Programowanie w logice

XI Wakacyjne Warsztaty Wielodyscyplinarne. *23.08.2015*

Wiadomo, że *backtracking*, który jest używany w Prologu może prowadzić do wolnych rozwiązań. Pewną receptą na radzenie sobie z tym problemem są *odcięcia*.

Jak dokładnie działa taka rzecz (oznaczana w Prologu przez `!`)? Przypomnijmy przykład z początku warsztatów:

```
max(A,B,A) :- A >= B, !.
max(A,B,B).
```

Program zawiera w pierwszej klauzuli wykrzyknik, co oznacza odcięcie. Jeżeli program tam dojdzie, to jesteśmy zobowiązani do naszych poprzednich wyborów – do wyboru odpowiednich zmiennych i odpowiedniego poddrzewa w drzewie prologowych poszukiwań.

Aby to pokazać dobitniej, przypomnijmy przykład z początku warsztatów:

```
p(X) :- a(X).
p(X) :- b(X), c(X), d(X), e(X).
p(X) :- f(X).

a(1).

b(1).
b(2).

c(1).
c(2).

d(2).

e(2).

f(1).
f(3).
```

Niby nic szczególnego, a zapytanie:

```
?- p(X)
```

zwracało `X=1; X=2; X=1; X=3.`.

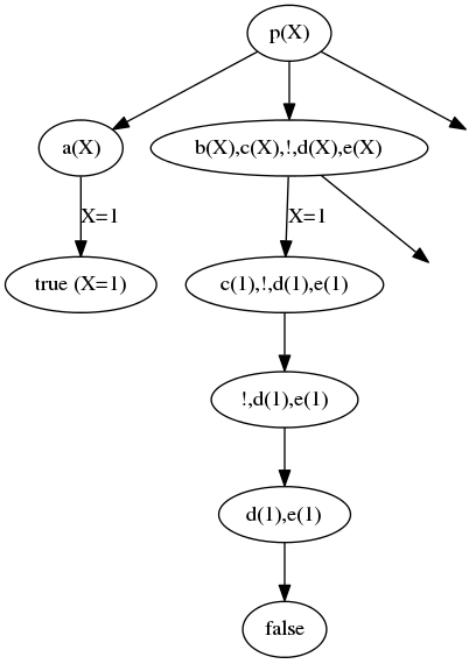
Zmieńmy zatem drugą linijkę, dodając odcięcie:

```
p(X) :- b(X), c(X), !, d(X), e(X).
```

Teraz zapytanie

```
?- p(X)
```

zwraca jedynie `X=1; false`. Dlaczego? Narysujmy nowe drzewko:



Cel `!` jest zawsze spełniony i odcina nam wszystkie gałęzie, do których mogliśmy nawrócić.

Zauważmy, że w pierwszym przykładzie odcięcie nie zmieniło nam zwracanych odpowiedzi, jedynie poprawiło wydajność programu. Takie cięcie nazywamy *zielonym*. W drugim przykładzie cięcie jest *czerwone*, gdyż faktycznie zmienia nam nawet zwracane odpowiedzi.

Ćwiczenie 1. Rozważmy następującą bazę danych:

```
p(1).
p(2) :- !.
p(3).
```

Co odpowie Prolog na następujące zapytania (wypisz wszystkie odpowiedzi w odpowiedniej kolejności)?

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

Używając odcięcia możemy zdefiniować predykat `not`:

```
not(Goal) :- Goal, !, fail.
not(Goal).
```

Powyższy predykat jest spełniony, wtedy kiedy `Goal` zawodzi. Używa on predykatu, `fail/0`, który po prostu zawodzi i wywołuje nawrót. Częściej używa się tego predykatu w innej postaci (poniżej), gdyż jest on mniej mylący. Dlaczego? Weźmy następującą bazę danych:

```
bachelor(P) :- male(P), \+ married(P).

male(adam).
male(abel).
male(cain).
```

```
female(eve).

married(adam).
married(eve).
```

Rozważmy teraz zapytania

```
?- \+ female(adam).
true.

?- bachelor(X).
X = abel ;
X = cain.
```

Wszystko wygląda poprawnie jednak:

```
?- \+ male(set).
true.

?- \+ female(set).
true.
```

Zatem predykat ten nie jest faktycznym logicznym operatorem nie, ale bardziej „nie da się udowodnić". Wciąż jest on przydatny w implementacji wielu innych predykatów:

```
% X nie unifikuje sie z Y
notequal(X,Y) :- \+ X = Y.
```