Paradigmas de Programação

Paradigma Lógico com Prolog (Parte 2)

Introdução

Este material contém a sequência das notas de aula sobre o Paradigma Lógico. O objetivo é aprofundar os conhecimentos no Paradigma Lógico e na linguagem Prolog no que diz respeito ao tipo de dados e seus subtipos, vinculação de variáveis a valores (unificação), operador `is`, listas, recursividade, o retrocesso (*backtracking*), entrada e saída e outros predicados, mostrando como esses elementos permitem a construção de programas que exploram a lógica e a manipulação de estruturas de forma declarativa.

Paradigma Lógico com Prolog

Como já visto nas Notas de Aula sobre o Paradigma Lógico, linguagens de programação do paradigma lógico fazem parte do paradigma declarativo e Prolog é a linguagem mais popular que representa esse paradigma. Além disso, Prolog se baseia nos conceitos de fatos, regras e consultas.

DESAFIO!

Identifique onde está(ão) o(s) fato(s) regra(s) e consulta(s):

```
pessoa(ricardo).
pessoa(joao).
naturalidade(ricardo, pelotas, rs).
```

```
naturalidade(joao, blumenau, sc).
catarinense(X) :- pessoa(X), naturalidade(X, _, sc).
?- catarinense(ricardo).
```

Tipos

Conforme visto nas Notas de Aula sobre o Paradigma Lógico, Prolog é uma linguagem dinâmica e fracamente tipada. Em Prolog, há somente um tipo de dado, o termo (`Term`), que possui os seguintes subtipos: constantes (átomos ou números), variáveis e termos compostos.

Constantes: átomos ou números. Exemplos: `2`, `-4.05`, `ricardo`.

Átomos: em Prolog, um átomo é um único item de dados. Pode ser de um dos quatro tipos:

1. Um átomo de *string*, como '`eu sou um texto`' ou
2. Um símbolo, como pai, joão e ricardo, em pai(joão, ricardo). Átomos deste tipo devem começar com uma letra minúscula. Eles podem incluir dígitos (após a letra minúscula inicial) e o caractere sublinhado (_).
3. A lista vazia `[]` (CHU, s.d.). Esta é estranha: outras listas não são átomos. Se um átomo é algo que não pode ser divisível em partes (o significado original da palavra átomo, embora subvertido pela física subatômica), então `[]` ser um átomo é menos surpreendente, pois certamente não é divisível em partes.
   Nota: no SWI-Prolog versão 7, a lista vazia não é um átomo, mas uma constante única e equivalente somente a ela mesma (SWI-PROLOG, 2025).
4. Sequências de caracteres especiais, como `<---->`, `...`, `===>`. Ao usar átomos desse tipo, é necessário ter alguns cuidados para evitar o uso de *strings* de caracteres especiais com significado predefinido, como os símbolos `:-`, `!`, e vários operadores aritméticos e de comparação.
   Números, em Prolog, **não** são considerados átomos.

Variáveis: letras maiúsculas ou *underlines* (_). A variável _ é dita *anônima* e pode assumir qualquer valor válido. Se uma variável se repete em uma cláusula, ela assume o mesmo valor em todas as suas ocorrências, exceto se for uma variável anônima (*underline*).

Termos compostos, por vezes traduzidos como **Estruturas**, são "formados por um funtor seguido de componentes separadas por vírgula e colocadas entre parênteses" (MEIDANIS, 2011). São utilizados para expressar fatos e regras. Em Prolog, expressões aritméticas como `-5 + 3.5 * 3.` também são tratadas como termos compostos, pois são representadas internamente exatamente na forma citada. Isso significa que a expressão pode ser escrita como `+(-5, *(3.5, 3)).`, mantendo o mesmo significado.

Exemplo:

```
anoFabricação('Polo', 2020).
```

Estruturas podem ser argumentos/parâmetros de fatos, formando assim termos mais complexos. Exemplo: Supondo que a base de conhecimento contém o fato abaixo:

```
pessoa(nome('Ricardo', 'Ladeira'), data(14, mai, 1989), masculino, 1.80).
```

Qual seria a resposta para a consulta abaixo?

```
?- pessoa(nome('Ricardo',X), _, _, _).
```

Funtor (*functor*)

Em Prolog, o funtor é usado para se referir ao átomo no início de uma estrutura, juntamente com sua aridade (quantidade de argumentos que ela recebe). Por exemplo, em `naturalidade(ricardo, pelotas, rs).`, o funtor é `naturalidade/3`.

Existe um predicado chamado `functor` com três argumentos (ou seja, com assinatura `functor/3`): `functor(Termo, Nome, Aridade)`, que retorna verdadeiro (`true`) se o `Termo` for um termo do funtor (SWI-PROLOG, 2025).

Exemplos:

A consulta:

```
?- functor(naturalidade(ricardo, _, _), X, Y).
```

Retorna :

```
X = naturalidade,
Y = 3
```

A consulta:

```
functor(1 + 9, X, Y).
```

Retorna:

```
X = (+),
Y = 2
```

Igualdade (Unificação)

Prolog realiza a igualdade por meio da técnica de unificação. Portanto, igualdade e unificação são sinônimos em Prolog. A unificação realiza a vinculação (também chamada de *binding*) de variáveis e valores.

Pode-se usar o símbolo de igualdade para testar dois termos quaisquer. Exemplo:

```
?- gato(A) = gato(rodolfo).
```

No caso acima, a consulta retorna `A = rodolfo`, pois é possível vincular a variável `A` ao valor `rodolfo`.

Algumas regras para unificação:

- constante com constante: se for a mesma.
- constante com variável: sempre possível.
- constante com estrutura: nunca.
- variável com variável: sempre possível.
- variável com estrutura: sempre possível.
- estrutura com estrutura: possível quando for o mesmo funtor, mesma aridade e houver unificação dos argumentos.

Operador is (ou função `is/2`)

`is` é um operador infixo (fica entre os dois operandos, ou seja, tem aridade 2) que serve para uma das duas situações abaixo:

1) Avaliar uma expressão e instanciar uma variável com o resultado dessa expressão. Exemplo:

Base de conhecimento:

```
imc(P, A, R) :- R is P/(A*A).
```

Consulta:

```
?- imc(72, 1.8, R).
```

Outro exemplo:

```
areaRetangulo(B, A, R) :- R is B*A.
```

Consulta:

```
?- areaRetangulo(3.0, 4.0, R).
```

2) Comparar dois valores. Exemplo: `?- 9 is 8-1+2.`

Ao lado direito do operador `is` deve sempre aparecer uma expressão aritmética envolvendo apenas números ou variáveis instanciadas com números. Do lado esquerdo pode aparecer uma variável não instanciada, que será então instanciada com o resultado da expressão, ou um número, ou uma variável instanciada a um número, caso em que is testa se o lado esquerdo e direito são iguais, servindo assim como operador de igualdade numérica (adaptado de MEIDANIS, 2011).

Lista

A lista é uma estrutura de dados composta por duas partes: cabeça e corpo (também chamado de cauda). Sendo assim, na lista `[a, b]`, diz-se que `a` é a cabeça e `[b]` é o corpo. Já a lista vazia é detonada por `[ ]`.

Exemplo:

Base de conhecimento:

```
contatos('ricardo.ladeira@ifc.edu.br', ['5381272727', '4799999999',
'4788888888']).
```

Consultas:

```
?- contatos(_, X).
?- contatos(_, [_, X, _]).
?- contatos(_, [X|Y]). % O caractere "|" separa a cabeça e o corpo.
```

Listas também podem conter listas. Esta lista é válida: `[a, b, c, [d, e, f, [g, h, i]]]`.

A junção de listas pode ser feita com `append`. Exemplo:

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

Recursividade

A recursividade é uma técnica de programação na qual um predicado se define, direta ou indiretamente, em termos dele mesmo, resolvendo um problema grande ao dividi-lo em subproblemas menores, até atingir um caso base que encerra as chamadas.

Em Prolog, usa-se recursividade com frequência porque não há laços de repetição (`for`, `while`, etc.).

Assim como em linguagens estruturadas, uma porção de código recursiva precisa ter um caso base e um caso recursivo. Exemplo:

Base de conhecimento:

```
% Caso base: se a lista está vazia, seu tamanho é 0.
tamanho([], 0).
% Caso recursivo: ignora a cabeça, conta o tamanho da cauda e soma 1
tamanho([_|T], N) :- tamanho(T, N1), N is N1 + 1.
```

Consulta:

```
?- tamanho([a, b, c, d, e], TAM).
```

Outro código comum para exemplificar a recursividade em Prolog é feito através da declaração do predicado `pertence`, que serve para identificar se um elemento pertence a uma lista. Esse predicado é definido como segue:

```
pertence(X, [X|_]).
```

```
pertence(X, [_|Y]) :- pertence(X, Y).
```

O código acima verifica se o primeiro argumento (elemento `X`) pertence ao segundo argumento (a lista). Primeiro, cria um predicado que restará verdadeiro se o elemento pesquisado for igual ao elemento da cabeça da lista. O segundo predicado verifica se `X` está na cauda da lista. Ali, `pertence` é chamado recursivamente com a lista sem a cabeça inicial, e assim sucessivamente até obter uma resposta (`true` ou `false`).
Exemplo de consulta:

```
?- pertence('Jose', ['Ana', 'Maria', 'Joao', 'Jose']).
```

EXERCÍCIOS

1. Na definição de `pertence`, quem é o caso base e o caso recursivo?
2. O predicado pertence é definido por um fato e uma regra. Essa afirmação é verdadeira ou falsa? Justifique.

Backtracking (retrocesso)

Ao realizar uma pergunta, todas as metas precisam ser satisfeitas. Por exemplo:

```
?- planta(maria, X), planta(joao, X).
```

Procura-se por algum `X` que satisfaça duas chamadas ao predicado `planta`, ou seja, algo que `maria` e `joao` plantem em comum. Genericamente falando, se uma meta *k* for satisfeita, a variável é instanciada com o valor válido e passa-se para a meta seguinte (*k+1*). Se não houver meta seguinte, ou se todas as metas já estiverem satisfeitas, Prolog informa o resultado.

Se a meta *k* não for satisfeita, a variável instanciada é liberada novamente e retrocede-se à meta imediatamente anterior (*k-1*). A isto damos o nome de *backtrack* (retroceder).

Voltando ao exemplo, suponha a seguinte base de conhecimento:

```
planta(maria, uva).
planta(maria, abobora).
```

```
planta(maria, salsinha).
planta(joao, salsinha).
```

Agora, ao realizar a consulta (`?- planta(maria, X), planta(joao, X).`), sabe-se que:
A meta 1 é cumprida por `planta(maria, uva)`, então passa-se à meta 2.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, abobora)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(joao, salsinha)`. Tenta-se a próxima.
Não há próxima. Ocorre o retrocesso, libera novamente a variável `X` e tenta-se cumpri-la com o predicado seguinte.
A meta 1 é cumprida por `planta(maria, abobora)`. Passa-se novamente à meta 2.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, abobora)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(joao, salsinha)`. Tenta-se a próxima.
Não há próxima. Ocorre o retrocesso, libera novamente a variável `X` e tenta-se cumpri-la com o predicado seguinte.
A meta 1 é cumprida por `planta(maria, salsinha)`. Passa-se novamente à meta 2.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, abobora)`. Tenta-se a próxima.
A meta 2 não é cumprida por `planta(maria, uva)`. Tenta-se a próxima.
A meta 2 é cumprida por `planta(joao, salsinha)`.

Nesse contexto, é importante também o conceito de *cut* (corte). O corte existe para limitar a busca por alternativas válidas. Esta ação pode poupar processamento e memória, e é útil quando procura-se por alguma solução específica ou um conjunto de soluções específicas, não todas. O corte é feito com a exclamação (`!`).

Exemplo 1. Suponha a seguinte base de conhecimento:

```
f(X, 1) :- X < 0.
f(X, 2) :- X = 0.
```

```
f(X, 3) :- X > 0.
```

Nota-se que todas as alternativas são exclusivas, ou seja, não é possível que um valor de `X` satisfaça mais de uma regra. No entanto, ao realizar a consulta abaixo, obtém-se como retorno `Y = 1` e a opção de *next*, ou seja, após encontrar uma resposta válida ele tenta realizar o retrocesso.

```
?- f(-30, Y).
```

O programador sabe que este retrocesso é infrutífero, e pode utilizar o corte para poupar processamento. Portanto, a consulta pode ser feita da seguinte forma:

```
?- f(-30, Y), !.
```

Realizando a consulta desta forma, não há retrocesso. No entanto, o corte não é exclusivo de consultas e também pode ser utilizado em regra, como feito no exemplo a seguir.

Exemplo 2 (BARANAUSKAS, 2022).

```
f(X, 0) :- X < 3. % Regra 1
f(X, 2) :- 3 =< X, X < 6. % Regra 2
f(X, 4) :- X >= 6. % Regra 3
```

Consulta:

```
?- f(1, Y), 2<Y.
```

No primeiro caso, a primeira meta é atingida pela regra 1, mas a segunda meta não é atingida. Como os três predicados são mutuamente exclusivos, não seria necessário testar as regras 2 e 3, pois essas são claramente falsas. As regras 1 e 2 podem ser alteradas, ficando da seguinte forma:

```
f(X,0) :- X < 3, !. % Regra 1
f(X,2) :- 3 =< X, X < 6, !. % Regra 2
f(X,4) :- X >= 6. % Regra 3
```

O teste `3 =< X` é desnecessário, pois quando não cair na condição um (`X < 3`), certamente será atingido. Portanto, pode ser removido:

```
f(X,0) :- X < 3, !. % Regra 1
f(X,2) :- X < 6, !. % Regra 2
```

```
f(X,4) :- X >= 6. % Regra 3
```

Agora, o mesmo pode ser feito com a regra 3, pois ela também é o oposto da regra 2 e será atingida se a regra 2 não for:

```
f(X, 0) :- X < 3, !. % Regra 1
f(X, 2) :- X < 6, !. % Regra 2
f(X, 4). % Regra 3
```

Observação: tal código acusa erro e sugere-se alterar a regra 3 para `f(_, 4).`

Sabendo disso, como alterar o código do exemplo 1 para que as regras utilizem cortes e estejam mais eficientes? Resposta:

```
f(X, 1) :- X < 0, !.
f(X, 2) :- X = 0, !.
f(_, 3).
```

ANALOGIA PARA PENSAR

Embora não exista equivalência funcional, o *cut* (corte) interrompe o fluxo de execução assim como o comando *break* em linguagens imperativas. Porém, enquanto o *break* encerra laços ou blocos lineares, o *cut* corta o retrocesso no mecanismo de busca de soluções do Prolog, descartando alternativas que poderiam ser exploradas no *backtracking*.

Entrada e Saída (E/S)

Já foi visto que é possível carregar um arquivo de extensão `pl` (exemplo: `times.pl`) utilizando `[arquivo]`, desde que ele esteja no mesmo diretório. É possível também carregar mais arquivos, bastante separá-los por vírgula:

```
?- [arquivo1, arquivo2, arquivo3].
```

Se o arquivo estiver em outro diretório, é possível carregá-lo na linha de comando ao abrir o `swipl`:

```
swipl -s ../arquivo.pl
```

```
prolog -s ../arquivo.pl
```

Para fazer a leitura de dados da entrada padrão, pode-se usar o predicado `read(X)`. Exemplos:

```
par :- read(N), 0 is mod(N, 2).
% impar :- read(N), 1 is mod(N, 2).
impar :- not(par).
```

Observação: nota-se que o terminal fica em prontidão com "`|:`". Quando isso aparecer, significa que o terminal aguarda ação do usuário.

Para fazer a escrita de dados na saída padrão, pode-se usar o predicado `write(X)`. Exemplo: `write('Olá'), write(', '), write('Mundo!'), nl, write('Como vai você?').`

No exemplo acima, nota-se um átomo diferente: `nl`. Esse átomo nomeia um predicado embutido (*built-in*, ou seja, já vem definido na própria linguagem) de aridade 0 que insere uma quebra de linha na saída.

Outros Predicados (lista não exaustiva)

`atom(X)`: satisfeito quando `X` é um átomo.
`?- atom(9).` retorna `false`.

`atomic(X)`: satisfeito quando `X` é um valor atômico (valor já em sua forma final, que não pode mais ser avaliado).
`?- atomic(9+1).` retorna `false`.

`number(X)`: satisfeito se `X` for um número.
`?- number(9).` retorna `true`.

`random(X)`: produz um número aleatório entre 0 e X-1, inclusive.
`?- random(X).` retorna `0.10140376991569817` (exemplo).

Conclusão

Este material aprofundou os conhecimentos em Programação Lógica e na linguagem Prolog, destacando a manipulação de tipos de dados e estruturas, a vinculação de variáveis por meio da unificação, o uso do operador `is` para cálculos, a construção e processamento de listas, a aplicação da recursividade, o funcionamento do *backtracking* para exploração de alternativas, os predicados para entrada e saída e o emprego de predicados auxiliares, recursos que permitem resolver problemas usando a lógica.

Exercícios

1) (Adaptada do ENADE 2021) O paradigma de programação em lógica constitui-se como um processo de definição de relações em que se constroem fatos e regras sobre os elementos e suas relações. A ativação dos programas acontece por meio de consultas (ou perguntas) sobre o relacionamento definido. Ao construir-se um banco de dados referente a uma família, inicia-se pelas relações de parentesco, tais como:
   `pai(joão, maria)` convencionado que João é pai de Maria
   `mãe(maria, luiz)` convencionado que Maria é mãe de Luiz

   As cláusulas, sem condições, definem os fatos sobre o domínio do problema. De outra maneira, é possível definir as regras que são cláusulas com condições:
   `avô_materno(joão,luiz) :- pai(joão,maria), mãe(maria,luiz).`

   Nesse contexto, avalie as afirmações a seguir.

   I. A consulta `?- mãe(maria, X)` retorna verdadeiro conforme identificado na cláusula.
   II. A conclusão *João é avô materno de Luiz* é identificada como cabeça da cláusula.
   III. A regra `irmão(X, Y) :- pai(Z, X), pai(Z, Y)` é uma regra genérica que define irmãos por parte de pai.
   IV. A regra `avô_materno(X, Y) :- pai(X, Z), mãe(Z, Y)` é uma regra válida para o programa.

É correto apenas o que se afirma em

a) I e II.
b) II e III.
c) II e IV.
d) I, II e IV.
e) I, III e IV.

2) Resolva a questão 1 da p. 20 de Meidanis (2011).
3) Resolva a questão 2 da p. 20 de Meidanis (2011).
4) A questão 2 da p. 20 de Meidanis (2011) contém um erro. Identifique o erro e refaça a questão anterior de forma a corrigi-lo.
5) Utilizando o operador `is`, implemente a regra da potenciação como `pot/3`, sendo seus argumentos, em ordem, `B` (base), `E` (expoente), `R` (resultado).
6) Pratique o uso dos predicados presentes neste material.

Referências

BARANAUSKAS, José Augusto. Conceitos avançados de prolog. Material didático. Departamento de Física e Matemática – FFCLRP-USP. Disponível em: https://dcm.ffclrp.usp.br/~augusto/teaching/ia/IA-Prolog-Conceitos-Avancados.pdf. Acesso em: 13 ago. 2025.

CHU, Chunbo. Introduction to Prolog. COMP 205. Disponível em: https://cs.franklin.edu/~chuc/COMP205/notes/week13-Prolog.pptx. Acesso em: 13 ago. 2025.

MEIDANIS, João. MC346 - Paradigmas de programação: Prolog. Disponível em: https://www.ic.unicamp.br/~meidanis/courses/mc346/2017s2/prolog/apostila-prolog.pdf. Acesso em: 15 ago. 2025.

SWI-Prolog. Datatypes. Disponível em: https://www.swi-prolog.org/datatypes.html. Acesso em: 13 ago. 2025.

SWI-Prolog. Predicate `functor/3`. Disponível em: https://www.swi-prolog.org/pldoc/man?predicate=functor/3. Acesso em: 13 ago. 2025.