:

```
length([ ], 0).
```

```
length([H|T], N) :- length(T, N1), N is N1 + 1.
```

Acumuladores

- A definição **com acumulador** baseia-se no mesmo princípio recursivo, mas acumula a resposta a cada passo num argumento extra:

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

```
lenacc([H|T], A, N) :- A1 is A + 1, lenacc(T, A1, N).
```

```
listlen(L, N) :- lenacc(L, 0, N).
```

- Sequência de submetas para o comprimento de `[a, b, c, d, e]`:

```
listlen([a, b, c, d, e], N)
```

```
lenacc([a, b, c, d, e], 0, N)
```

```
lenacc([b, c, d, e], 1, N)
```

```
lenacc([c, d, e], 2, N)
```

```
lenacc([d, e], 3, N)
```

```
lenacc([e], 4, N)
```

```
lenacc([ ], 5, N)
```

Acumuladores

- Acumuladores não precisam ser números inteiros. Segue definição do predicado `rev` (Prolog tem sua própria versão chamada `reverse`) para inversão da ordem dos elementos de uma lista:

```
rev(L1, L3) :- revacc(L1, [ ], L3).
```

```
revacc([ ], L3, L3).
```

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

- O segundo argumento `revacc` serve como acumulador. A sequência de metas para `?- rev([a, b, c, d], L3).`:

```
rev([a, b, c, d], L3)
```

```
revacc([a, b, c, d], [ ], L3)
```

```
revacc([b, c, d], [a], L3)
```

```
revacc([c, d], [b, a], L3)
```

```
revacc([d], [c, b, a], L3)
```

```
revacc([ ], [d, c, b, a], L3)
```

Exercícios

- 1 Escreva um predicado `last(L, X)` que é satisfeito quando o termo `X` é o último elemento da lista `L`.
- 2 Escreva um predicado `efface(L1, X, L2)` que é satisfeito quando `L2` é a lista obtida pela remoção da primeira ocorrência de `X` em `L1`.
- 3 Escreva um predicado `delete(L1, X, L2)` que é satisfeito quando `L2` é a lista obtida pela remoção de todas as ocorrências de `X` em `L1`.
- 4 Construa regra(s) para calcular o preço total de uma lista de compras
`preco(maca, 2).`
`preco(manga, 3).`
`?- precoTotal ([maca, manga], P).`
`P = 5.`