

Programação Lógica

Regras, aritmética e estruturas de dados

Prof. Edson Alves

Faculdade UnB Gama

2020

Sumário

1. Regras
2. Aritmética
3. Manipulação da base de dados
4. Recursão
5. Estruturas de Dados

Regras

- ▶ Um predicado é definido por cláusulas, as quais podem ser fatos ou regras
- ▶ Uma regra é uma consulta armazenada. A sintaxe é

`?- head :- body.`

onde **head** é um predicado, `:-` é o pescoço (*neck symbol*), lido como “se” e **body** é composto por uma consulta

- ▶ Exemplo: a regra ‘capital/2’, que lista as capitais X de uma região Y , pode ser definida por

`capitais(X, Y) :- regioao(Y, Z), cidade(X, Z), capital(X).`

- ▶ Consultas possíveis seriam:

`?- capitais(X, norte).`

`?- capitais(X, Y).`

- ▶ Uma mesma regra pode ser definida múltiplas vezes, cada uma com um corpo diferente

Processamento das regras

- ▶ Uma regra é processada da seguinte maneira:
 1. Inicialmente, o padrão do objetivo é unificado com a cabeça (*head*) regra
 2. Se a unificação é bem sucedida, inicia-se uma consulta com os objetivos listados no corpo da regra
- ▶ Deste modo, regras permitem consultas em múltiplos níveis
- ▶ O primeiro nível é composto pelos objetivos iniciais
- ▶ O segundo são os objetivos que aparecem no corpo da regra
- ▶ No corpo da regra pode ser utilizadas novas regras, aumentando o nível da consulta
- ▶ Regras podem ser usadas para definir conexões não-direcionadas

Processamento das regras

- ▶ Por exemplo:

```
connected(X, Y) :- edge(X, Y).  
connected(X, Y) :- edge(Y, X).
```

- ▶ Note o “ou” implícito na definição das duas regras
- ▶ Regras ou predicados que sempre falham não pode ser utilizadas em consultas compostas, porque não transferem o fluxo de execução adiante, via porta **exit**
- ▶ Nestes casos, é útil adicionar uma nova definição a tal regra ou predicado que retorna verdadeiro sempre.
- ▶ Um predicado ou regra como uma única variável (sem corpo) é sempre bem sucedido
- ▶ Nestes caso, pode se usar uma variável especial, denominada variável anônima, representada pelo símbolo ‘_’ (*underscore*)

Exemplo de definição regras e consultas envolvendo regras

```
1 male(homer).
2 male(bart).
3
4 female(marge).
5 female(lisa).
6 female(maggie).
7
8 father(homer, bart).
9 father(homer, lisa).
10 father(homer, maggie).
11
12 mother(marge, bart).
13 mother(marge, lisa).
14 mother(marge, maggie).
15
16 siblings(X, Y) :- father(F, X), father(F, Y).
17 brothers(X, Y) :- male(X), siblings(X, Y).
18 sisters(X, Y) :- female(X), siblings(X, Y).
19
20 % ?- sisters(X, bart).
21 % ?- siblings(X, maggie). -- Explique o resultado desta consulta!
```

Unificação em regras

1. A unificação é o processo de *pattern matching* do Prolog, utilizada na comunicação entre regras e predicados
2. A execução é controlada pelo mecanismo de *backtracking* do Prolog
3. `fail/0` pode ser usado para forçar o retorno do *backtracking* (via porta `fail`)
4. Pode se forçar o sucesso de um predicado ou regra por meio de uma definição extra com variáveis anônimas e sem corpo
5. O *backtracking* substitui os laços de outras linguagens
6. O *pattern matching* substitui os testes condicionais e as estruturas de seleção
7. As regras podem ser testadas individualmente, permitindo o desenvolvimento modular
8. Regras que usam outras regras encorajam a abstração de procedimentos e dados

Operadores aritméticos

- ▶ Para computar expressões aritméticas, Prolog disponibiliza o predicado pré-definido `is`, cuja sintaxe é:

`X is <expressão aritmética>`

- ▶ A variável `X` recebe o valor da expressão, e é desatada no *backtracking*
- ▶ As expressões são semelhantes às utilizadas em outras linguagens
- ▶ Exemplos de expressões:

`?- X is 2 + 2.`

`?- X is 3*4 + 2.`

- ▶ Parêntesis podem ser utilizado para evitar ambiguidades e alterar a ordem de precedência dos operadores

`?- X is 3*(4 + 2).`

Operadores relacionais

- Para evitar que os operadores relacionais se assemelhem às setas, a ordem dos símbolos é diferente do usual:

```
X > Y
Y < Y
X >= Y
X =< Y
X \= Y           % X é diferente de Y
```

- Exemplos de regras baseadas em expressões aritméticas:

```
juros_simples(X, P, J, T) :-
    X is P * (1 + J * T).
```

```
juros_compostos(X, P, J, T) :-
    X is P * (1 + J) ** T.
```

```
% ?- juros_simples(X, 100, 0.12, 24).
```

```
% ?- juros_compostos(X, 100, 0.12, 24).
```

Exemplo de uso de operadores aritméticos e relacionais

```
1 % Calcula as raízes reais de um polinômio
2 %
3 %  $p(x) = ax^2 + bx + c$ 
4
5 root_signal(S, Delta) :-
6     Delta >= 0,
7     S is 1.
8
9 root_signal(S, Delta) :-
10    Delta > 0,
11    S is -1.
12
13 roots(X, A, B, C) :-
14    A \= 0,
15    Delta is B ** 2 - 4*A*C,
16    root_signal(S, Delta),
17    X is (-B + S*sqrt(Delta))/(2*A).
18
19 % ?- roots(X, 1, -5, 6).
20 % ?- roots(X, 1, 0, 1).
21 % ?- roots(X, 0, 2, 4).
```

Manipulação da base de dados

- ▶ Prolog permite a manipulação direta da base de dados por meio de predicados pré-definidos:
 1. `asserta(X)`: adiciona a cláusula `X` como primeira cláusula para o seu predicado. Como as rotinas de I/O, sempre falha no *backtracking* e não desfaz seu trabalho
 2. `assertz(X)`: igual a anterior, mas adiciona como última cláusula do predicado
 3. `retract(X)`: remove a cláusula `X` da base de dados
- ▶ Para remover uma cláusula, é preciso marcar o predicado como dinâmico, antes da definição do mesmo
- ▶ A sintaxe para tal é

```
:- dynamic  
   pred/N.
```

Prolog e variáveis globais

- ▶ Não há variáveis globais em Prolog: as variáveis são locais às cláusulas
- ▶ A base de dados “substitui” as variáveis globais
- ▶ Ela permite que as cláusulas compartilhem informações entre si
- ▶ **asserts** e **retracts** são as ferramentas para manipular os dados da base, dados estes que correspondem à variáveis globais
- ▶ Naturalmente, este recurso deve ser utilizado com parcimônia e cuidado, pois ele modifica o estado do programa
- ▶ Alguns programadores tentam eliminar dados globais e o uso de **asserts** e **retracts** em seus códigos Prolog

Programas sem variáveis globais

- ▶ É possível escrever programas que não modificam a base de dados, o que elimina o problema das variáveis globais
- ▶ Isto pode ser feito passando as informações necessárias por meio dos argumentos dos predicados
- ▶ A versão de `assert` apresenta a seguir desfaz seu trabalho no *backtracking*:

```
backtracking_assert(X):-  
    asserta(X).  
backtracking_assert(X):-  
    retract(X), fail.
```

- ▶ Inicialmente, a primeira cláusula é executada
- ▶ Se um objetivo posterior falhar, o *backtracking* vai tentar a segunda cláusula, que desfaz o trabalho da primeira e falhar, resultando no efeito desejado

Recursão em Prolog

- ▶ Em Prolog, a recursão acontece quando um predicado contém um objetivo que se refere ao próprio predicado
- ▶ Como a cada consulta Prolog usa o corpo da regra para criar uma nova consulta com novas variáveis, a recursão acontece naturalmente
- ▶ Uma chamada recursiva é composta por duas partes:
 1. casos-base, e
 2. chamada recursiva
- ▶ Os casos-base são condições limítrofes (de contorno) que são sabidamente verdadeiras
- ▶ O caso recursivo resolve o problema por meio de nova chamada da regra, em uma versão reduzida do problema
- ▶ A cada etapa da recursão, os casos-base são verificados: caso ocorram, a recursão termina; caso contrário, a recursão continua

Exemplo de recursão em Prolog

```
1 % Implementação recursiva da função fatorial
2 fact(F, 0) :- F is 1.
3
4 fact(F, N) :-
5     N > 0,
6     NewN is N - 1,
7     fact(X, NewN),
8     F is X*N.
9
10
11 % Implementação recursiva de cauda
12 factTR(F, 0, Acc) :- F is Acc.
13
14 factTR(F, N, Acc) :-
15     N > 0,
16     NewN is N - 1,
17     NewAcc is Acc * N,
18     factTR(F, NewN, NewAcc).
19
20 factorial(F, N) :- factTR(F, N, 1).
```

Características da recursão em Prolog

- ▶ O escopo das variáveis de uma regra é a própria regra
- ▶ Cada nível da recursão tem seu próprio conjunto de variáveis
- ▶ A unificação entre o objetivo e a cabeça cláusula forçam as relações entre as variáveis de diferentes níveis
- ▶ Para garantir que os casos base sejam sempre testados, eles devem ser definidos antes da chamada recursiva
- ▶ Cuidado: na cláusula correspondente à chamada recursiva, é preciso garantir que os valores não correspondem ao casos bases
- ▶ Isto porque, devido ao fluxo de *backtracking*, o retorno à porta **redo** por meio de um ponto-e-vírgula pode fazer com que um valor que casasse com um dos casos-base também seja testado na chamada recursiva!
- ▶ Importante: a ordem dos predicados na chamada recursiva pode afetar a performance das consultas

Exemplo de estruturas recursivas

```
1 % Cada matrisoska pode conter outras de tamanho menor
2 matrisoska(a, b).
3 matrisoska(a, c).
4
5 matrisoska(b, d).
6 matrisoska(b, c).
7 matrisoska(b, f).
8
9 matrisoska(c, e).
10 matrisoska(c, g).
11
12 matrisoska(d, g).
13
14 matrisoska(e, f).
15
16 matrisoska(f, g).
17 matrisoska(f, h).
18
```

Exemplo de estruturas recursivas

```
19 % Lista todas as matriszkas contidas em M
20 list_all(M) :-
21     list(M, []).
22 list_all(_).
23
24 list(M, L) :-
25     nl,
26     tab(L),
27     print(->),
28     tab(1),
29     print(M),
30     matriszka(M, X),
31     NewL is L + 4,
32     list(X, NewL).
```

Estruturas de dados em Prolog

- ▶ Uma estrutura de dados combina tipos primitivos (átomos, inteiros, etc) e estruturas em tipos compostos
- ▶ A sintaxe de declaração de uma estrutura de dados é

```
functor(arg1, arg2, ..., argN).
```

- ▶ Cada argumento pode ser um tipo primitivo ou outra estrutura
- ▶ Sintaticamente, a declaração de uma estrutura é semelhante à declaração de um fato ou regra

```
car(peugeot, black, 2).  
car(honda, red, 4).
```

Consultas envolvendo estruturas de dados

- ▶ A ordem dos argumentos é importante nas consultas

```
car(X, red, _).
```

- ▶ Campos podem ser ignorados com a variável anônima
- ▶ Estruturas podem ser utilizadas em outras estruturas com o intuito de aumentar a legibilidade

```
car(honda, color(red), doors(4)).
```

- ▶ O predicado `not/1` recebe um objetivo como argumento e é bem sucedida quando o objetivo falha, e falha quando o objetivo é bem sucedido

Referências

1. **MERRIT**, Dennis. *Adventure in Prolog*, Amzi! Inc, 191 pgs, 2017.
2. **SHALOM**, Elad. *A Review of Programming Paradigms Throughout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
3. SWI Prolog. [SWI Prolog](#), acesso em 10/11/2020.
4. Wikipédia. [Prolog](#), acesso em 10/11/2020.