Programação Lógica

Unificação e listas

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Processos de unificação

Elementos	Comportamento
variável e qualquer termo	a variável é unificada e atada com qualquer termo, inclusive outras variáveis
primitivo e primitivo	dois termos primitivos (átomos ou inteiros) só se unificam se ambos são idênticos
estrutura e estrutura	se unificam se tem mesmo construtor, mesma aridade e cada par de argumentos correspondentes se unifica

Igualdade

- O predicado pré-definido =/2 é bem sucedido se ambos argumentos se unificam e falha, caso contrário

```
?- a = a.
true.

?- a = b.
false.

?- p(a, b) = p(a, b).
true

?- p(a, b) = p(a, c).
false.

?- x(y, z(i, j)) = x(y, z(i, j)).
true.

?- x(y, z(i, j)) = x(y, z(j, i)).
false.
```

Variáveis e unificação

- Se uma variável se unificar com uma primitiva, ela assume seu valor

```
?- X = a.
X = a.

?- 2 = Y.
Y = 2.

?- f(x, y) = f(X, y).
X = x.

?- f(x, y) = f(X, z).
false.

?- f(x, y) = f(X, Y).
X = x.
Y = y.
```

Variáveis e unificação

- Variáveis também se unificam:

```
?- A = B.
A = B.

?- f(X, y) = f(Z, y).
X = Z.

?- X = Y, Y = teste.
X = Y, Y = teste.

?- X = Y, a(Z) = a(Y), X = teste.
X = Y, Y = Z, Z = teste.

?- X = Y, Y = 2, write(X).
2
X = Y, Y = 2.
```

Variáveis e unificação

- A variável anônima é um *wildcard* e não ata a valor algum
- Múltiplas instâncias de uma variável não implicam em valores iguais
- o predicado \=/2 unifica apenas se os valores são distintos

```
?- f(A, B) = C, write(C), nl, A = a, B = 8, write(C).
f(_6142,_6144)
f(a,8)
A = a,
B = 8,
C = f(a, 8).

?- f(b, X) = f(b, c(Y, d)), X = teste.
false.

?- f(X) = f(x, y).
false.

?- f(x, y, z) = f(X, X, z).
false.
```

Listas em Prolog

- Em Prolog, uma lista é uma coleção de termos
- Os termos podem ser qualquer tipo de dados do Prolog, incluindo estruturas e outras listas
- A sintaxe para declaração de uma lista é:

```
[term1, term2, ..., termN]
```

- A lista vazia é denotada por `[]` e é também denominada `nil`
- A unificação trata as listas de forma semelhante às estruturas de dados

```
?- asserta(f([a, b, c], d)).
true.

?- f(X, d).
X = [a, b, c].

?- [_, X, _] = [1, 2, 3].
X = 2.
```

Casamento de padrão em listas

- A notação `[X | Y]` ata `X` ao primeiro elemento da lista, chamado *head*, e ata `Y` a uma lista com todos os demais, denominados *tail*

```
?- [H|T] = [1, 2, 3, 4, 5].
H = 1,
T = [2, 3, 4, 5].

?- [H|T] = [1].
H = 1,
T = [].

?- [H|T] = [].
false.
```

- Podem ser listados vários elementos antes da barra, separados por vírgulas
- Contudo, após a barra deve haver um único elemento (uma lista)

```
?- [A, B, C | T] = [1, 2, 3, 4, 5].
A = 1,
B = 2,
C = 3,
T = [4, 5].
```

Listas e predicados

- Embora tenha uma sintaxe especial, uma lista é, de fato, um predicado de dois argumentos
- O primeiro argumento do predicado `./2` é o *head* da lista, e o segundo é o *tail*
- No SWI-Prolog, este operador foi substituído pelo operador `'[|]'`

```
?- L = '[|]'(1, '[|]'(2, '[|]'(3, []))).
L = [1, 2, 3].
```

- A estrutura interna da lista é adequada para rotinas recursivas

Teste de pertinência

- O predicado `member/2` determina se um termo é ou não membro da lista
- Ela pode implementada da seguinte maneira:

```
member(H, [H|T]).
member(X, [H|T]) := member(X, T).
```

- A primeira representa o caso base: o elemento pertence a lista se ele for o *head*
- Um fato com variáveis como argumentos funciona como uma regra
- A segunda é a chamada recursiva.
- Observe que o caso base é declarado antes da chamada recursiva
- O caso base da lista vazia já está incluso na segunda clásula: `member(X, [])` falha, pois `[]` e `[H|T]` não unificam

Concatenação de listas

- O predicado append/3 anexa o segundo argumento ao primeiro, guardando o resultado no terceiro:

```
?- append([1, 2], [3, 4, 5], X).
X = [1, 2, 3, 4, 5].
```

- Ele pode ser implementada da seguinte forma:

```
append([], X, X).
append([H|T1], X, [H, T2]):-
    append(T1, X, T2).
```

- append/3 também pode ser utilizado para decompor listas:

```
?- append(X, Y, [1, 2, 3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 2, 3],
Y = [] ;
false.
```

Listas e fatos

- É possível criar vários fatos a partir de uma única lista, chamando `assertz/1` no *head* da lista recursivamente:

```
list_to_facts([]).
list_to_facts([H|T]):-
    assertz(fact(H)),
    list_to_facts(T).
```

- O processo inverso é mais complicado (fatos para lista)
- Para isto, Prolog fornece o predicado `findall/3`: o primeiro argumento é o padrão para os termos da lista resultante, o segundo o objetivo e o terceiro a lista resultante
- Exemplo:

```
findall(X, fact(X), L).
L = [fact1, fact2, ..., factN].
```

Operadores e construtores

- Os operadores aritméticos em Prolog são construtores (*functors*)
- Por exemplo, o predicado adição `+/2` é um construtor:

```
?- display(2 + 2).
+(2,2)
true.

?- display(1*2 + 3).
+(*(1,2),3)
true.
```

- Qualquer construtor pode ser um operador, de modo que Prolog pode ler a estrutura de dados em um formato diferente
- Os operadores podem ser
  1. infixados (por exemplo, `3 + 4`)
  2. prefixados (por exemplo, `-7`)
  3. pós-fixados (por exemplo, `8 factorial`)

Ordem de precedência

- A cada operador é associado uma precedência, representada por um inteiro entre 1 e 1200
- Quanto maior a precedência, menor o número
- Operadores podem ser definidos por meio da diretiva op/3, cujos argumentos são a precedência, associatividade e o nome do operador
- A associatividade é definida por padrões (por exemplo, 'fxx', 'xfx' e 'xxf'), onde f indica a posição do operador
- Para declarar um operador em um arquivo fonte, a sintaxe é

```
:- op(precedencia, associatividade, nome).
```

pois op/3 é uma diretiva, não um predicado

Exemplo de declaração e uso de operadores

```
1  polygon(triangle, 3).
2  polygon(square, 4).
3  polygon(rectangle, 4).
4  polygon(hexagon, 6).
5
6  :- op(100, 'xf', has_four_sides).
7
8  has_four_sides(X) :-
9      polygon(X, 4).
10
11 % Exemplo de queries:
12 %
13 % ?- X has_four_sides.
14 % X = square ;
15 % X = rectangle.
16
17 % ?- triangle has_four_sides.
18 % false.
19
20 % ?- square has_four_sides.
21 % true .
```

Fatos e operadores

- É possível adicionar fatos com a notação de operadores
- A notação de operadores, em conjunto com a ordem de precedência dos operadores, podem levar a resultados inesperados
- Por outro lado eles permitem interfaces mais naturais para o usuário, dispensando o uso de vírgulas e parêntesis
- No caso de dois operadores de mesma precedência, a ordem de associatividade fica a cargo do interpretador Prolog

```
:- op(100, 'xfx', f).

x f y.

?- a f b.
false.

?- x f y.
true.
```

Associatividade de operadores

- O caractere `'y'` no parâmetro associatividade de `op/3` descreve a associatividade do operador
- Se o operador é não-associativo (ausência de `'y'`) ou a direção da associatividade (posição do `'y'`):
  1. infixo: `'xfx'` (não associativo), `'xfy'` (R to L), `'yfx'` (L to R)
  2. pré-fixa: `'fx'` (não associativo), `'fy'` à (L to R)
  3. pós-fixada: `'xf'` (não associativo), `'yf'` (R to L)
- Os parêntesis podem ser utilizados para modificar a ordem de precedência ou a associatividade

Exemplo de associatividade em Prolog

```
1 % Exemplos de operadores e associatividade em Prolog
2 :- op(100, 'xfx', f).
3 :- op(100, 'xfy', g).
4 :- op(100, 'yfx', h).
5
6 %% Exemplos de consultas
7 %
8 % ?- a f b f c = f(a, f(b, c).
9 % ERROR: Syntax error: Operator expected
10 % ERROR: a f b f c = f(a, f(b, c)
11 % ERROR: ** here **
12 % ERROR:  .
13 %
14 % ?- a g b g c = g(a, g(b, c)).
15 % true.
16 %
17 % ?- a h b h c = h(h(a, b), c).
18 % true.
```

Predicado `is`

- O predicado pré-definido `is` computa a expressão dada

```
?- X is 2 + 3.
X = 5.

?- X = 2 + 3.
X = 2+3.

?- 5 = 2 + 3.
false.
```

- Observe que o predicado `=/2` unifica os resultados sem computá-los

Operador pescoço

- Em Prolog, as cláusulas são estruturas de dados escritas com sintaxe de operadores
- O construtor pescoço (*neck*) `:-` é um operador infixado de dois argumentos

```
:-(Head, Body).
```

- O corpo é uma estrutura de dados com o construtor and (vírgula):

```
,(objetivo1, ,(objetivo2, ..., ,(objetivoN))).
```

- Note a ambiguidade semântica do operador vírgula na expressão acima: ele tanto é o separador dos argumentos quando o "e" lógico

Predicato cut

- O predicado *cut* (!, ponto de exclamação) permite a interrupção o processo de *backtracking*
- Ele congela todas as decisões tomadas pelo *backtracking* até o momento
- O principal motivo para o uso deste operador é o aumento de performance, pois habilita a poda no *backtracking*
- Ele é motivo de controvérsia entre os puristas do paradigma lógico e os pragmáticos
- Ele é considerado o goto do paradigma

Negação

- O predicado not/1 é implementado em termos do operador *cut* e do predicado call/1, o qual invoca um predicado:

```
not(X) :- call(X), !, fail.
not(X).
```

- Observe que se a chamada do predicado X for bem sucedida, ela seguirá para a direita, via porta exit
- Daí a execução encontrará o *cut*, interrompendo o *backtracking*
- Por fim o predicado fail/0 é encontrado, determinando que a consulta retorne falso
- Por outro lado, caso a chamada de X falhe, será avaliada a segunda cláusula, a qual sempre é verdadeira
- Assim, o predicado not/1 inverte o resultado da consulta X

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.