Linguagem de Programação
Linguagem Lógica
Parte 1: Prolog

Prof. Arnaldo Candido Junior

UNESP – IBILCE

São José do Rio Preto, SP

Introdução

- Neste tópico, iniciaremos estudando a linguagem Prolog
- Estudaremos primeiro a linguagem e depois formalizaremos os conceitos do paradigma lógico
- Prolog segue a lógica de predicados

Lógicas

- Proposicional (**foco**)
- De Predicados (**foco**)
  - 1ª ordem, 2ª ordem, …
- Várias outras: modal, temporal, difusa (fuzzy), quântica

Declarações

- As instruções em Prolog são realizadas por meio de **statements**
  - Possíveis traduções: declarações (convenção), enunciados ou sentenças
- Exemplo: amigo(maria, jose).
  - Convenção: leitura na ordem 2 1 3 (maria é amiga de josé)

Declarações (2)

- Predicado vem primeiro (“amigo” no exemplo)
- Constantes e predicados iniciam por letras minúsculas
- Variáveis tem sempre letra maiúscula (exemplos a seguir)
- Toda a declaração termina com ponto

Operadores Lógicos

Símbolo	Conectivo	Operação Lógica
:-	IF	Implicação
,	AND	Conjunção
;	OR	Disjunção
not	NOT	Negação

Operadores Relacionais

Operador	Significado
X = Y	Igual a
X \= Y	Não igual a
X < Y	Menor que
Y > X	Maior que
Y =< X	Menor ou igual a
Y >= X	Maior ou igual a

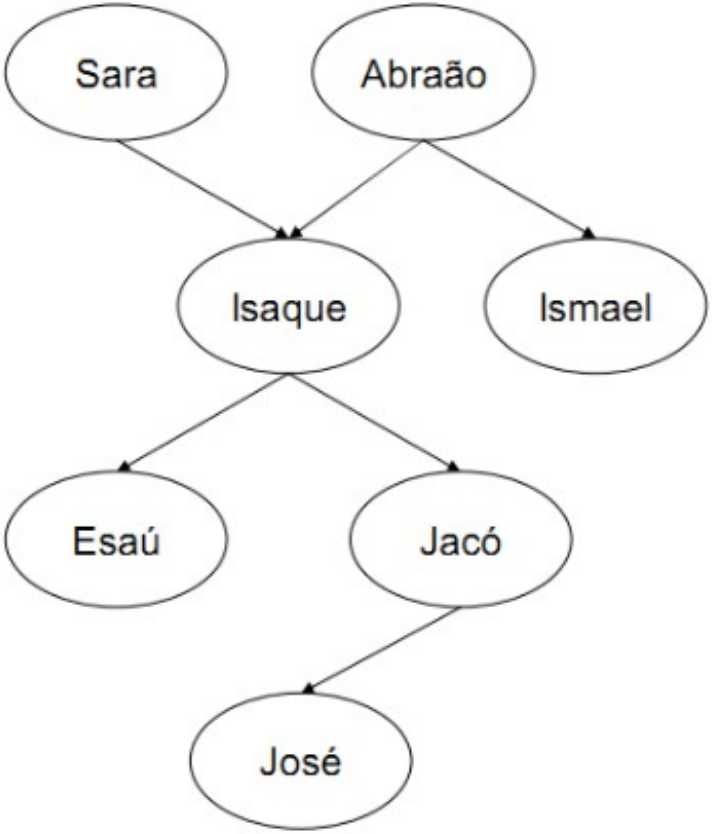
Regras

- Regras expressam fatos que podem ser inferidos logicamente de outros
- Regras simples:
  - crianca(X) :- gosta(X,sorvete).
  - crianca(X) :- not odeia(X,sorvete).

Regras (2)

- Regra com “ou”:
  - avo_fem(X,Z) :- (mae(X,Y),mae(Y,Z)); (mae(X,Y),pai(Y,Z)).
- Regra com lista
  - compra(ana, [roupa, comida, brinquedo])

Base de fatos: exemplo



Base de fatos: exemplo (2)

- Relação progenitor (parent): pai ou mãe

```
progenitor(sara,isaque).        % (fato 1)
progenitor(abraao,isaque).      % (fato 2)
progenitor(abraao,ismael).      % (fato 3)
progenitor(isaque,esau).        % (fato 4)
progenitor(isaque,jaco).        % (fato 5)
progenitor(jaco,jose).          % (fato 6)
```

Base de fatos: exemplo (3)

- Declaração: progenitor(abraao,isaque).
  - Define uma **relação** entre abraão e isaque
  - Tem como **predicado** “progenitor”
  - Duas constantes: “abraao” e “isaque”
- Obs: o predicado também pode ser chamado de **functor**

Execução

- Considerando o gprolog (Gnu Prolog) como interpretador
  - Salvar a base em um arquivo com a extensão .pl (exemplo: fatos.pl)
  - Ao executar o interpretador na linha de comando
  - O prompt de comandos do gprolog começa com: “| ?- ”

Perguntas simples

- A base pode ser importada com a declaração:

```
| ?- [fatos].
```

- Uma vez que a base seja importada, é possível fazer perguntas:

```
| ?- progenitor(ismael,jaco).
  true
| ?- progenitor(isaque,jaco).
  false
```

Perguntas com variáveis

- Quem é o progenitor de Ismael?

```
| ?- progenitor(X, ismael).
  X = abraao
```

- Quais são os filhos de Isaque?

```
| ?- progenitor(isaque,X).
  X = abraao;
  X = jaco;
  false
```

- Obs: no exemplo anterior, existem múltiplas respostas. O comando “;” avança entre elas

Perguntas com variáveis (2)

- Quem são os netos de Abraão?

```
| ?- progenitor(abraao,X), progenitor(X,Y).
  X = esau;
  X = jaco;
  false
```

Regras

- Podemos expandir a base de fatos com a seguinte regra:

```
filho(Y,X) :- progenitor(X,Y).
```

- Com essa regra, é possível fazer novas perguntas relacionadas a pais e filhos

```
| ?- filho(ismael, abraao).
  true
```

Regras (2)

- Algo semelhante pode ser feito para avôs. Na base:

```
avo(X,Y) :- progenitor(X, Z), progenitor(Z, Y).
```

- No prompt de perguntas:

```
| ?- avo(abraao, X).
  X = isaque
  Y = esau;
  X = isaque
  Y = jaco;
  false
```

Declarações Prolog

- Em resumo, Prolog tem três tipos de declarações:
  - Fatos: algo que se assume como verdade
    ```
    progenitor(sara,isaque).
    ```
  - Regra: permite descobrir novos fatos por inferência lógica
    ```
    avo(X,Y) :- progenitor(X, Z), progenitor(Z, Y).
    ```
  - Perguntas: algo que se deseja saber se é verdadeiro
    ```
    | ?- avo(abraao, X).
    ```

Recursividade

- Considere a relação “ancestral”, que engloba progenitores, avôs, bisavôs, etc
- Em Prolog, essa relação pode ser modelada usando recursividade
- Para isso, duas regras são necessárias

```
ancestral(X, Z) :- progenitor(X, Z). % (regra 1)
ancestral(X, Z) :- progenitor(X, Y), % (regra 2)
                   ancestral(Y, Z).
```

Recursividade (2)

- Primeira regra: recupera os progenitores diretos
  - É semelhante à condição de parada em programas imperativos e funcionais
- Segunda regra: recupera os avôs, bisavôs, etc
  - Será aplicada quando a primeira regra falhar
  - A aplicação ocorre quantas vezes necessária

Recursividade: exemplos (1)

- Primeiro exemplo

```
| ?- ancestral(jacó, jose). % (novo fato, fato 7)
  yes
```

- Este é o caso mais simples, em que combinamos a regra 1 (condição de parada) com o fato 6:

```
ancestral(jaco, jose) :-    % (regra 1)
  progenitor(jaco, jose).   % (fato 6)
```

Recursividade: exemplos (2)

- Segundo exemplo:

```
| ?- ancestral(isaque, jose). % (fato 8)
  yes
```

- Aqui, a regra 1 não se aplica diretamente
- Mas podemos combinar os fato 5 e 7 (exemplo anterior) com a regra 2:

```
ancestral(isaque, jose) :-      % (regra 2)
  progenitor(isaque, jaco),     % (fato 5)
  ancestral(jaco, jose).        % (fato 7)
```

Recursividade: exemplos (3)

- Terceiro exemplo:

```
| ?- ancestral(sara, jose). % (fato 9)
  yes
```

- Como no exemplo anterior, podemos combinar fatos antigos (1) e novos (8) com a regra 2

```
ancestral(sara, jose) :-        % (regra 2)
  progenitor(sara, isaque),     % (fato 1)
  ancestral(isaque, jose).      % (fato 8)
```

Recursividade e variáveis

- Por simplicidade, os exemplos contaram com perguntas do tipo “sim” ou “não”
- Mas, pelo mesmo raciocínio, é possível o uso de variáveis

```
| ?- ancestral(X, jose).
  X = jaco;
  X = isaque;
  X = sara;
  X = abraão;
```

Variáveis

- Durante a busca pela solução de uma pergunta, o Prolog atribui valores temporários a cada variável
- Dizemos que uma variável está **instanciada** quando tem um valor atribuído
- Por outro lado, uma variável que ainda não tem valor atribuído é chamada de **livre**

Variáveis anônimas

- São representadas por “_” (sublinha, underline, underscore)
- Sinalizam ao interpretador que o seu valor não é importante
- Exemplo de regra usando variável anônima:
```
tem_filho(X) :- progenitor(X,_).
```
- Outro exemplo:
```
alguem_tem_filho :- progenitor(_,_).
```

Estruturas de dados

- Podemos usar predicados para simular estruturas de dados
- Exemplo:

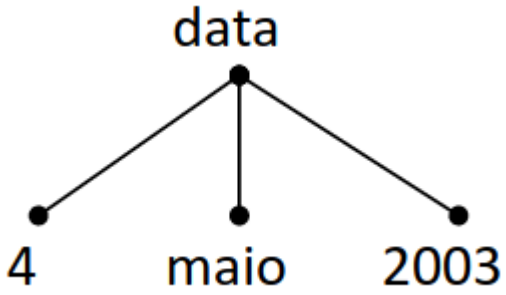
```
data(1,maio,2020).
data(2,maio,2020).
...
```

- Dias de Maio:

```
| ?- data(X,maio,2020).
```

Estruturas de dados (2)

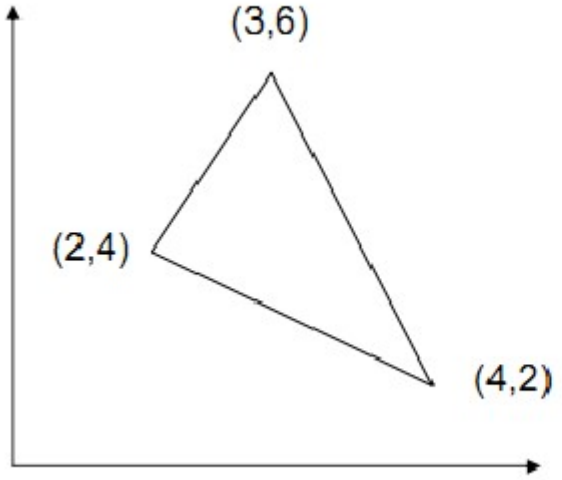
- São representadas por meio de árvores cuja a raiz é um functor
- Exemplo:
  `data(4,maio,2003).`

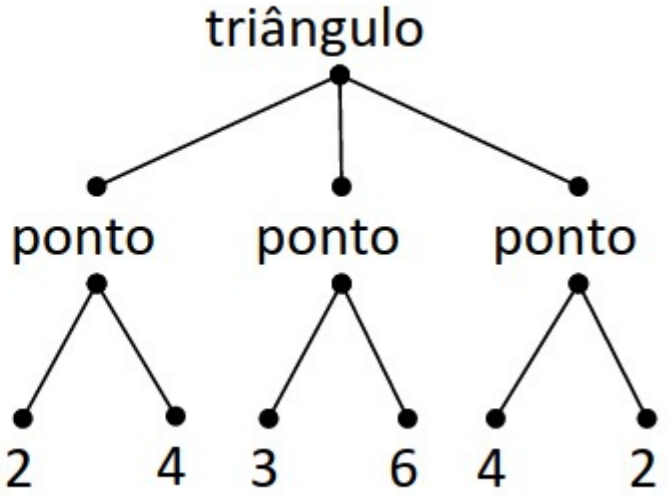


Estruturas de dados (3)

- Estruturas de dados podem ser aninhadas

```
triangulo(ponto(2,4), ponto(3,6), ponto(4,2))
```





Operadores

Operadores Aritméticos	
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Divisão Inteira	//
Resto da Divisão	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

Operador de unificação

- Operador “=” unifica a uma variável dentro de uma declaração

```
| ?- X = 1 + 2.
  X = 1+2
```

- É similar a atribuição no paradigma imperativo
  - Mas não resolve expressões aritméticas
- O escopo da variável é **local** para declaração na qual ela está inserida

Operador “is”

- O operador “is” é capaz de resolver expressões aritméticas

```
| ?- X is 1 + 2.
  X = 3
```

- Caso um valor tem sido previamente atribuído, o operador apenas compara com o valor anterior

```
| ?- X = 5, X is 1 + 2.
  false
```

Unificação

- A unificação pode ocorrer dentro de functores (predicados)
- Variáveis podem ser unificadas com valores ou com outras variáveis:
- Exemplo:

```
| ?- data(D,M,2020) = data(D1,maio,A).
  A = 2020
  D1 = D
  M = maio
```

Unificação (2)

- Outro exemplo:

```
| ?- data(D,M,2020) = data(D1,maio,A),
     data(D,M,2020) = data(15,maio,A1).
  D = 15
  M = maio
  D1 = 15
  A = 2020
  A1 = 2020
```

- Note que D e D1 foram unificadas. Assim, quando D é unificado com 15, D1 também é

Unificação (3)

- A unificação pode também falhar:

```
| ?- data(D,M,2020), data(D1,M1,1948).
  <mensagem de erro>
```

- Note que o processo é tudo ou nada: ou todas as variáveis unificam, ou nenhuma unifica

Regras da Unificação

- Se S e T são constantes, então S e T unificam somente se são o mesmo.
  Ex.: jaco unifica com jaco
- Se S for uma variável e T for qualquer termo, então unificam e S é instanciado para T.
  Ex.: abraao unifica com X (slide 15)
- Se S e T são estruturas, elas unificam somente se tem o mesmo functor e todos os argumentos se unificam

Operadores de Unificação

- **X = Y**: X unifica com Y, é verdadeiro quando dois termos são o mesmo.

```
| ?- pai(X, Y) = pai(X, Z).
  yes
```

  - Se um dos termos é uma variável, o operador = causa a instanciação da variável.
  - O exemplo, retorna true pois Y e Z podem ser unificados
- **X \= Y**: X não unifica com Y

Operadores de Unificação (2)

- **X == Y**: X é literalmente igual a Y (igualdade literal)

```
| ?- pai(X, Y) == pai(X, Z).
  no
```

  - Verdadeiro se os termos X e Y são idênticos
  - Precisam ser idênticos (estrutura, argumentos, nomes das variáveis)
- **X \== Y**: X não é literalmente igual a Y que é o complemento de X==Y

Operadores de Unificação (3)

```
| ?- 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
| ?- g(X,f(a,Y)) == g(X,f(a,Y)).
  yes
```

Predicados especiais

- var(X): X é uma variável não instanciada
- nonvar(X): X não é uma variável ou X é uma variável instanciada
- atom(X): X é uma sentença atômica
- integer(X): X é um inteiro
- float(X): X é um número real
- atomic(X): X é uma constante
- compound(X): X é uma estrutura

Predicados especiais (2)

```
| ?- var(Z), Z = 2.
  Z = 2
| ?- Z = 2, var(Z).
  no
| ?- integer(Z), Z = 2.
  no
| ?- Z = 2, integer(Z),
  nonvar(Z).
  Z = 2
  yes
| ?- atom(3.14).
  no
```

```
| ?- atomic(3.14).
  yes
| ?- atom(jaco).
  yes
| ?- atom(p(1)).
  no
| ?- compound(2+X).
  yes
```

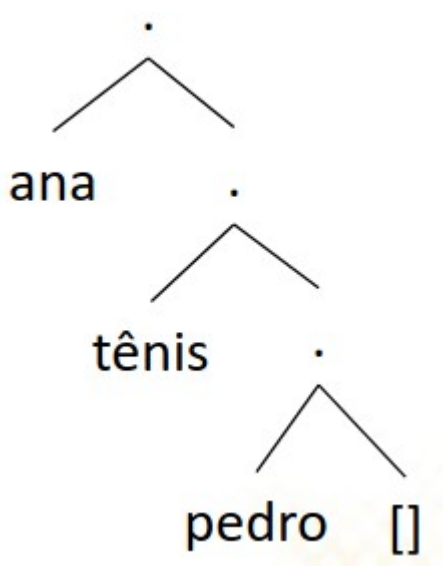
Listas

- Prolog fornece suporte a listas em duas notações:
  - Completa: .(a, .(b, .(c, [])))
  - Simplificada: [a,b,c]
- Exemplo:

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

Listas (2)

- Listas são representadas em forma de árvore.
- Considere a lista: .(ana, .(tênis, .(pedro, [])))
- Ela pode ser reprsentada por:



Listas (3)

- Listas podem ser aninhadas

```
| ?- Lista = [cozinha, [mesa, cadeira], sala,
              [sofa, estante]].
```

- Uma lista vazia é denotada por “[ ]”
- Toda a lista que não é vazia pode ser dividida em cabeça e cauda
  - A cabeça (head - H) sempre tem um elemento
  - A cauda (tail – T) pode ser vazia e pode ter vários elementos

Listas (4)

- A cabeça da lista e a cauda podem ser separadas das seguinte formas:

```
| ?- [H | T] = [a, b, c].
  H = a
  T = [b,c]

| ?- L = [a, b, c], L = [H | T].
  H = a
  T = [b,c]
  L = [a,b,c]
```

Listas (5)

- Da mesma forma, a partir de uma cabeça e uma cauda, é possível compor uma nova lista

```
| ?- H = a, T = [b,c], L = [H | T].
  H = a
  T = [b,c]
  L = [a,b,c]
  yes
```

Listas (6)

- Note que o exemplo anterior usa “|”. O operador “,” criaria uma lista aninhada:

```
| ?- H = a, T = [b,c], L = [H, T].
  H = a
  T = [b,c]
  L = [a,[b,c]]
  yes
```

Buscas

- Como exemplo, vamos criar um functor “pertence” para buscar elementos em uma lista
  - Argumentos: um elemento e uma lista em questão
  - Retorno: true se o elemento pertence à lista
  - Usaremos duas regras: uma para a cabeça e outra para a causa (recursiva)

Buscas (2)

- Functor:

```
% regra 1: X pertence a lista se
% quando ele é a cabeça da lista
pertence(X, [X|T]).

% regra 2: X pertence a lista se
% quando ele é a cabeça da cauda
% ou a cabeça da cauda da cauda
% ou a cabeça da cauda da cauda da cauda, etc
pertence(X, [H|T]) :- pertence(X, T).
```

Buscas (3)

- Exemplo de uso:

```
| ?- pertence(a, [a, b, c]).
  true
| ?- pertence(b, [a, b, c]).
  true
| ?- pertence(c, [a, b, c]).
  true
| ?- pertence(d, [a, b, c]).
  false
| ?- pertence(a, [a, b, a, c]).
  true;
  true
```

Buscas (4)

- Mais exemplos:

```
| ?- pertence(X, [a, b, c]).
  X = a;
  X = b;
  X = c;

| ?- pertence(d, [a, b, X, Y, c]).
  X = d;
  Y = d;
```

Buscas (5)

- Retornando o último elemento de uma lista:

```
% um elemento é o último da lista se ela é unitária
ultimo(X, [X]).
% caso contrário, checar o último na cauda da lista
ultimo(X, [H|T]) :- ultimo(X, T).
```

- Uso:

```
| ?- ultimo(a, [a, b, c]).
  no
| ?- ultimo(c, [a, b, c]).
  true
| ?- ultimo(X, [a, b, c]).
  X = c
```

Commando assert

- A linguagem Prolog foi planejada de tal forma que suas queries (perguntas) são feitas interativamente
- Já seus fatos e regras são declarados em um arquivo separado (base de fatos)
- O comando assert permite inserir fatos diretamente do prompt de perguntas

Commando assert (2)

- No dialeto do Gprolog, o comando aparece como **assertz**. Exemplo:

```
| ?- pai(adao, abel).
  no
| ?- assertz(pai(adao, abel)).
  yes
| ?- pai(adao, abel).
  yes
```

- Obs: para remover fatos da base, é possível usar o comando **retract**

Comando Trace

- Trace é um comando para verificar como uma inferência é feita
- No dialeto que estudamos, o comando trace aparece como spy. Exemplo:

```
| ?- spy(pertence).
  yes
| ?- pertence(c, [a, b, c, d]).
  % verifique a saída no terminal
| ?- nospy(pertence).
  yes
```

Lógico vs Imperativo

- O paradigma declarativo é o mais diferente dos paradigmas analisados
- Guardadas as devidas proporções, podemos trazer paralelos com estruturas imperativas
- Já verificamos que a unificação é similar à atribuição das linguagens imperativas (operadores “=”, “is”).

Lógico vs Imperativo (2)

- De forma semelhante, predicados podem ser usado como structs para estruturas de dados
- Listas podem se comportar de forma semelhante aos arrays
- A seguir, veremos como emular funções, estruturas de decisão e laços

Funções e If

- Emulando uma função com uma regra:

```
main :-
  write('regras pode ser usadas '), nl,
  write('emulando comandos de uma função'), nl,
  write('separados por vírgulas'), nl,
  write('e terminados em ponto final').
```

- Emulando um if com duas regras:

```
maior_igual_5(X) :-
  X >= 5, write('maior igual a 5').
maior_igual_5(X) :-
  X < 5, write('menor que 5'), false.
```

Laços

- Emulando laços com recursão, estratégia 1:

```
% condição de parada
loop(0).
% laço recursivo
loop(N) :- N>0,
  write('value of N is: '), write(N), nl,
  S is N-1, loop(S).
```

Laços (2)

- Emulando laços com recursão, estratégia 2:

```
% condição de parada: atingiu o valor de N
count(N, N) :-  write(N), nl.
% caso recursivo: ainda não atingiu o valor de N
count(Counter, N) :-
  Counter < N,
  write(Counter), nl,
  NextCounter is Counter + 1,
  count(NextCounter, N).
```

Referências

- Adaptado do material de aula dos professores:
  - Edirlei Soares de Lima (PUC – Rio)
  - Matheus Gonçalves Ribeiro (UNESP - IBILCE)