Meta-programovanie v Prologu

Meta-programovanie

- Je to technika programovania, ktorá umožňuje manipuláciu s programovými štruktúrami
- Vhodná pre všetky jazyky, ktoré používajú rovnaké štruktúry pre program a pre údaje
  - Lisp, Prolog a podobne
- V iných jazykoch to môžeme do istej miery nahradiť špeciálnymi vzormi, napríklad „Abstract factory“

Špeciálne predikáty v Prologu

- `call(<predikát>)`
  - Ako argument dostane predikát, ktorý sa snaží splniť
  - Tento predikát nijako nemodifikuje
- `not(<predikát>)`
  - Ako predchádzajúci, len vráti negáciu výsledku volaného predikátu
- Potrebovali by sme predikát aj nejako modifikovať

Predikát (operátor) „=..“

- `<Predikát> =.. [<funktor>|[<argumenty>]]`
- `?- P =.. [min,2,3,4,5].`
- `P = min(2, 3, 4, 5)`
- Pre korektnú činnosť potrebuje mať definovanú aspoň jednu stranu
- Môžeme teda transformovať zoznam na predikát alebo predikát na zoznam
- Hotový predikát je možné vykonať pomocou call

Príklad mapc

```
mapc(FunctionName,[H|T],[NH|NT]):-
    Function=..[FunctionName,H,NH],
    call(Function),
    mapc(FunctionName,T,NT).
mapc(_,[],[]).
inv(A,B):- B is -A.

?- mapc(inv,[2,3,-5],L).
L = [-2,-3,5]
true
```

Príklad filter

```
filter(F,[H|T],[H|T1]):-
    P =..[F,H],
    call(P),
    filter(F,T,T1).
filter(F,[_|T],T1):- filter(F,T,T1).
filter(_,[],[]).

raz(_).


?- filter(raz,[1,2,3],L),writeln(L),fail;true.
```

Výstup

```
?- filter(raz,[1,2,3],L),writeln(L),fail;true.

[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
true.
```

Predikát clause/2

- `clause(H,T)`
  - vstavaný predikát, ktorý pre H ako hlavičku klauzuly vráti v T telo klauzuly.
- Môže sa použiť len na nami definované predikáty

```
?- clause(inv(N,M),T).
T = (N is -M).
```

Meta-interpret

- Elementárny:

```
solve(Goal):-call(Goal).
```

- Náročnejší:

```
solve((A,B)):-
    !,solve(A),solve(B).
solve(A):-
    predicate_property(A,built_in),!,
    call(A).
solve(A):-
    clause(A,B),solve(B).
```

Zmena poradia volania cieľov

```
solve((A,B)):-
    !,solve(B),
    solve(A).
solve(Goal):- call(Goal).


?- solve((write(jeden),write(dva),write(tri))).
tridvajeden
true
```

Rozšírenie interpreta

```
solve((A,B)):- !,
     solve(A),
     solve(B).
solve(A):-
     predicate_property(A,built_in),!,
     call(A).
solve(A):-
     clause(A,B),
     demon_in(A),
     solve(B),
     demon_out(A).
```

Démon na jednoduché trasovanie

```
demon_in(A):-
       write('Volanie: '),
       writeln(A).
demon_in(A):-
       write('Neuspech: '),
       writeln(A),
       fail.

demon_out(A):-
       write('Uspesne ukoncenie: '),
       writeln(A).
demon_out(A):-
       write('Opakovane volanie: '),
       writeln(A),
       fail.
```

Jednoduché trasovanie

```
member(H,[H|_]).
member(H,[_|T]):-member(H,T).

?- solve(member(a,[a,b])).
Volanie: member(a,[a,b])
Uspesne ukoncenie: member(a,[a,b])
true ;
Opakovane volanie: member(a,[a,b])
Neuspech: member(a,[a,b])
Volanie: member(a,[a,b])
Volanie: member(a,[b])
Neuspech: member(a,[b])
Neuspech: member(a,[a,b])
false.
```

Trasovanie s ďalšou informáciou

```
infopoint(_).
member(H,[H|_]).
member(H,[_|T]):-infopoint(T),member(H,T).

?- solve(member(a,[a,b])).
Volanie: member(a,[a,b])
Uspesne ukoncenie: member(a,[a,b])
true ;
Opakovane volanie: member(a,[a,b])
Neuspech: member(a,[a,b])
Volanie: member(a,[a,b])
Volanie: infopoint([b])
Uspesne ukoncenie: infopoint([b])
Volanie: member(a,[b])
Volanie: infopoint([])
Uspesne ukoncenie: infopoint([])
Opakovane volanie: infopoint([])
Neuspech: infopoint([])
Neuspech: member(a,[b])
Opakovane volanie: infopoint([b])
Neuspech: infopoint([b])
Neuspech: member(a,[a,b])
false.
```

Ďalšie možnosti

- Logovanie
- Definovanie a interpretovanie vlastného jazyka (procedurálne až znalostné)
- Automatické dôkazy správnosti a/alebo vypočítateľnosti v iných jazykoch (pascal)
- Agentový prístup – rôzne databázy pre rovnakú hlavičku predikátu (iné informácie)
- Aspektový prístup