Fundamentos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Definições

Uma **cláusula** é um fato ou uma regra.

Um **predicado** é a coleção de cláusulas com o mesmo nome e aridade.

Termos

Termos

Um programa em Prolog é construído com termos.

Um **termo** é um dos valores

- Constantes
  - Átomos
  - Números
- Variáveis
- Estruturas

Cada termo é definido com uma sequência de caracteres

- Letras maiúsculas: `A .. Z`
- Letras minúsculas: `a .. z`
- Dígitos: `0 .. 9`
- Símbolos: `+ - * / \ ~ ^ < > : . ? @ # $`

Constante

Constantes nomeiam objetos ou relações específicas

- Átomos começam com letra minúscula ou símbolo, ou entre apóstrofos '

```
casa
-->
'Jose da Silva'
'123'
```

- Números

```
89
-17
2.67e10
```

Parecem átomos mas começam com letras maiúsculas ou _

```
X
Reposta
Nome_longo
```

Variáveis anônimas são definidas com o caractere _.

Cada ocorrência de uma variável anônima refere-se a um valor (que pode ser diferente das ocorrências anteriores).

Usamos variáveis anônimas quando não estamos interessados no valor

```
?- gosta(joao, _). % existe alguém que joao gosta?
true.
```

Uma **estrutura** é um único objeto composto de outros objetos chamados de argumentos (ou componentes)

- Estruturas também são chamadas de termos compostos.
- Semelhante a registros em linguagens imperativas, mas os componentes não são nomeados.

Fato: João possui o livro Algoritmos escrito pelo Cormem

```
possui(joao, livro(algoritmos, cormem)).
```

O nome da estrutura é chamado de **functor**, no exemplo o functor é `livro`.

`algoritmos` e `cormem` são os **componentes** da estrutura `livro(algoritmos, cormem)`.

A quantidade de componentes em uma estrutura é a **aridade** da estrutura.

Operadores

Operadores

As vezes é conveniente escrever functors como operadores.

- `x + y` ao invés de `+(x, y)`
- Observe que os dois exemplos descrevem o mesmo objeto, a estrutura com o functor `+` e os componentes `x` e `y`

O Prolog usa regras de precedência e associatividade semelhantes a de outras linguagens

- `2 + 4 * 3` é o mesmo que `+(2, *(4, 3))`
- `8 / 2 / 2` é o mesmo que `/(/(8,2), 2)`

Lembre-se: estas construções descrevem estruturas, elas só são interpretadas (e avaliadas) como expressões aritméticas em alguns contexto (veremos a seguir)

```
?- X = 2 + 4 * 3, write_canonical(X).
+(2,*(4,3))
X = 2+4*3.
```

O símbolo `=` é um operador

```
?- X = (casa = 2).
X =  (casa=2).
```

`casa = 2` é o mesmo que `=(casa, 2)`

`X = (casa = 2)` é o mesmo que `=(X, =(casa, 2))`

Ambas as construções `casa = 2` e `X = (casa = 2)` são termos, mas uma diferença é que o termo `X = (casa = 2)` está sendo interpretado como código (consulta), enquanto o termo `casa = 2` como um dado.

Unificação

Ideia: dois termos unificam se eles são os mesmos termos ou se eles contêm variáveis que podem ser instanciadas de maneira que os termos resultantes sejam iguais.

O operador = é usado para especificar unificação.

Duas constantes unificam se eles são iguais.

Uma variável não instanciada unifica com qualquer outro termo. No caso de duas variáveis não instanciadas é criado uma co-referência, neste caso, quando uma das variáveis é instanciada, a outra também é.

Duas estruturas unificam se

- Elas têm o mesmo functor e a mesma aridade
- Todos os argumentos correspondentes unificam (observe que a definição é recursiva)
- A instanciação das variáveis são compatíveis

```
?- casa = casa.
true.

?- casa = carro.
false.

?- X = Y, gosta(joao, Y) = gosta(joao, pizza).
X = Y, Y = pizza.

?- livro(X, algoritmos) = livro(autor(thomas, cormem), Y).
X = autor(thomas, cormem),
Y = algoritmos.
```

Além do paradigma lógico

Além do paradigma lógico

Nós podemos escrever muitos programas utilizando apenas as construções que vimos até agora.

No entanto, trabalhar com números de forma eficiente requer construções “além do paradigma lógico”.

Primeiro veremos a forma tradicional de trabalhar com número em Prolog.

Depois veremos a forma moderna usando restrições.

Aritmética

Termos que representam expressões aritméticas são avaliados quando usados com os predicados `=:=`, `=\=`, `>`, `>=`, `<`, `=<`

```
?- 3 + 4 =:= 10 - 3.      % igual
true.
?- 4 * 3 =\= 4 + 4 + 4.   % diferente
false.
?- X = 3, Y = 5, X + Y > 2 * X.
X = 3,
Y = 5.
?- X = 3, Y = 5, X + Y < 2 * X.
false.
?- X > 2.
ERROR: >/2: Arguments are not sufficiently instantiated
```

Observe que os termos não podem ter variáveis não instanciadas.

O operador `is` pode ser usado para instanciar uma variável com o resultado de uma expressão aritmética.

O termo da direita é interpretado como um expressão aritmética e o resultado da avaliação da expressão é unificado com o termo da esquerda

```
?- X is 3 + 4 * 2.
X = 11.

?- 2 + 2 is 2 + 2.
false.

?- 2 is X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Antes de fazermos alguns exemplos, vamos discutir a forma que vamos projetar predicados.

Vamos seguir um processo similar ao que usamos para escrever funções em Racket

- Especificação
  - Nome do predicado e seus argumentos (com o modo dos argumentos e o determinismo)
  - Propósito do predicado
  - Exemplos
- Implementação
- Verificação
- Revisão

Propósito do predicado

De acordo com PLdoc.

Um predicado pode ser

- `det` (determinístico) satisfeito uma vez sem escolha
- `semidet` (semi determinístico) falha ou é satisfeito uma vez sem escolha
- `nondet` (não determinístico) sem limite de vezes que o predicado é satisfeito e pode deixar escolha na última vez que é satisfeito

Modo dos argumentos (descrição simplificada)

- `+` argumento precisa estar completamente instanciado
- `-` argumento não pode estar instanciado
- `?` argumento pode ou não estar instanciado

Exemplos

Usaremos a biblioteca plunit.

Quadrado

Defina um predicado `quadrado(X, Y)` que é verdadeiro se $Y = X^2$.

Quadrado

```
:- use_module(library(plunit)).

%% quadrado(+X, ?Y) is semidet
%
%  Verdadeiro se Y é o quadrado de X.

:- begin_tests(quadrado).

test(quadrado4) :- quadrado(4, 16).
test(quadrado4, fail) :- quadrado(4, 20).
test(quadrado3, Q == 9) :- quadrado(3, Q).

:- end_tests(quadrado).
```

```
quadrado(X, Y) :-
    Y is X * X.
```

Para executar os utilize o predicado `run_tests`

```
?- run_tests.
% PL-Unit: quadrado .. done
% All 3 tests passed
true.
```

Fatorial

Defina um predicado `fatorial(N, F)` que é verdadeiro se o fatorial de `N` é `F`.

Fatorial

```
:- use_module(library(plunit)).

%% fat(+N, ?F) is semidet
%
%  Verdeiro se F é o fatorial de N.

:- begin_tests(fatorial).

test(f0) :- fat(0, 1).
test(f1) :- fat(1, 1).
test(f2) :- fat(2, 2).
test(f3) :- fat(3, 6).
test(f4, fail) :- fat(4, 22).
test(f5, F == 120) :- fat(5, F).

:- end_tests(fatorial).
```

Fatorial

```
fat(N, F) :-
    N >= 1,
    N0 is N - 1,
    fat(N0, F0),
    F is N * F0.

fat(0, 1).
```

Considerações

Nós vimos que a implementação desses predicados não foram “suaves”.

Além disso, os predicados restringem o modo de alguns parâmetros para instanciados.

Veremos outra forma de implementar esses predicados.

No paradigma de programa por restrições o usuário específica as restrições que as solução de uma problema deve atender e o ambiente de execução tentar encontrar a solução que atenda as restrições.

Nós podemos combinar programação por restrições com programação lógica

- Isto vai nos permitir escrever programas mais simples e mais genéricos ao mesmo tempo que facilitará raciocinar sobre os programas
- Não veremos como o ambiente procura por soluções que atendam as restrições... (vai ser como uma caixa preta para nós)

Já vimos dois predicado que criam restrições: a unificação e o `dif`.

```
?- dif(7, 4 + 3).
true.
?- dif(7, A).
dif(A, 7).
?- dif(A, A).
false.
?- dif(A, B).
dif(A, B).
?- dif(A, B), A = nada.
A = nada,
dif(nada, B).
?- dif(A, B), A = nada, B = nada.
false.
```

Precisamos incluir a biblioteca CLP(FD) – Constraint Logic Programming over Finite Domains

```
:- use_module(library(clpfd))
```

Restrições básicas

- `#=`
- `#\=`
- `#<`
- `#=<`
- `#>`
- `#>=`

Veja a lista completa na documentação.

Outros predicados

- `in`
- `ins`
- `all_distinct`
- `label`

Restrições sobre inteiros

```
?- X #> 3, X #\= 10.
X in 4..9\/11..sup.

?- X + 1 #= 3 * 2.
X = 5.

?- 168 + X #= Y * Y, 0 #= X -1 .
X = 1,
Y in -13\/13
```

```
?- X in 1..3, Y in 1..3, all_distinct([X, Y]).
X in 1..3,
all_distinct([X, Y]),
Y in 1..3.
```

```
?- X in 1..3, Y in 1..3, all_distinct([X, Y]), label([X, Y]).
X = 1,
Y = 2 ;
X = 1,
Y = 3 ;
X = 2,
Y = 1 ;
X = 2,
Y = 3 ;
X = 3,
Y = 1 ;
X = 3,
Y = 2.
```

Exercício

Reescreva os predicados `quadrado` e `fatorial` usando restrições sobre inteiros.

Exercício

Projete um predicado que é verdadeiro se os seus argumentos formam a solução para um mini sudoku. Faça uma versão usando apenas o predicado `dif` e outra versão usando restrições de inteiros.

```
?- mini_sudoku(A, 2, C, 3, E, F, G, 2, 2, J, K, L, 1, N, 2, P).
A = G, G = J, J = P, P = 4,
C = F, F = L, L = 1,
E = K, K = N, N = 3 ;
false.
```

Referências

Referências básicas

- Seção Data Structures e Integer Arithmetic do livro The Power Of Prolog
- Capítulos 2 e 3 da apostila Paradigmas de programação - Prolog
- Capítulos 1 e 2 do livro Programming in Prolog
- Capítulos 1 , 2 e 5 do livro Learn Prolog Now.

Referências complementares

- Introduction to Prolog