Programowanie w logice

XI Wakacyjne Warsztaty Wielodyscyplinarne. *22.08.2015*

W Prologu będziemy używać list. Poniżej mamy kilka przykładów list:

```
[mia, vincent, jules, yolanda]
[mia, robber(honey_bunny), X, 2, mia]
[]
[mia, [vincent, jules], [butch,
girlfriend(butch)]]
[[], dead(z), [2, [b, c]], [], Z,
[2, [b, c]]]
```

Powinniśmy o nich wiedzieć następujące rzeczy:

1. Listy w Prologu umieszczamy w nawiasach klamrowych ([oraz]), a ich elementy oddzielamy od siebie przecinkami.
2. Długość listy to po prostu liczba jej elementów. Np. długość listy z pierwszego przykładu to 4.
3. Z drugiego przykładu, widzimy, że nie tylko atomy mogą być elementami listy, także liczby, zmienne oraz wyrażenia złożone.
4. Dodatkowo, elementy nie muszą być różne (w 2. przykładzie mamy dwukrotnie element `mia`).
5. Trzeci przykład pokazuje specjalny typ listy: pustą listę. Omówimy ją konkretnie później.
6. W czwartym przykładzie mamy, że lista może być elementem listy. Lista ta ma 3 elementy: atom `mia`, listę `[vincent, jules]` oraz listę [butch, girlfriend(butch)].

Każda lista ma dwa elementy: głowę oraz ogon. Głową listy jest po prostu jej pierwszy element, natomiast ogonem – cała reszta. Głową listy z pierwszego przykładu jest atom `mia`, natomiast ogon – `[vincent, jules, yolanda]`. Dla listy z ostatniego przykładu głową jest `[]`, natomiast ogonem – `[dead(z), [2, [b, c]], [], Z, [2, [b, c]]]`.

Zauważmy, że ogon listy jest zawsze listą. Co zatem jest ogonem listy `dead(z)`? Jest to po prostu pusta lista (`[]`).

A co z listą pustą? Prolog niestety nie ma ani głowy, ani ogona. W zasadzie nie ma ona wewnętrznej struktury. Zobaczymy później, że jest to dość sprytnie pomyślane.

Prolog ma wbudowany operator |, który oddziela głowę od ogona listy, np.:

```
?- [Head|Tail] = [mia, vincent,
jules, yolanda].
Head = mia,
Tail = [vincent, jules, yolanda].
```

Dla pustej listy natomiast mamy:

```
?- [X|Y] = [].
false.
```

Czasami możemy użyć tzw. anonimowych zmiennych. Rozważmy predykat `second/2` zdefiniowany następująco:

```
second([_,X|_], X).
```

Przez symbol _ określamy zmienne, których nie potrzebujemy używać. Powyższy predykat znajduje drugi element listy.

Mamy zatem:

```
?- second([1,2,3],X).
X = 2.
```

Co się stanie jednak, jeżeli zapytamy:

```
?- second(X,2).
```

Prolog odpowie:

```
X = [_G5, 2|_G9].
```

Znaczy to, że wszystkie listy, które na mają jako drugi element 2 spełniają dany cel. Przez `_Gnumer` Prolog podstawia sobie swoje, nieukonkretnione jeszcze, zmienne (których użytkownik zwykle nie używa).

Ten przykład pokazuje też nam, co to jest tryb argumentów predykatu. W pierwszym przykładzie mieliśmy `second(+X,-Y)`. Oznacza to, że pierwszy argument był daną wejściową, natomiast drugi wyjściową. Z kolei w drugim przypadku użyliśmy `second(-X,+Y)`, gdzie było odwrotnie (choć niezbyt przydatne). Jeżeli dana zmienna może być zarówno wejściem i wyjściem, będziemy oznaczać ją przez `?`.

Rozważymy teraz jeden ze standardowych predykatów `append/3`, który właśnie będzie miał tryby `append(?X,?Y,?Z)`. Będziemy chcieli, aby Z było konkatenacją list X i Y.

Jest on zdefiniowany następująco:

```
append([],L,L).
append([H|T],L2,[H|L3]) :-
append(T,L2,L3).
```

Aby zobaczyć, jak to działa prześledźmy realizację celu `append([a,b,c],[1,2,3],X)`:

```
append([a, b, c], [1, 2, 3], _G518)
append([b, c], [1, 2, 3], _G587)
append([c], [1, 2, 3], _G590)
append([], [1, 2, 3], _G593)
append([], [1, 2, 3], [1, 2, 3])
append([c], [1, 2, 3], [c, 1, 2,
3])
append([b, c], [1, 2, 3], [b, c,
1, 2, 3])
append([a, b, c], [1, 2, 3], [a,
b, c, 1, 2, 3])
X = [a, b, c, 1, 2, 3]
```

> **Ćwiczenie 1.** Narysuj drzewo poszukiwań zapytania
>
> ```
> ?- append(X,Y,[a,b,c]).
> ```

Wiemy zatem, że `append(+X,+Y,-Z)` działa dość dobrze. A jakieś inne zastosowania?

Ćwiczenie 2. Zaprogramuj w Prologu poniższe predykaty:

- `prefix(P,L)`, spełniony gdy P jest listą początkowych elementów (prefiksem) listy L,
- `suffix(L,S)`, spełniony gdy S jest listą końcowych elementów (sufiksem) listy L.

Ćwiczenie 3. Zaprogramuj w Prologu predykat `member/2`, który dla celu `member(A,B)`, sprawdza, czy element A jest w liście B. Zadbaj, aby działał on w trybach `member(+A,+B)` oraz `member(-A,+B)`. Rozważ pozostałe tryby.

Ćwiczenie 4. Załóżmy, że mamy następujące fakty:

```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```

Napisz predykat `listtran(G,E)`, który będzie tłumaczył listę niemieckich liczb na angielskie, np:

```
listtran([eins,neun,zwei],X).
```

winno zwrócić listę

```
X = [one,nine,two].
```

Twój program powinien także działać w drugą stronę, dla przykładu:

```
listtran(X,[one,seven,six,two]).
X = [eins,sieben,sechs,zwei].
```

Ćwiczenie 5. Zapoznaj się w dokumentacji z predykatem `select/3` (`?- help(select/3)`) i zaimplementuj taki predykat.

Wcześniej, prosiłem Was o sprawdzenie arytmetyki w Prologu. Wiele z Was się pewnie przeraziło, kiedy okazało się, że

```
?- 2+2 = 4.
false.
```

Prawidłowym operatorem, który nakazuje wykonywanie operacji arytmetycznych jest `is`, jednak tu także możemy się spotkać z kubłem zimnej wody[1]:

```
?- 2+2 is 4.
false.
```

Zasada jest taka, że działanie do wykonania musi być po prawej stronie"

[1] bądź błędem

```
?- 4 is 2+2.
true.
```

Możemy dodatkowo złożyć to z funkcjami i zmiennymi:

```
double(X,Y) :- Y is 2*X.
?- double(2,Z).
Z = 4.
```

Jednakże:

```
double(X,Y) :- Y is 2*X.
?- double(Z,2).
ERROR: is/2: Arguments are not
   sufficiently instantiated
```

Dlaczego?

W szczególności znaki `+`, `-`, `*`, `/` są też operatorami i możemy je wywołać:

```
?- X is +(2,3).
X = 5.
```

Warto wiedzieć, że dzielenie jest całkowitoliczbowe, a modulo (resztę z dzielenia) wywołujemy funkcją `mod/2`. Oczywiście Prolog przestrzega praw kolejności działań.

Postaramy się zatem napisać funkcję, która podaje długość listy:

```
len([],0).
len([_|T], X) :-
   len(T, Xold), X is Xold+1.
```

To jest dość łatwy do zrozumienia program. Pokażemy jednak alternatywną wersję, która pokaże pojęcie akumulatora.

```
len([], A, A).
len([|T], A, L) :-
   Anew is A+1, len(T, Anew, L).
```

Powinniśmy zauważyć, że Prolog schodzi do samego dołu (pustej listy), unifikując na końcu drugą wartość z trzecią. Ta druga wartość jest właśnie nazywana akumulatorem, ponieważ zbiera (akumuluje) długość listy w trakcie wywołań rekurencyjnych. Akumulatory są bardzo często stosowane w prologowych programach.

Możemy na końcu sprytnie napisać:

```
len(List,Length) :- len(List,0,Length).
```

Do porównywania liczb używamy następujących operatorów:

```
?- 2 < 4.
true.
?- 2 =< 4.
true.
?- 3+1 =< 4.
true.
?- 4 =:= 4. % jest rowne
true.
?- 4 =\= 5. % jest rozne
true.
?- 3+1 =\= 2+2.
false.
?- 4 >= mod(4,2).
true.
```

```
?- 4-1 > 2.
true.
```

Jak widać, powyższe operatory wymuszają wykonanie działań po obu stronach.

Jeżeli używamy zmiennych, to musimy zagwarantować, że są one już zadeklarowane, tj. poniższe zapytania są poprawne:

```
?- X is 3, X < 4.
X = 3.
?- X is 3+1, X < 2.
false.
```

Natomiast poniższe powodują błędy:

```
?- X < 3.
ERROR: </2: Arguments are not
  sufficiently instantiated
?- X =:= X.
ERROR: =:=/2: Arguments are not
  sufficiently instantiated
?- X = a, X =:= a.
ERROR: =:=/2: Arithmetic: 'a/0'
  is not a function
```

Ćwiczenie 6. Napisz predykat, używający akumulatora, który znajduje maksimum w tablicy liczb.

Ćwiczenie 7. Permutacją listy $[x_1, \ldots, x_n]$ nazywamy dowolną listę postaci $[x_{i_1}, \ldots, x_{i_n}]$, gdzie $i_1, \ldots, i_n$ są parami różnymi liczbami z przedziału $1, \ldots, n$. Mniej formalnie: permutacja powstaje przez przestawienie kolejności elementów na liście. Lista n-elementowa ma $n!$ permutacji.

Permutacje można generować *przez wybieranie*, zgodnie z następującym schematem rekurencyjnym:

- Jedyną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację pewnej niepustej listy, wybierz z niej dowolny element. Wybrany element będzie głową tworzonej permutacji. Jej ogonem będzie natomiast permutacja listy pozostałej po wybraniu tego elementu.

Zaprogramuj predykat `perm/2` implementujący powyższy algorytm.

Ćwiczenie 8. Baker, Cooper, Fletcher, Miller, i Smith żyją na różnych piętrach pewnego domu, który ma dokładnie pięć pięter. Wiadomo, że:

- Baker nie mieszka na najwyższym piętrze.
- Cooper nie mieszka na najniższym piętrze.
- Fletcher nie mieszka ani na najniższym, ani na najwyższym piętrze.
- Miller mieszka wyżej niż Cooper.
- Smith nie mieszka na piętrze sąsiadującym z piętrem Fletchera.
- Fletcher nie mieszka na piętrze sąsiadującym z piętrem Coopera.

Kto gdzie mieszka?

Ćwiczenie 9. Podlistą listy $[x_1, \ldots, x_n]$ nazywamy dowolną listę postaci $[x_{i_1}, \ldots, x_{i_k}]$ dla $1 \leq i_1 < \ldots < i_k \leq n$. Podlista powstaje zatem przez usunięcie z listy niektórych elementów. Lista n-elementowa ma 2^n podlist. Zaprogramuj w Prologu predykat `sublist/2`, że cel `sublist(A,B)` jest spełniony, gdy lista B jest podlistą listy A. Przy kolejnych nawrotach predykat ten powinien wygenerować wszystkie podlisty (w dowolnej kolejności).

Ćwiczenie 10. Ile z poniższych zdań jest prawdziwych?

1. To jest ponumerowana lista dwunastu zdań.
2. Dokładnie 3 z ostatnich 6 zdań jest prawdziwych.
3. Dokładnie 2 ze zdań ponumerowanych liczbami parzystami jest prawdziwych.
4. Jeżeli zdanie 5. jest prawdziwe, to także zdania 6. i 7. są oba prawdziwe.
5. Poprzednie 3 zdania są wszystkie fałszywe.
6. Dokładnie 4 ze zdań ponumerowanych liczbami nieparzystymi są prawdziwe.
7. Albo zdanie 2. albo 3. jest prawdziwe, ale nie oba.
8. Jeżeli zdanie 7. jest prawdziwe, to zdanie 5. i 6. są oba prawdziwe.
9. Dokładnie 3 z pierwszych 6 zdań są prawdziwe.
10. Następne dwa zdania są oba prawdziwe.
11. Dokładnie jedno ze zdań 7., 8., 9. jest prawdziwe.
12. Dokładnie 4 z poprzednich zdań są prawdziwe.