Programação Declarativa

Paradigmas de Programação Avançados

Salvador Abreu, Universidade de Évora, 2019/20

Técnicas de Programação

Aritmética

Manipulação de termos

Strings

Input-output

Base de dados

Meta-interpretadores

Aritmética

Fazer contas em Prolog

No Prolog, os predicados de sistema para aritmética permitem aceder à aritmética nativa do computador.

O predicado **is/2** (i.e. is(VALOR, EXPRESSAO)):

- É utilizado para avaliação de expressões aritméticas.
- Normalmente escrito na notação infixa, i.e. VALOR is EXPRESSAO.
- Suporta muitas funções matemáticas (sin/1, …) e algumas especiais.
- Consultar a documentação de cada sistema Prolog para referência.

Existem outros operadores (>, <, ...) que avaliam os seus argumentos como expressão aritmética. Por exemplo, na query (A < B), A e B são primeiro avaliados como expressões aritméticas e só depois são comparados os resultados.

Fazer contas em Prolog

Um goal da forma

VALOR is EXPR

é interpretado do seguinte modo: a expressão aritmética **EXPR** é avaliada e o resultado é unificado com **VALOR**.

- (X is 3+5) tem a solução X=8.
- A query (8 is 3+5) sucede
- a query (3+5) is (3+5) falha na unificação
- a query (X is 3+a) tem um erro pois 3+a não pode ser avaliado.
- a query (X is 3+Y) pode ter um erro.

O predicado **is/2** **não é a afetação** das linguagens procedimentais: (N is N+1) falha sempre.

Exemplo: maior divisor comum

Exemplo dum predicado que calcula o maior divisor comum:

```
mdc(I, 0, I).
mdc(I, J, MDC) :-
    J > 0,
    R is I mod J,
    mdc(J, R, MDC).
```

Outra implementação:

```
mdc(N, N, N).
mdc(A, B, C) :- A>B, X is A-B, mdc(X, B, C).
mdc(A, B, C) :- A<B, X is B-A, mdc(A, X, C).
```

Exemplo: factorial

Para começar, o mau exemplo:

```
fact(0, 1).
fact(N, F) :-
    N > 0,
    N1 is N-1,
    fact(N1, F1),
    F is N*F1.
```

Uma melhor implementação:

```
fact(N, F) :- fact(N, 1, F).

fact(0, F, F).
fact(N, T, F) :-
    N > 0,
    T1 is T*N,
    N1 is N-1,
    fact(N1, T1, F).
```

Exemplo: somar uma lista

Para começar, o mau exemplo:

```
soma([], 0).
soma([I|Is], S) :-
    soma(Is, SI),
    S is I + SI.
```

Uma melhor maneira:

```
soma(Is, S) :- soma(Is, 0, S).

soma([], S, S).
soma([I|Is], T, S) :-
    T1 is T+I,
    soma(Is, T1, S).
```

Manipulação de Termos

Tipo dum termo

O goal:	Sucede se TERM for...
`var(TERM), nonvar(TERM)`	... respetivamente variável ou não
`atom(TERM)`	... um átomo
`integer(TERM), float(TERM), number(TERM)`	... respetivamente um inteiro, um real ou um dos dois
`atomic(TERM)`	... um átomo ou número
`compound(TERM)`	... um termo composto
`callable(TERM)`	... um átomo ou termo composto
`ground(TERM)`	… um termo sem variáveis livres
`list(TERM), partial_list(TERM)`	... uma lista ou lista inacabada

Pausa 2019.10.08

Análise e construção de termos

O goal:	Sucede se...
`functor(TERM, FUNC, ARITY)`	O funtor principal de TERM unificar com FUNC e tiver aridade ARITY
`arg(POS, TERM, ARG)`	O subtermo na posição POS de TERM unificar com ARG
`TERM =.. LTERM`	LTERM for uma lista cuja cabeça é o funtor principal de TERM e cuja cauda é a lista dos subtermos de TERM **=..** pronuncia-se “univ”

Exemplos: **functor/3**

```
| ?- functor(nome(a,b), F, A).

A = 2
F = nome

yes
| ?- functor(T, nome, 2).

T = nome(_,_)

yes
| ?- functor(T, F, 2).
uncaught exception: error(instantiation_error,functor/3)
| ?-
```

Exemplos: **arg/3**

```
| ?- arg(2, nome(a,b), X).

X = b

yes
| ?- functor(T,nome,2), arg(1,T,a), arg(2,T,b).

T = nome(a,b)

yes
| ?-
```

Exemplos: **=../2**

```
| ?- nome(a,b) =.. X.

X = [nome,a,b]

| ?- X =.. [pai, manel, quim].

X = pai(manel,quim)

| ?- length(L, 2), X =.. [nome | L].

L = [A,B]
X = nome(A,B)

| ?-
```

Relação entre **=../2** e os outros

Podemos definir =.. em função dos outro predicados built-in; por exemplo:

```
univ(A, [A]) :- atomic(A), !. % clausula dispensável
univ(T, [F|As]) :-
    functor(T, F, A),
    univ(0, A, T, As).

univ(A, A, _, []).
univ(I, A, T, [ST|STs]) :-
    I < A, I1 is I+1,
    arg(I1, T, ST),
    univ(I1, A, T, STs).
```

univ/2 comporta-se como =../2.

Exemplo: aplanar uma lista (*flatten*)

Ideia: fazer uma lista com os elementos duma lista de listas.

```
flatten([X|Xs], Ys) :-
    flatten(X, Y1), flatten(Xs, Y2), append(Y1, Y2, Ys).
flatten(X, [X]) :- atomic(X), X \== [].  % não lista
flatten([], []).
```

Uma melhor forma de fazer:

```
flatten(Xs, Ys) :- flatten(Xs, [], Ys).

flatten([], [X|S], Ys) :- flatten (X, S, Ys).
flatten([X|Xs], S, Ys) :-
    list(X),
    flatten(X, [Xs|S], Ys).
flatten([X|Xs], S, [X|Ys]) :-
    atomic(X),
    X \== [],
    flatten(Xs, S, Ys).
```

Strings

Strings e Átomos

Um átomo é, por definição, indivisível. I.e. constitui um elemento de base da linguagem.

O Prolog não inclui um tipo de dados “char” nem “string”.

Tem uma notação especial para listas de códigos de caracteres, que - abusivamente - designamos por “string”.

```
| ?- X=ola.

X = ola

yes
| ?- X="ola".

X = [111,108,97]

yes
| ?-
```

Strings e Átomos

Predicado name/2 para converter entre átomos e os strings correspondentes.

- name(ATOM, STRING)

  Sucede se STRING for a representação textual do átomo ATOM. Por exemplo:

```
| ?- name(ola, X).
X = [111,108,97]
```

  Pelo menos um dos argumentos tem de ser ground. Pode ser usado para analisar ou construir átomos, por exemplo:

```
| ?- X="xpto", name(Y, X).
X = [120,112,116,111]
Y = xpto
```

Manipulação de strings

Em Prolog, sendo que as strings são representadas como listas de inteiros, as manipulações destas são feitas com os predicados sobre listas.

Exemplos, o **append/3**, **sublist/2**, **prefix/2**, assim como todas as operações sobre listas com base na unificação.

Podem-se construir predicados auxiliares para ajudar a ler o código, por exemplo:

```
| ?- name(a,A), name(z,Z).
A = [97]
Z = [122]
```

Pode-nos levar a definir um predicado de “letra minúscula” (para strings), por exemplo assim:

```
minusc(L) :- 97 =< L, L =< 122.
```

Input-Output

Input-output de termos e outros

Basico:

- **`write/1, writeq/1`**

  Escreve no output atual o termo que é o seu argumento, sem “plicas”. O predicado **`writeq/1`** faz igual mas de forma a poder ser lido de volta com **`read/1`**.

- **`read/1`**

  Lê um termo (terminado por “.”). Afetado pelas definições de operadores. Quando chega ao fim do stream (fim de ficheiro, control-D, …) é como se tivesse lido o átomo **`end_of_file`**.

- **`see/1, seeing/1, seen/0`**
- **`tell/1, telling/1, told/0`**

  Aponta, interroga e conclui o I/O para um ficheiro.

I/O de carateres

- **`get/1, get0/1`**

  Lê um carater do input atual. No caso de **`get/1`**, passa por cima de todos os espaços em branco (inclui newlines). Retorna o código numérico lido.

  No end-of-file retorna a constante **`-1`**.

- **`put/1`**

  Escreve o símbolo correspondente ao código numérico dado no output atual.

- **`nl/0`**

  Escreve um fim-de-linha no output atual. Em sistemas unix é como se fizesse **`put(10)`**. Em sistemas windows poderá ser diferente (p/ex CR-LF).

I/O formatado

- **`format(FORMAT, LIST)`**

  Escreve o string em **`FORMAT`**, interpretando os strings da forma "**~F**" como elementos formatadores dos termos na lista **`LIST`**.

  Alguns exemplos mais comuns de formato (**~F**), aplicável ao elemento **`X`** da lista:

~w	**`write(X)`**
~q	**`writeq(X)`**
~d	**`write(X)`**, verificando que **`X`** seja um número inteiro
~s	Escreve o **`X`** como um string (assume que é uma lista de inteiros/códigos)
~n	Escreve um newline

Bases de Dados

Predicados como **dados**

Um predicado “factual” pode ser usado para exprimir uma relação simples.

Por exemplo

```
nome(evora, ‘Eborae’).
nome(beja, ‘Pax Julia’).
```

Podemos aceder a esta relação invocando o predicado:

```
| ?- nome(beja, X).
X = ‘Pax Julia’

| ?- nome(lisboa, X).
No
```

Acrescentar dados

Podemos usar os predicados built-in de *asserção*, que permitem adicionar factos a um predicado utilizador.

São eles:

```
asserta(CLAUSE).
assertz(CLAUSE).
```

Em que CLAUSE é uma clausula, como no texto dum programa.

No caso de **`assertz/1`**, é como se tivessemos acrescentado o termo CLAUSE ao predicado respetivo, no final, e no de **asserta/1**, no início.

Formato duma clausula para os asserts

O argumento CLAUSE de asserta/1 e assertz/1 é entendido assim

- Se for **CLAUSE = (HEAD :- BODY)**, então acrescentamos essa clausula diretamente ao predicado **P/A** em que **functor(HEAD, P, A)**
- Se CLAUSE = HEAD, então agimos como se fosse CLAUSE = (HEAD :- true)

Por exemplo

```
| ?- assertz(nome(lisboa, 'Olissipo')),
     asserta(nome(roma, 'Roma')).
```

Resultaria na base de dados:

```
nome(roma, 'Roma').
nome(evora, 'Eborae').
nome(beja, 'Pax Julia').
nome(lisboa, 'Olissipo').
```

Remover clausulas

De igual modo, podemos remover clausulas com o built-in retract/1.

```
retract(CLAUSE)
```

Em que **CLAUSE** tem a mesma estrutura que para os asserts que já vimos.

O efeito é remover a primeira clausula que **unificar** com **CLAUSE**. Este goal é backtrackável, i.e. podemos remover todas as clausulas dum predicado P/A assim:

```
kill(P/A) :-
    functor(HEAD, P, A),
    retract((HEAD :- _)),
    fail.
kill(_).
```

Ciclo geração-fail típico

Mais remoção

Existe o built-in **`retractall(HEAD)`** que faz o óbvio.

Por exemplo, **`retractall(nome(_, _))`** remove todas as clausulas para nome/2.

Da mesma forma, podemos usar o predicado built-in **`abolish(P/A)`**, por exemplo **`abolish(nome/2)`**.

Cuidado: o **abolish/2** destroi todo o rasto da sua “vítima”, incluindo a eventual informação de “dinâmico”.

Mas...

Qualquer predicado que queiramos manipular com os asserts ou retracts tem de ser declarado dinâmico, no ficheiro source onde é introduzido (e antes de qualquer cláusula…)

Faz-se com a diretiva dynamic:

```
:- dynamic(F/A).
```

Por exemplo, teriamos

```
:- dynamic(nome/2).
nome(evora, 'Eborae').
nome(beja, 'Pax Julia').
```

Para poder usar os **assert**, **retract**, etc.

Aceder a uma clausula

Podemos aceder a uma clausula dum predicado (***declarado dinâmico***) com o predicado **`clause/2`**:

```
clause(HEAD, GOAL)
```

Sucede se houver uma clausula para **HEAD** cujo corpo unifique com **GOAL**.

Exemplo de clause/2

Supondo que temos:

```
:- dynamic(nome/2).
nome(a, 'A').
nome(b, 'B').
```

Podemos fazer:

```
| ?- H=nome(_,_), clause(H,B).

B = true
H = nome(a,'A') ? ;

B = true
H = nome(b,'B')

| ?-
```

Meta Interpretadores

Interpretadores

Interpretador de autómato finito não-determinístico (NFA)

```
aceita(S) :- inicial(Q), aceita(Q, S).

aceita(F, []) :- final(F).
aceita(S, [X|Xs]) :- trans(S, X, NS), aceita(NS, Xs).
```

Parecido com o predicado de cálculo de caminho.

Não lida bem com ciclos...

Meta-interpretadores

O interpretador mais simples faz… **batota**! :)

```
interp(G) :- G.
```

Porque usa o meta-call do Prolog.

Meta-interpretadores

Uma coisa mais “séria”: vamos explorar a estrutura das clausulas do programa e, para demonstrar um goal G, para o qual exista uma clausula, vamos resolver o corpo da mesma:

```
interp(true).
interp((A, B)) :- interp(A), interp(B).
interp(G) :- clause(G, B), interp(B).
```

Este interpretador serve para Prolog “puro”.

Interpretador de linha de comando

Podemos fazer um processo “iterativo” assim:

1. Ler um termo (o nosso “goal”)
2. Se for o fim do stream (end_of_file), consideramos que acabou
3. Se for um termo ground (i.e. sem variáveis), vemos se é verdade e escrevemos o resultado (“sim ou não”)
4. Senão, é um termo com variáveis e portanto é possível que suceda com valores diferentes das mesmas: vamos ver se se consegue resolver e, para cada solução, escrevemos o termo tal como ficou depois da mesma. Após isso, **falhamos** para obter a próxima solução.

Nos passos 3 e 4 também voltamos ao início (passo 1).

Interpretador de “linha de comando”

```
tl :- write('> '), read(G), tl(G).

tl(end_of_file) :- !.
tl(G) :- ground(G), !, tl_ground(G), tl.
tl(G) :- tl_solve(G), tl.

tl_solve(G) :- G, write(G), nl, fail.
tl_solve(_) :- write(acabou), nl.

tl_ground(G) :- G, !, write(sim), nl.
tl_ground(_) :- write(nao), nl.
```

```
ground(V) :- var(V), !, fail.
ground(A) :- atomic(A), !.
ground([T|Ts]) :- !, ground(T), ground(Ts).
ground(T) :- T =.. TT, ground(TT).
```

Top-level

Exemplo de execução

```
| ?- tl.
> true.
sim
> X=1.
1=1
acabou
> member(X,[a,b,c]).
member(a,[a,b,c])
member(b,[a,b,c])
member(c,[a,b,c])
acabou
> 2<1.
nao
>
(4 ms) yes
| ?-
```

Aqui foi um EOF (**Ctl-D**)