Programowanie w logice

XI Wakacyjne Warsztaty Wielodyscyplinarne. *21.08.2015*

W Prologu będziemy używać trzy zasadnicze konstrukcje:

- fakty (np. Ala jest kobietą),
- reguły (np. matka to rodzic, który jest kobietą),
- zapytania (np. kto jest matką Bartka?).

Zbiór faktów i reguł będziemy nazywali bazą wiedzy (albo także bazą danych). Programy w Prologu to po prostu baza danych. Jak zatem będziemy obsługiwać takie bazy danych? Oczywiście przez zapytania. Będziemy zadawali pytania, na które odpowiedzi znajdują się w naszej bazie danych.

Oczywiście nie brzmi to bardzo programistyczne. Przede wszystkim: czy programowanie nie polega na tym, że mówimy komputerowi po prostu, co robić? Okazuje się jednak, że podejście prologowe ma wiele zastosowań. Jednym z najczęściej przytaczanych przykładów są łamigłówki logiczne, które zwykle w Prologu mają bardzo proste i naturalne implementacje. Prolog jest także użyteczny np. w lingwistyce oraz sztucznej inteligencji.

Przejdźmy od razu do przykładów, żeby sprawdzić, jak to wszystko działa. W pliku `przyklad1.pl` umieśćmy następującą zawartość:

```
party.

guest(ala).
guest(bartek).
guest(czarek).
guest(daria).
```

Prolog używa „narodowego" rozszerzenia `.pl`, co niestety prowadzi czasami do niejasności z Perlem i niektóre edytory wymagają przestawienia języka. Rzadziej używa się też rozszerzenia `.pro`. Wszystkie komendy kończą się kropką. Proszę także zauważyć, że wszystkie nazwy zaczynają się z małej litery – jest to istotne i będzie omówione później.

To jest nasza pierwsza baza danych. Uruchamiamy teraz interaktywny edytor Prologa i kompilujemy nasz program używając wpisując `[przyklad1].` bądź `consult(przyklad1).` Mamy pierwsze fakty, to teraz sprawdźmy zapytania:

```
?- guest(bartek).
```

(nie piszemy `?-`, jedynie same zapytanie, ale będziemy dodawać znak `?-` z edytora, dla czytelności, że jest to zapytanie).

Prolog odpowie `true.` (ewentualnie `yes.`), zatem Bartek jest gościem na przyjęciu. Natomiast jeżeli spytamy:

```
?- guest(eustachy).
```

Dostaniemy `false.` (ewentualnie `no.`).

Wszystko proste? Prawda? Co się stanie jeżeli, wpiszemy np.:

```
?- pizza.
```

Dostaniemy błąd[1] (ewentualnie `no`). Dlaczego? Zapytanie o pizzę szuka w bazie danych informacji, jednak nie jest w stanie odpowiedzieć, że pizza to prawda (ba, nawet w ogóle nie jest w stanie powiedzieć co to pizza).

Możemy się zapytać jednak o party:

```
?- party.
```

i wtedy dostaniemy pozytywną odpowiedź[2].

Kolejny przykład (`przyklad2.pl`).

```
happy(ala).
happy(bartek) :- listenToMusic(bartek).
happy(czarek) :- listenToMusic(czarek).

listenToMusic(czarek) :- hasNet(czarek).

hasNet(czarek).
```

Tutaj już coś się dzieje: mamy dwa fakty – Ala jest szczęśliwa, a Czarek ma dostęp do Internetu. Z kolei pozostałe trzy klauzule to reguły. Znaczek `:-` powinien się kojarzyć z $\Leftarrow$ i oznacza po prostu if'a (jeżeli, albo jest implikowane przez). Jeżeli prawa strona (ciało reguły) jest prawdziwe, to lewa (głowa) strona jest prawdziwa.

Sprawdźmy, czy Bartek jest szczęśliwy – jest jedynie wtedy, kiedy słucha muzyki, ale nie mamy faktu, że faktycznie Bartek słucha muzyki, zatem Bartek nie jest szczęśliwy i zapytanie

```
?- happy(bartek).
```

zwróci fałsz. Natomiast Czarek, jest szczęśliwy, jeżeli także słucha muzyki, natomiast słucha on muzyki, jeżeli ma dostęp do internetu. Mamy fakt, że ma dostęp do internetu, stąd wnioskujemy, że słucha on muzyki i także jest szczęśliwy. Zatem:

```
?- happy(czarek).
yes.
```

Możemy pójść dalej i po prostu zapytać Prologa, kto jest szczęśliwy, używając zapytania:

```
?- happy(X).
```

Proszę zauważyć, że tym razem użyliśmy wielkiej litery – X jest w tym wypadku zmienną (nie jest to nazwa, lecz po prostu placeholder dla informacji). Rozróżniamy zatem zmienne od nazw zależnie od wielkości pierwszej litery (jest wielka różnica między `happy(X)` a `happy(x)`). Prolog przeszukuje naszą bazę danych i zwraca:

```
X = ala
```

Faktycznie, Ala jest szczęśliwa, jednak to nie koniec, możemy poprosić Prologa o poszukanie innych szczęśliwych osób – alternatyw dla tej odpowiedzi – używając średnika:

```
X = ala;
X = czarek.
```

[1] `ERROR: toplevel: Undefined procedure: pizza/0 (DWIM could not correct goal)`

[2] niektóre kompilatory Prologa są w stanie poprawiać „prawie" dobre zapytania (np. paty, parry, etc.)

Po drugiej odpowiedzi mamy kropkę, co oznacza, że więcej odpowiedzi nie znajdziemy. Mogliśmy także skończyć nasze poszukiwania już po pierwszej odpowiedzi, sami stawiając kropkę.

```
X = ala .
```

Zapamiętajmy zatem, że średnik oznacza alternatywę (lub) i wykorzystajmy to w kolejnym przykładzie (`przyklad3.pl`):

```
canSee(X) :- isAccepted(X); admin(X).

isAccepted(ala).
isAccepted(bartek).
isAccepted(czarek).

admin(bartek).
```

Zwrócimy tutaj uwagę na dwie rzeczy: pierwsza klauzula oznacza po prostu X może oglądać $\Leftarrow$ X jest zaakceptowany $\vee$ X jest adminem. Jednak po wpisaniu

```
?- cansee(X).
```

i wyświetleniu wszystkich odpowiedzi otrzymamy:

```
X = ala ;
X = bartek ;
X = czarek ;
X = bartek.
```

Dzieje się tak bowiem Bartek może na dwa sposoby zobaczyć daną rzecz (poprzez zaakceptowanie i poprzez fakt bycia adminem).

Ćwiczenie 1. Oto jeden z najsłynniejszym sylogizmów:

Wszyscy ludzie są śmiertelni.
Sokrates jest człowiekiem.

Zatem Sokrates jest śmiertelny.

Sformalizuj jego przesłanki w postaci klauzul prologowych i zadaj Prologowi pytanie, czy konkluzja jest prawdziwa.

Ćwiczenie 2. Rozważmy następującą bazę wiedzy:

```
wizard(ron).
hasWand(harry).
quidditchPlayer(harry).
wizard(X) :- hasBroom(X),
hasWand(X).
hasBroom(X) :- quidditchPlayer(X).
```

Jak Prolog odpowie na poniższe zapytania?

```
?- wizard(ron).
?- witch(ron).
?- wizard(harry).
?- wizard(Y).
?- wizard(hermione).
?- witch(Y).
?- witch(hermione).
```

Podaj wszystkie możliwe odpowiedzi.

Ćwiczenie 3. Wyraź poniższe twierdzenia w postaci klauzul prologowych.

1. Ptaki lubią dżdżownice.
2. Koty lubią ryby.
3. Przyjaciele lubią się wzajemnie.
4. Mój kot jest moim przyjacielem.
5. Mój kot jada wszystko to, co lubi.

Odpowiedz na pytanie, co jada mój kot.

To jeszcze `przyklad4.pl`:

```
% mother(X,Y) -- X jest matka Y
mother(X,Y) :- woman(X), parent(X,Y).

% father(X,Y) -- X jest ojcem Y
father(X,Y) :- man(X), parent(X,Y).

% parent(X,Y) -- X jest rodzicem Y
% child(X,Y) -- X jest dzieckiem Y
parent(X,Y) :- child(Y,X).

woman(ala).
woman(daria).

man(bartek).
man(czarek).

child(daria,ala).
child(daria,czarek).
child(bartek,ala).
child(bartek,czarek).
```

Komentarze w Prologu dodaje się poprzez dodanie `%` na początku linii, lub używając `/* komentarz */`.

Tutaj mamy już dość sporą bazę danych, w której mamy już określone pewne fakty rodzinne. Przecinek, jak można się domyślić, oznacza tutaj koniunkcję (zatem pierwsza klauzula to X jest matką Y $\Leftarrow$ X jest kobietą $\wedge$ X jest rodzicem Y). Prologa możemy zapytać, np.:

- kto jest rodzicem Bartka? (`?- parent(X,bartek)`)
- kto jest matką Darii? (`?- mother(X,daria)`)
- kto jest czyim dzieckiem? (`?- child(X,Y)`)

Poza ostatnim zapytaniem, w którym pierwszy raz użyliśmy dwóch zmiennych (oraz dostajemy pary odpowiedzi), ciekawe jest drugie zapytanie – przyjrzyjmy się mu bardziej.

```
?- mother(X,daria).
X = ala ;
false.
```

Tutaj mamy przypadek, w którym Prolog na początku pyta nas, czy ponownie chcemy przejść dalej, ale po wpisaniu średnika dostajemy od razu `false`. Żeby zobaczyć dlaczego tak się dzieje musimy opisać jak działa maszyna prologowa.

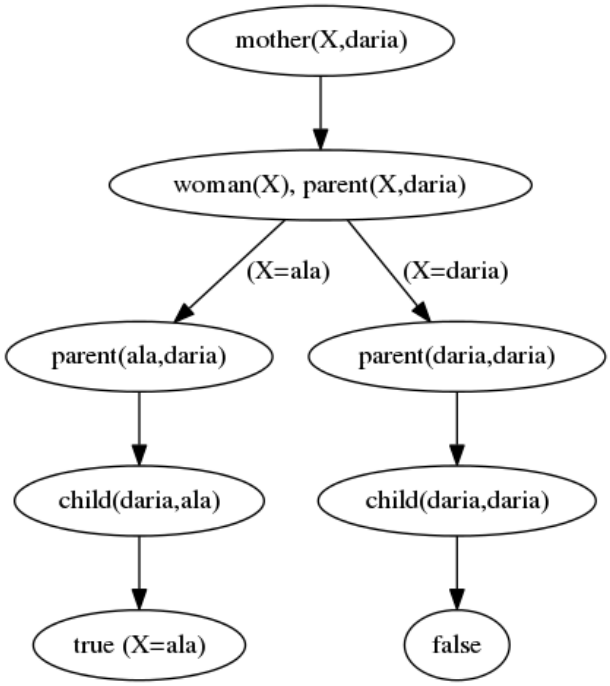
Prolog dostając zapytanie `?- mother(X,daria).` szuka w bazie danych klauzul związanych z matką. Ma takie jedno:

```
mother(X,Y) :- woman(X), parent(X,Y).
```

W tym momencie klauzula ta unifikuje się z naszym zapytaniem, to jest pod Y podstawiona zostaje daria. Zatem teraz aby spełnić cel `mother(X,daria)` musi spełnić dwa cele: `woman(X).` oraz `parent(X,daria).`

Ten pierwszy jest podobnie sprawdzany po kolei w bazie danych i znajdujemy `woman(ala).` zatem Prolog podstawia X=ala i sprawdza kolejny warunek `parent(ala,daria).` on jest spełniony jedynie gdy `child(daria,ala).` i faktycznie taki fakt mamy dany w bazie danych, zatem X to Ala. Możemy jednak szukać innych odpowiedzi i Prolog będzie szukał innych odnóg (takie podejście nazywamy *nawrotem*). Znajdzie konkretnie jedną: `woman(daria)` i dalej będzie sprawdzał, czy `parent(daria, daria)` i `child(daria, daria)`, tutaj jednak nie znajdzie żadnej możliwości i zwróci fałsz.

Całość możemy przestawić na tzw. drzewku poszukiwań:



Pokażemy jeszcze jeden przykład:

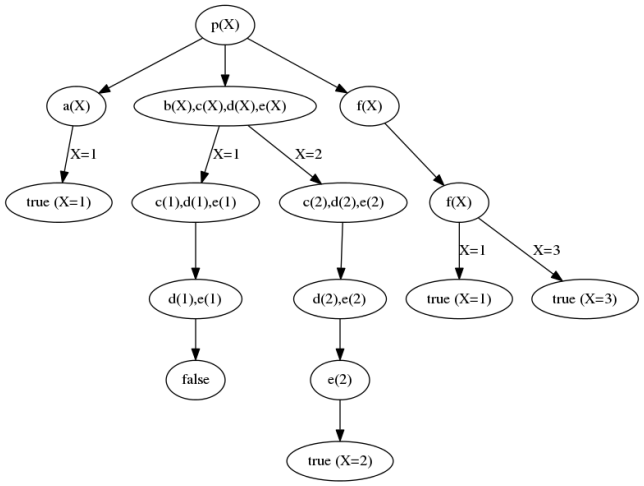
```
p(X) :- a(X).
p(X) :- b(X), c(X), d(X), e(X).
p(X) :- f(X).

a(1).

b(1).
b(2).

c(1).
c(2).

d(2).

e(2).

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

Razem z zapytaniem:

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

Cel `p(X)` ma trzy możliwości (każda to nowa odnoga drzewa). Cel `a(X)` ma jedynie jedną możliwość w bazie danych (`X=1`), zatem zwracamy `X=1`. Druga możliwość jest bardziej skomplikowana i musi wpierw znaleźć elementy spełniające `b(X)`. Mamy dwie możliwości, które sprawdzamy. `X=1` zwróci fałsz przy celu `d(1)`, natomiast po nawrocie mamy `X=2`, które jest spełnione. Z kolei trzecia możliwość daje nam elementy spełniające `f(X)` – mamy tu `X=1` (ponownie) oraz `X=3`.

Naszkicujemy drzewko:



Faktycznie:

```
?- p(X).
X = 1;
X = 2;
X = 1;
X = 3.
```

Postarajmy się teraz wszystko uporządkować. W Prologu mamy cztery główne typy danych: atomy, liczby, zmienne i wyrażenia złożone. Atomy to są najprostsze jednostki (takie jak `party`, `czarek`, `mother`), które już używaliśmy. Także symbole `:-` lub `,` są atomami. Podobnie używaliśmy zmiennych (`X`, `Y`, ale także może to być `Kto`, `JakasDlugaNazwa`). Te wszystkie trzy elementy możemy składać w bardziej skomplikowane struktury, takie jak `woman(ala)`. Dodatkowo, możemy budować wyrażenia złożone z innych wyrażeń złożonych. Dla przykładu weźmy następujące wyrażenie:

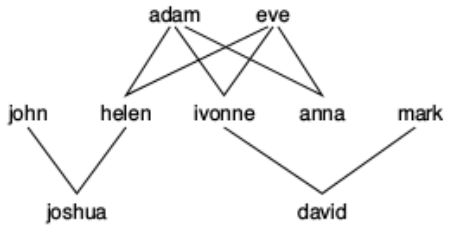
```
funkcja(Zmienna, zloz(zloz(zloz(stala)))).
```

Wyrażenie to składa się z funktora (pierwszego elementu) `funkcja`, po którym następują argumenty (w nawiasie, oddzielone przecinkami). Nasze wyrażenie ma dwa argumenty: zmienną `Zmienna` oraz wyrażenie `zloz(zloz(zloz(stala)))`. Liczbę argumentów danego wyrażenia nazywamy arnością tego predykatu i oznaczamy `funkcja/2` – szczególnie jest to używane np. w dokumentacji Prologa. Jest dopuszczalne używanie takich samych nazw funktorów z różną liczbą argumentów, np: `children/2` oraz `children/3`.

Ćwiczenie 4. Dane są predykaty: `parent/2`, `male/1`, `female/1`. Cel `parent(a,b)` jest spełniony wówczas, gdy a jest rodzicem b, zaś cele `male(a)` i `female(a)` – gdy a jest odpowiednio mężczyzną lub kobietą. Korzystając z powyższych predykatów zdefiniuj predykaty:

- `sibling(a,b)` (spełniony gdy a i b są rodzeństwem),
- `sister(a,b)` (spełniony gdy a jest siostrą b),
- `grandson/2` (spełniony gdy a jest wnukiem b),
- `cousin/2` (gdy a jest kuzynem b),
- `descendant/2` (gdy a jest potomkiem b),
- `ismother/1` (gdy a jest matką),

Dołącz do programu zbiór faktów definiujących predykaty `parent`, `male` oraz `female` dla następujących zależności rodzinnych:



Zadaj Prologowi następujące pytania:

1. Czy John jest potomkiem Marka?
2. Kto jest potomkiem Adama?
3. Kto jest siostrą Ivonne?
4. Kto ma w tej rodzinie kuzyna i kim ten kuzyn jest?

Narysuj prologowe drzewa poszukiwań dla powyższych zapytań.

Ćwiczenie 5. Zbuduj prolgową bazę danych o bezpośrednich połączeń kolejowych między miastami (fakt, że istnieje połączenie z miasta A do miasta B nie implikuje, że istnieje połączenie odwrotne):

z	do
wroclaw	warszawa
szczecin	gniezno
wroclaw	krakow
warszawa	katowice
wroclaw	szczecin
gniezno	gliwice
szczecin	lublin
lublin	gliwice

Zapytaj maszynę prologową:

- czy istnieje bezpośrednie połączenie z Wrocławia do Lublina?
- z jakimi miastami ma Wrocław bezpośrednie połączenie?
- z jakich miast można dojechać do Gliwic z dokładnie jedną przesiadką?
- z jakich miast można dojechać do Gliwic z co najwyżej dwoma przesiadkami? Czemu niektóre miasta są wymienione więcej niż raz?

Spróbuj zaimplementować na wzór predykatu `descendant/2` z poprzedniego zadania predykat `connection/2` spełniony wówczas, gdy istnieje połączenie pomiędzy podanymi miastami z dowolną liczbą przesiadek.

Bartosz Kostka