Programação Lógica

Regras, aritmética e estruturas de dados

Prof. Edson Alves

Faculdade UnB Gama

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 `capitais`/2, que lista as capitais X de uma região Y, pode ser definida por

```
capitais(X, Y) :- regiao(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*) da 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 podem 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 podem ser utilizados 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 sem corpo é sempre bem sucedido
- Neste caso, pode se usar uma variável especial, denominada variável anônima, representada pelo símbolo '_' (*underscore*)

Exemplo de definição regras e consultas envolvendo regras

```
male(homer).
male(bart).
female(marge).
female(lisa).
female(maggie).
father(homer, bart).
father(homer, lisa).
father(homer, maggie).
mother(marge, bart).
mother(marge, lisa).
mother(marge, maggie).
siblings(X, Y) :- X \= Y, father(F, X), father(F, Y).
brothers(X, Y) :- male(X), siblings(X, Y).
sisters(X, Y) :- female(X), siblings(X, Y).

% ?- sisters(X, bart).
% ?- siblings(X, maggie).  -- Explique o resultado desta consulta!
```

Características do Prolog

- A unificação é o processo de *pattern matching* do Prolog, utilizada na comunicação entre fatos e regras
- A execução é controlada pelo mecanismo de *backtracking* do Prolog
- `fail/0` pode ser usado para forçar o retorno do *backtracking* (via porta `fail`)
- Pode se forçar o sucesso de um predicado ou regra por meio de uma definição extra com variáveis anônimas e sem corpo
- O *backtracking* substitui os laços de outras linguagens
- O *pattern matching* substitui os testes condicionais e as estruturas de seleção
- As regras podem ser testadas individualmente, permitindo o desenvolvimento modular
- 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 arimética>.
```

- A variável `X` é atada ao 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

```
% Calcula as raizes reais do polinômio p(x) = ax^2 + bx + c
root_signal(S, Delta) :-
    Delta >= 0,
    S is 1.
root_signal(S, Delta) :-
    Delta > 0,
    S is -1.

roots(X, A, B, C) :-
    A \= 0,
    Delta is B ** 2 - 4*A*C,
    root_signal(S, Delta),
    X is (-B + S*sqrt(Delta))/(2*A).

% ?- roots(X, 1, -5, 6).
% ?- roots(X, 1, 0, 1).
% ?- 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 às variáveis globais
- Naturalmente, este recurso deve ser utilizado com parcimônia, 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` apresentada 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 desfará 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
- A chamada recursiva 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

```
% Implementação recursiva da função fatorial
fact(F, 0) :- F is 1.
fact(F, N) :-
    N > 0,
    NewN is N - 1,
    fact(X, NewN),
    F is X*N.

% Implementação recursiva de cauda
factTR(F, 0, Acc) :- F is Acc.
factTR(F, N, Acc) :-
    N > 0,
    NewN is N - 1,
    NewAcc is Acc * N,
    factTR(F, NewN, NewAcc).

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 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 casa 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

Estruturas de dados em Prolog

- Uma estrutura de dados combina termos 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

Exemplo de estruturas recursivas

```
% Cada matrioska pode conter outras de tamanho menor
matrioska(a, b).
matrioska(a, c).

matrioska(b, d).
matrioska(b, c).
matrioska(b, f).

matrioska(c, e).
matrioska(c, g).

matrioska(d, g).

matrioska(e, f).

matrioska(f, g).
matrioska(f, h).
```

Exemplo de estruturas recursivas

```
19 % Lista todas as matrioskas contidas em M a partir do nível L
20 list(M, L) :-
21     nl,
22     tab(L),
23     print(->),
24     tab(1),
25     print(M),
26     matrioska(M, X),
27     NewL is L + 4,
28     list(X, NewL).
```

Referências

1. **MERRIT**, Dennis. *Adventure in Prolog*, Amzi! Inc, 191 pgs, 2017.
2. **SHALOM**, Elad. *A Review of Programming Paradigms Througout 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.