LAB: Metaprogramowanie w Prologu

1 Temat: Sprawdzanie typów termów

Prolog dostarcza szeregu predykatów, pozwalających na analizowanie typów termów.

Po pierwsze można stwierdzić, czy term jest niewiadomą (zmienną logiczną) o nieustalonej wartości, czy też ma określoną wartość, czyli jest stałą, lub zmienną o wcześniej ustalonej wartości (po pomyślnej unifikacji).

Służą do tego predykaty:

```
var(X).
nonvar(X).
```

Poza tym, jest szereg predykatów sprawdzających, czy term jest:

```
atom
```

atomem logicznym (stałą, napisem)

```
atomic
```

liczbą lub atomem

```
number
```

liczbą

```
compound
```

złożoną strukturą

```
integer
```

liczbą całkowitą

```
float
```

liczbą zmiennoprzecinkową

Ćwiczenie:

Proszę przetestować poniższe:

```
?- var(X).
?- var(X),X=2.
?- X=2,var(X).

?- atom(X).
?- atom(3).
?- atom(a).
?- atom(+).
?- atom(:=).
?- atom('ala').

?- atomic(a).
?- atomic(3).
?- atomic(+).
?- atomic(X).

?- number(ala).
?- number(3).

?- integer(3).
?- integer(3.14).
?- float(3).
?- float(3.14).

?- compound(ala).
?- compound(ala(ma,kota)).
?- compound(3).
```

Uwaga: `compound` nie nadaje się do „wykrywania” list, bo:

```
?-  compound([]).
```

2 Temat: Konstruowanie i dekompozycja termów

Z racji tego, iż termy sa podstawową metodą strukturalizacji danych w Prolog, istnieje kilka mechanizmów wspomagających ich przetwarzanie:

```
=..
```

operator pozwala na dynamiczną zamianę termu na listę i vice versa

```
functor(T,N,A)
```

predykat jest prawdziwy, jeżeli N pokrywa się z nazwą termu T o arności A

```
arg(N,T,A)
```

predykat jest prawdziwy, jeżeli A jest Ntym argumentem termu T

Ćwiczenie:

Proszę przećwiczyć przetwarzanie termów:

```
?- A =.. [ala, ma, asa].
?- ala(ma,kota,w(ciapki(rozowe))) =.. A.
```

Przeanalizować:

```
?- functor(ala(ma,kota),F,A).

?- CzyTo = ala, OLiczbie = 2, functor(ala(ma,kota),CzyTo,OLiczbie).

?- CzyTo = kasia, OLiczbie = 2, functor(ala(ma,kota),CzyTo,OLiczbie).

?- functor(ala(ma,kota),F,_), write('To jest funktor \''), write(F), write('\'.').

?- arg(X,ala_ma(kota,psa,schiza),A).
?- arg(2,ala_ma(kota,psa,schiza),A).

?- functor(A,riverside,3).
?- functor(A,riverside,4), arg(1,A,voices), arg(4,A,head).
```

Poniższy kod (predykaty `wyp0/2`, `wyp1/2`, `wyp2/2`, `wyp3/2`) prezentuje sposoby odwoływania się do predykatów przekazywanych jako argumenty. Przykłady użycia znajdują się w części dalszej.

```
a(1). a(2). b(4). b(3).

wyp0(F,_) :-
  call(F).

wyp1(F,X) :-
  F,
  F =.. [_,X].

wyp2(F,X) :-
  functor(Pred,F,1),
  Pred,
  Pred =.. [_,X].

wyp3(F/A,X) :-
  A = 1,
  functor(Pred,F,A),
  Pred,
  Pred =.. [_,X].
```

Wykonaj zapytania będące przykładami użycia powyższych predykatów:

```
?- wyp0(a(X),X).
?- wyp0(b(X),X).

?- wyp1(a(_),X).
?- wyp1(b(_),X).

?- wyp2(a,X).
?- wyp2(b,X).

?- wyp3(a/1,X).
?- wyp3(b/1,X).
```

3 Temat: Definiowanie operatorów

W prologu bardzo łatwo można definiować własne operatory, co ułatwia przetwarzanie danych.

Realizowane jest to przez predykat :- op(P, T, N), który definiuje N, jako operator typu T, o priorytecie P. Zobacz:

- http://www.swi-prolog.org/pldoc/man?predicate=op/3 [http://www.swi-prolog.org/pldoc/man?predicate=op/3]
- http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse40 [http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse40]

Zdefiniowane w standardzie ISO operatory to:

```
1200    xfx     -->, :-
1200    fx      :-, ?-
1150    fx      dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile
1100    xfy     ;, |
1050    xfy     ->, op*->
1000    xfy     ,
954     xfy     \
900     fy      \+
900     fx      ~
700     xfx     <, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600     xfy     :
500     yfx     +, -, /\, \/, xor
500     fx      +, -, ?, \
400     yfx     *, /, //, rdiv, <<, >>, mod, rem
200     xfx     **
200     xfy     ^
```

Wszystkie mogą być zredefiniowane!.

Gdyby nie operatory, to prosty program w Prologu:

```
go :- write('Hello '), write('World\n').

:- go.
```

Musialby wyglądać następująco (wszytko w notacji prefiksowej, zapisane jako termy):

```
:- (go,','(write('Hello '), write('World\n'))).

:- (go).
```

Uwaga: obydwa programy są równoważne. W drugim programie widać, że klauzule złożone również zapisane są jako termy.

Patrz również:

- Lean Prolog Now [http://cs.union.edu/~striegnk/learn-prolog-now/html/node84.html#subsec.l9.operators.def]

Ćwiczenie:

Proszę wpisać do pliku oper1.pl

```
:- op(100,xfy, matka).
julia matka marcin.
```

a nastepnie przetestować:

```
?- X matka Y.
```

Dopisać do pliku:

```
:- op(300, xfx, ma).
:- op(200, xfy, i).

jas ma kota i psa.
ala ma jasia i angine i dosc_agh.
rybki i kanarki.
```

Przetestować:

```
?- ma(X,Y).

?- ma(X,i(A,B)).

?- ma(A,i(B,i(C,D))).

?- Kto ma Co.

?- Kto ma Co i Cosinnego.

?- Kto ma Cos i CosInnego i Jeszcze.

?- display(jas ma kota i psa).

?- display(ala ma jasia i angine i dosc_agh).
```

Co zwróci poniższe zapytanie?

```
?- i(A,B).
```

Podpowiedź: zwróć uwagę na priorytety operatorów.

4 Temat: Konstruowanie klauzul i Metainterpretery

W Prologu występują dwa silne mechnizmy wspomagające metainterpretację kodu:

```
call(X)
```

wywołuje X, jako cel Prologu,

```
clause(Head,Body)
```

odszukuje klauzulę o nagłówku Head, gdzie Body jest unifikowane z ciałem klauzuli; w przypadku faktów Body=true.

Meta programowanie to tworzenie programów, które przetwarzają kod innych programów. Przykłady metaprogramowania znaleźć można w np.: kompilatorach, analizatorach kodu, generatorach kodu. W Prologu metaprogramowanie jest naturalną techniką, dzięki czemu pisanie programów działających w diametralnie różnych paradygmatach jest proste.

Ćwiczenie:

Proszę wczytać program rodzina1.pl

Uruchomić:

```
?- listing(kobieta).

?- call(kobieta(X)).
?- clause(kobieta(X),B).

?- listing(matka).
?- Kto = kasia, call(matka(Kto,Kogo)), write(Kto), write(' jest matka '), write(Kogo).
?- Matka = kasia, Dziecko = robert, clause(matka(Matka,Dziecko),Kiedy), write(Matka), write(' jest matka '), write(Dziecko), write(' wtedy gdy: '), writ
```

Proszę do pliku meta.pl wpisać predykat:

```
:- include(readstr).

odpowiedz :-
        write('\'matka\' czy \'ojciec\'? '),
        read_atom(X),
        write('kogo? '),
        read_atom(Y),
        Q =.. [X,Kto,Y],
        display(Q),
        call(Q),
        write(Kto), nl.
```

Uwaga: potrzebny jest dodatkowy plik: readstr.pl

a następnie przetestować i przemyśleć:

```
?- odpowiedz.

'matka' czy 'ojciec'? ojciec
kogo? robert
```

Dopisać do pliku meta.pl następujące proste metainterpretery Prologu:

```
rozwiaz1(G) :- call(G).
```

```
rozwiaz2(true) :- !.
rozwiaz2((G1,G2)) :- !,
        rozwiaz2(G1),
        rozwiaz2(G2).
rozwiaz2(G) :-
        clause(G,B),
        rozwiaz2(B).

rozwiaz3(true) :-!.
rozwiaz3((G1,G2)) :- !,
        rozwiaz3(G1),
        rozwiaz3(G2).
rozwiaz3(G) :-
        write('Wywoluje:  '), write(G), nl,
        clause(G,B),
        rozwiaz3(B),
        write('Wyjscie:   '), write(G), nl.
```

Pierwszy z nich po prostu wywołuje pojedynczy cel, tak jak powłoka SWI.

Drugi pozwala na zadanie celu złożonego z 2 części.

Trzeci działa podobnie, ale śledzi wykonywanie.

Przetestować ich działanie:

```
?- rozwiaz1(matka(kasia,X)).

?- rozwiaz1((matka(kasia,X), matka(Y,robert))).

?- rozwiaz2((matka(kasia,X), matka(Y,robert))).

?- rozwiaz3((matka(kasia,X), matka(Y,robert))).
```

Proszę pobrać kod tracerdepth.pl

Przetestować:

```
?- traced(matka(kasia,Y)).
```

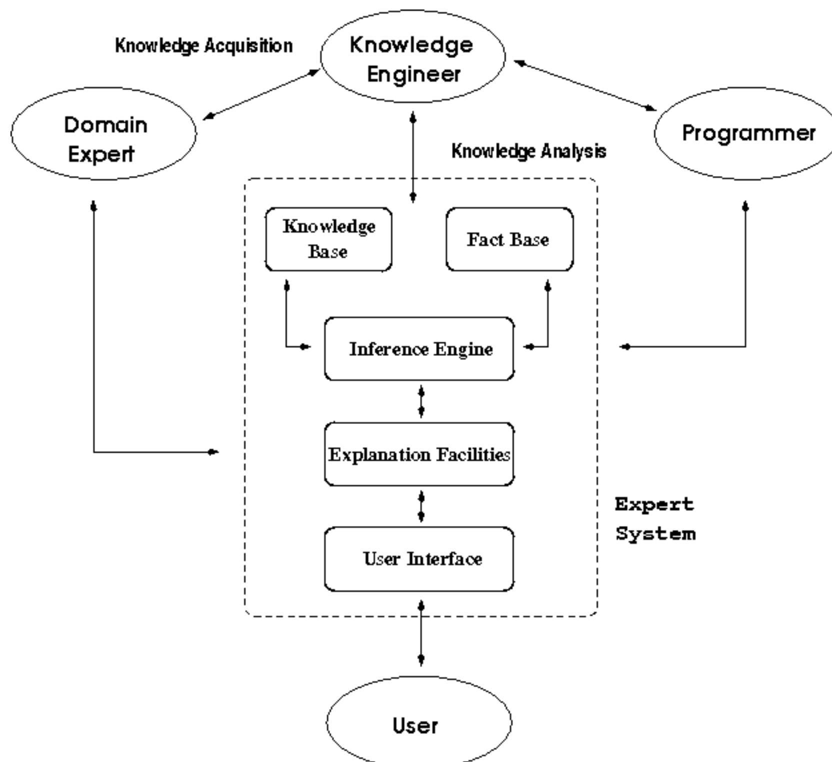
5 Temat: Systemy ekspertowe

W Prologu niezwykle prosto tworzy się systemy ekspertowe.

W systemie ekspertowym można wyróżnić następujące elementy:

- baza wiedzy, czasami dzielona na: właściwą bazę wiedzy (czyli tą którą system dysponuje stale, od początku/uruchomienia) i bazę faktów, które system odkrywa, dostaje, wypracowuje,
- mechanizm wnioskujący, który przeprowadza właściwy proces wnioskowania, tj. odnajduje rozwiązanie/odpowiedź
- mechanizm wyjaśniający, dlaczego jest to odpowiedź poprawna/dopuszczalna,
- interfejs użytkownika, pozwalający na komunikację z systemem.

Rysunek 1: Struktura Systemu Ekspertowego.



Poniżej podane są przykłady różnych systemów.

5.1 System: CAR

Źródło: Michael A. Covington, Donald Nute and André Vellino, *Prolog programming in depth*, Prentice-Hall, 1996.

Cechy:

- klauzule Prologu jako reprezentacja wiedzy
- wbudowany mechanizm Prologu
- wnioskowanie wstecz (abdukcja) [1)]
- trywialny 🙂

System: car.pl Dodatkowo korzysta on z pliku getyesno.pl

Przetestować działanie systemu. System diagnozuje przyczyną awarii samochodu.

W przypadku tego systemu elementy systemu ekspertowego są odwzorowane przez:

- predykat *defect_may_be/1* → baza wiedzy
- predykat *try_all_possibilities/0* oraz mechanizm wnioskujący Prologu → mechanizm wnioskujący
- predykat *explain/1* → mechanizm wyjaśniania
- predykaty *ask_question/1* i *user_says/2* → interfejs użytkownika

System: car.pl

Plik pomocniczy: getyesno.pl

5.2 System: BIRDS

Źródło: Dennis Merritt, *Building Expert Systems in Prolog [http://www.amzi.com/ExpertSystemsInProlog/]*, Springer-Verlag, 1989.

Cechy:

- prologowa reprezentacja reguł
- własny mechanizm wnioskujący - metainterpreter
- wnioskowanie wprzód [2)]
- wymienne bazy wiedzy

Mechanizm wnioskujący: native.pl

Bazy wiedzy: birds_kb.pl

Uruchomienie: załadować *native.pl* i wywołać `main.`

5.3 System: OOPS

Źródło: Dennis Merritt, *Building Expert Systems in Prolog [http://www.amzi.com/ExpertSystemsInProlog/]*, Springer-Verlag, 1989.

Cechy:

- własna reprezentacja reguł
- kodowanie reguł na termach Prologu
- własny mechanizm wnioskujący
- wnioskowanie wprzód
- wymienne bazy wiedzy

Mechanizm wnioskujący: oops.pl

Bazy wiedzy: room_kb.pl animal_kb.pl

Uruchomienie: załadować *oops.pl* i wywołać `main.`

5.4 System: XSHELL

Źródło: Michael A. Covington, Donald Nute and André Vellino, *Prolog programming in depth [http://www.covingtoninnovations.com/books.html#ppid]*, Prentice-Hall, 1996.

Cechy:

- klauzule Prologu jako reprezentacja wiedzy
- rozbudowana reprezentacja reguł
- wbudowany mechanizm Prologu
- wnioskowanie wstecz
- wymienne bazy wiedzy
- rozbudowane przetwarzanie

Mechanizm wnioskujący: xshell.pl

Baza wiedzy: cichlid.pl

Pliki pomocnicze: readstr.pl , readnum.pl , writeln.pl , getyesno.pl

Należy załadować plik z bazą wiedzy (ten z kolei ładuje mechanizm wnioskujący) i użyć predykatu `xshell.`

Ćwiczenie:

Przetestuj powyższe systemy.

Analizując ich pracę i sposób implementacji, proszę zwrócić uwagę na:

- sposób reprezentacji reguł, jak są zapisywane reguły w bazie wiedzy systemu, z jakich operatorów korzystają,
- sposób implementacji mechanizmy wnioskującego, na jakich rozwiązaniach się opiera.

Temat: własne systemy

Zbuduj bazę wiedzy dla własnego systemu regułowego (dla wybranej implementacji). Propozycje dziedzin podane są poniżej.

System rozpoznaje psy. Należy opisać kilka/naście znanych ras psów na podstawie: http://pl.wikipedia.org/wiki/Grupy_FCI [http://pl.wikipedia.org/wiki/Grupy_FCI], http://atlaspsow.w.interia.pl [http://atlaspsow.w.interia.pl], http://rasy-psow.com [http://rasy-psow.com], http://www.psy.elk.pl/rasypsow/ [http://www.psy.elk.pl/rasypsow/]. Warto wybrać przedstawicieli z różnych grup FCI.

W celu identyfikacji rasy trzeba wybrać kilka podstawowych cech, w tym płeć (powiązana z rozmiarem!), wagę, rozmiar, umaszczenie, kształt głowy, uszy, etc.

Podobny do w.w. system rozpoznający ptaki występujące w Polsce. Należy oprzeć się na: http://ptaki.luzik.proste.pl [http://ptaki.luzik.proste.pl], http://ptaki.zwierzeta.ekologia.pl [http://ptaki.zwierzeta.ekologia.pl].

Uwagi, komentarze, propozycje

Tu studenci mogą wpisywać swoje uwagi…

— *Grzegorz J. Nalepa [mailto:gjn@agh.edu.pl] 2009/05/06 09:13*

---

[1] Backward chaining inference, goal driven inference - stosowane w systemach diagnostycznych, w którym mamy ograniczoną (niewielką) liczbę możliwych hipotez, system stara się dowieść każdej z nich po kolei, zbierając informacje „po drodze”

[2] Forward chaining inference, data driven inference - stosowane w celu uniknięcia eksplozji kombinatorycznej, gdy możliwe jest (nieskończenie) wiele poprawnych odpowiedzi, system na podstawie danych „odpala” odpowiednie reguły