Sistemas Baseados em Conhecimento
Aula 12

Renata Wassermann

renata@ime.usp.br

2017

O que é Prolog?

- Prolog (*pro*gramming in *log*ic) é uma linguagem de programação baseada em lógica: programas correspondem a conjuntos de fórmulas lógicas e o interpretador Prolog usa métodos lógicos para resolver consultas.

O que é Prolog?

- Prolog (*pro*gramming in *log*ic) é uma linguagem de programação baseada em lógica: programas correspondem a conjuntos de fórmulas lógicas e o interpretador Prolog usa métodos lógicos para resolver consultas.
- Prolog é uma linguagem declarativa: você especifica o problema a ser resolvido e não como resolvê-lo.

Fatos

Um pequeno programa consistindo de quatro fatos:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

Consultas

Depois de carregar o programa, podemos fazer consultas:

```
?- bigger(donkey, dog).
Yes
```

```
?- bigger(monkey, elephant).
No
```

Um Problema

```
?- bigger(elephant, monkey).
No
```

O predicado `bigger/2` não é exatamente o que procuramos.

Um Problema

```
?- bigger(elephant, monkey).
No
```

O predicado `bigger/2` não é exatamente o que procuramos.
O que queremos, na verdade, é o *fecho transitivo* be `bigger/2`.

Regras

As seguintes regras definem `is_bigger/2` como o fecho transitivo de `bigger/2` usando recursão:

```
is_bigger(X, Y) :- bigger(X, Y).
```

```
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Regras

As seguintes regras definem `is_bigger/2` como o fecho transitivo de `bigger/2` usando recursão:

```
is_bigger(X, Y) :- bigger(X, Y).
```

```
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

:- $\Longrightarrow$ "if"
, $\Longrightarrow$ "and"

Agora funciona

```
?- is_bigger(elephant, monkey).
Yes
```

Agora funciona

```
?- is_bigger(elephant, monkey).
Yes
```

Melhor ainda, podemos usar a *variável* `X`:

Agora funciona

```
?- is_bigger(elephant, monkey).
Yes
```

Melhor ainda, podemos usar a *variável* `X`:

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

Tecle ponto e vírgula (;) para obter soluções alternativas.

Outro exemplo

Existem animais que são menores que um burro e maiores que um macaco?

```
?- is_bigger(donkey, X), is_bigger(X, monkey).
No
```

Termos

Termos em Prolog podem ser *números*, *átomos*, *variáveis* ou *termos compostos*.

- Átomos começam com letra minúscula ou aparecem entre aspas simples:

```
elephant, xYZ, a_123, 'Mais uma cerveja, por favor'
```

- Variáveis começam com letra maiúscula ou sublinhado:

```
X, Elephant, _G177, MyVariable, _
```

Termos

- Termos compostos têm um functor (um átomo) e um número de argumentos (termos):

```
is_bigger(horse, X)
f(g(Alpha, _), 7)
'My Functor'(dog)
```

Termos

- Termos compostos têm um functor (um átomo) e um número de argumentos (termos):

```
is_bigger(horse, X)
f(g(Alpha, _), 7)
'My Functor'(dog)
```

- Atomos e números são chamados *termos atômicos*.

Termos

- Termos compostos têm um functor (um átomo) e um número de argumentos (termos):

```
is_bigger(horse, X)
f(g(Alpha, _), 7)
'My Functor'(dog)
```

- Atomos e números são chamados *termos atômicos*.
- Termos sem variáveis são chamados *termos básicos* (*ground terms*).

Fatos e Regras

Fatos são predicados seguidos por um ponto. São usados para definir algo que deve ser incondicionalmente verdadeiro.

```
bigger(elephant, horse).
parent(john, mary).
```

Fatos e Regras

Fatos são predicados seguidos por um ponto. São usados para definir algo que deve ser incondicionalmente verdadeiro.

```
bigger(elephant, horse).
parent(john, mary).
```

Regras consistem de uma *cabeça* e um *corpo* separados por `:-`. A cabeça de uma regra é verdadeira se todos os predicados do corpo forem verdadeiros.

```
grandfather(X, Y) :-
    father(X, Z),
    parent(Z, Y).
```

Programas e Consultas

- Fatos e regras são chamados de *cláusulas*. Um programa Prolog é uma lista de cláusulas.

Programas e Consultas

- Fatos e regras são chamados de *cláusulas*. Um programa Prolog é uma lista de cláusulas.
- *Consultas* são predicados (ou sequências de predicados) seguidos de um ponto. Eles são digitados no prompt do Prolog e fazem o sistema dar uma resposta.

```
?- is_bigger(horse, X), is_bigger(X, dog).
X = donkey
Yes
```

Predicados da Linguagem (Built-in)

- Carregando um arquivo de programa:

```
?- consult('big-animals.pl').
 Yes
```

- Escrevendo na tela

```
?- write('Hello World!'), nl.
Hello World!
Yes
```

Matching

- Dois termos *casam* (*match*) se eles são idênticos ou podem se tornar idênticos substituindo suas variáveis com termos básicos adequados.

Matching

- Dois termos *casam* (*match*) se eles são idênticos ou podem se tornar idênticos substituindo suas variáveis com termos básicos adequados.
- Podemos explicitamente perguntar para o Prolog se dois termos dados casam usando o predicado de igualdade (usado com notação infixa):

```
?- born(mary, yorkshire) = born(mary, X).
X = yorkshire
Yes
```

Exemplos

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes
```

```
?- p(X, 2, 2) = p(1, Y, X).
No
```

A Variável Anônima

- A variável _ é chamada de *variável anônima*.
- Toda ocorrência de _ representa uma variável diferente (e as instanciações não são apresentadas).

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

Respondendo Consultas

- Se um objetivo casa com um fato, então ele é satisfeito.

Respondendo Consultas

- Se um objetivo casa com um fato, então ele é satisfeito.
- Se um objetivo casa com a cabeça de uma regra, ele é satisfeito se o objetivo representado pelo corpo da regra é satisfeito.

Respondendo Consultas

- Se um objetivo casa com um fato, então ele é satisfeito.
- Se um objetivo casa com a cabeça de uma regra, ele é satisfeito se o objetivo representado pelo corpo da regra é satisfeito.
- Se um objetivo consiste em uma lista de subobjetivos separados por vírgulas, então ele é satisfeito se todos os subobjetivos forem satisfeitos.

Respondendo Consultas

- Se um objetivo casa com um fato, então ele é satisfeito.
- Se um objetivo casa com a cabeça de uma regra, ele é satisfeito se o objetivo representado pelo corpo da regra é satisfeito.
- Se um objetivo consiste em uma lista de subobjetivos separados por vírgulas, então ele é satisfeito se todos os subobjetivos forem satisfeitos.
- Quando tenta satisfazer objetivos com predicados da linguagem, como `write/1`, o Prolog também executa a ação associada.

Exemplo: Filósofos Mortais

Considere o seguinte argumento:

Todo homem é mortal
Sócrates é um homem

Sócrates é mortal

Ele possui duas premissas e uma conclusão.

Traduzindo para Prolog

As duas premissas podem ser expressas como um pequeno programa Prolog:

```
mortal(X) :- man(X).
man(socrates).
```

A conclusão pode ser formulada como uma consulta:

```
?- mortal(socrates).
Yes
```

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.
2. Prolog procura o primeiro fato ou cabeça de regra que casam e encontra `mortal(X)`. Instanciação: `X=socrates`.

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.
2. Prolog procura o primeiro fato ou cabeça de regra que casam e encontra `mortal(X)`. Instanciação: `X=socrates`.
3. A instanciação é estendida pata o corpo da regra, i.e., `man(X)` torna-se `man(socrates)`.

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.
2. Prolog procura o primeiro fato ou cabeça de regra que casam e encontra `mortal(X)`. Instanciação: `X=socrates`.
3. A instanciação é estendida pata o corpo da regra, i.e., `man(X)` torna-se `man(socrates)`.
4. Novo objetivo: `man(socrates)`.

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.
2. Prolog procura o primeiro fato ou cabeça de regra que casam e encontra `mortal(X)`. Instanciação: `X=socrates`.
3. A instanciação é estendida pata o corpo da regra, i.e., `man(X)` torna-se `man(socrates)`.
4. Novo objetivo: `man(socrates)`.
5. Sucesso, porque `man(socrates)` é um fato do programa.

Execução

1. A consulta `mortal(socrates)` é o objetivo inicial.
2. Prolog procura o primeiro fato ou cabeça de regra que casam e encontra `mortal(X)`. Instanciação: `X=socrates`.
3. A instanciação é estendida pata o corpo da regra, i.e., `man(X)` torna-se `man(socrates)`.
4. Novo objetivo: `man(socrates)`.
5. Sucesso, porque `man(socrates)` é um fato do programa.
6. Portanto o objetivo inicial é satisfeito.

Listas em Prolog

Uma das estruturas de dados mais úteis do Prolog é a *lista*.
Exemplo:

```
[elephant, horse, donkey, dog]
```

Os elementos de uma lista podem ser quaisquer termos Prolog (incluindo outras listas).
A lista vazia é `[]`.
Outro exemplo:

```
[a, X, [], f(X,y), 47, [a,b,c], bigger(cow,dog)]
```

Representação Interna

Internamente, a lista

```
[a, b, c]
```

corresponde ao termo

```
.(a, .(b, .(c, [])))
```

Ou seja, listas são apenas uma nova notação para termos compostos com o functor . e o átomo especial [].

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

A Notação com Barra

Se uma barra (|) é colocada antes do último termos da lista, isso significa que ele é uma sublista. Inserindo os elementos antes da barra no início da sublista resulta na lista inteira.
Por exemplo: `[a, b, c, d]` é o mesmo que `[a, b | [c, d]]`.

Exemplos

Extrair o segundo elemento de uma lista:

```
?- [a, b, c, d, e] = [_, X | _].
X = b
Yes
```

Exemplos

Extrair o segundo elemento de uma lista:

```
?- [a, b, c, d, e] = [_, X | _].
X = b
Yes
```

Verificar se o primeiro elemento é 1 e obter a lista após o segundo elemento:

```
?- MyList = [1, 2, 3, 4, 5], MyList = [1, _ | Rest].
MyList = [1, 2, 3, 4, 5]
Rest = [3, 4, 5]
Yes
```

Cabeça e Cauda

O primeiro elemento de uma lista é chamado de *cabeça* e o resto é chamado de *cauda*. A lista vazia não tem cabeça.
Um caso especial da notação com barra, com exatamente um elemento antes da barra, pode ser usado para extrair a cabeça e/ou a cauda de uma lista:

```
?- [elephant, horse, tiger, dog] = [Head | Tail].
Head = elephant
Tail = [horse, tiger, dog]
Yes
```

Cabeça e Cauda

Outro exemplo:

```
?- [elephant] = [X | Y].
X = elephant
Y = []
Yes
```

Concatenando Listas

Queremos escrever um predicado `concat_lists/3` que funcione da seguinte forma:

```
?- concat_lists([1, 2, 3, 4], [dog, cow, tiger], L).
L = [1, 2, 3, 4, dog, cow, tiger]
Yes
```

Concatenando Listas

O predicado `concat_lists/3` é implementado *recursivamente*.

Concatenando Listas

O predicado `concat_lists/3` é implementado *recursivamente*.
O caso base é quando uma das listas é vazia.

Concatenando Listas

O predicado `concat_lists/3` é implementado *recursivamente*.
O caso base é quando uma das listas é vazia.
A cada passo, "corta-se" a cabeça e o mesmo predicado é reutilizado, com uma cauda menor, até alcançar o caso base da recursão.

```
concat_lists([], List, List).

concat_lists([Elem|List1], List2, [Elem|List3]) :-
    concat_lists(List1, List2, List3).
```

Concatenando Listas

`concat_lists/3` também pode ser usado para decompor listas:

```
?- concat_lists(Begin, End, [1, 2, 3]).
Begin = []
End = [1, 2, 3] ;
Begin = [1]
End = [2, 3] ;
Begin = [1, 2]
End = [3] ;
Begin = [1, 2, 3]
End = [] ;
No
```

Expressões Aritméticas em Prolog

O Prolog tem uma série de funções e operadores aritméticos predefinidos.
Expressões como `3 + 5` são termos válidos em Prolog.
Porém,

```
?- 3 + 5 = 8.
No
```

Matching vs. Avaliação Aritmética

Os termos `3 + 5` e `8` não *casam*!
Para somar os números `3` e `5`, precisamos usar o operador `is`:

```
?- X is 3 + 5.
X = 8
Yes
```

O Operador `is`

O operador `is` faz com que o termo à direita seja avaliado como uma expressão aritmética.
O resultado desta avaliação é então casado como o termo à esquerda.
Exemplo:

```
?- Value is 3 * 4 + 5 * 6, OtherValue is Value / 11.
Value = 42
OtherValue = 3.81818
Yes
```

Exemplo: Comprimento de uma Lista

Ao invés de usar `length/2`, podemos escrever nosso próprio predicado:

```
len([], 0).


len([_ | Tail], N) :-
     len(Tail, N1),
     N is N1 + 1.
```

Retrocesso (Backtracking)

- *Pontos de Escolha*: Subobjetivos que podem ser satisfeitos de mais de uma forma.
  Por exemplo `..., member(X, [a, b, c]), ...`
- *Backtracking*: Durante a execução de um objetivo, o Prolog guarda os pontos de escolha. Se um caminho particular termina em falha, ele volta para o ponto de escolha mais recente e tenta a próxima alternativa.

Usando o Backtracking

Dada uma lista na primeira posição, o predicado `permutation/2` gera todas as permutações possíveis da lista no segundo argumento:

```
permutation([], []).

permutation(List, [Element | Permutation]) :-
     select(Element, List, Rest),
     permutation(Rest, Permutation).
```

Exemplo

```
?- permutation([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
No
```

Problemas com o Backtracking

Pedir soluções alternativas gera respostas erradas na definição deste predicado:

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
    member(Head, Tail),
    remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
    remove_duplicates(Tail, Result).
```

Exemplo

```
?- remove_duplicates([a, b, b, c, a], List).
List = [b, c, a] ;
List = [b, b, c, a] ;
List = [a, b, c, a] ;
List = [a, b, b, c, a] ;
No
```

Corte (Cut)

Às vezes queremos impedir o retrocesso do Prolog em determinados pontos de escolha, ou porque as alternativas que restam geram respostas erradas, ou por questão de eficiência.

Isto pode ser feito usando o *cut*, representado por `!`.
Este predicado é sempre satisfeito e impede que o Prolog retroceda para subobjetivos que venham antes do cut na mesma regra.

Exemplo

```
remove_duplicates([], []).

remove_duplicates([Head | Tail], Result) :-
     member(Head, Tail), !,
     remove_duplicates(Tail, Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
     remove_duplicates(Tail, Result).
```

Corte (Cut)

Quando um corte é encontrado, todas as escolhas feitas entre o casamento da cabeça da regra até o corte são finais, isto é, pontos de escolha são ignorados.

Exercício

Usando cortes, implemente um predicado add/3 para inserir um elemento em uma lista caso o elemento não seja um membro da lista. Verifique que não há soluções alternativas erradas:

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit] ;
No
```

```
?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit] ;
No
```

Solução

```
add(Element, List, List) :-
    member(Element, List), !.

add(Element, List, [Element | List]).
```

Problemas com o Corte

O predicado `add/3` não funciona como esperado quando o último argumento já está instanciado!

```
?- add(dog, [dog, cat, bird], [dog, dog, cat, bird]).
Yes
```

Problemas com o Corte

Podemos usar a seguinte implementação alternativa:

```
add(Element, List, Result) :-
    member(Element, List), !,
    Result = List.

add(Element, List, [Element | List]).
```

Veja como o uso do corte afeta o caráter declarativo do Prolog.

Negação e Corte

Podemos usar o corte para definir a negação por falha finita.

```
neg(Goal) :- Goal, !, fail.
neg(Goal).
```

Negação em Prolog

Podemos lidar com exceções:

```
enjoys(vincent,X)  :-  burger(X),
                       neg(big_kahuna_burger(X)).
```

Negação em Prolog

Podemos lidar com exceções:

```
enjoys(vincent,X)  :-  burger(X),
                       neg(big_kahuna_burger(X)).
```

Em Prolog, a negação é pré-definida:

```
enjoys(vincent,X)  :-  burger(X),
                       \+(big_kahuna_burger(X)).
```