Przedmowa do artykułu „Prolog. Zalety programowania w logice”

autorstwa Piotra Orzeszka w czasopiśmie *Linux*+, 2010

W artykule na temat zalet programowania w logice prezentowany jest program w Prologu, który rozwiązuje zagadkę Einsteina (a także treść zagadki). Autor przesadził pisząc o wielotygodniowych rozwiązaniach dla tej zagadki w takich językach jak C++, ponieważ istnieją całkiem szybkie rozwiązania w C++, niemniej żadne z tych rozwiązań nie jest tak krótkie i treściwe jak w Prologu. Dla ułatwienia oceny przejrzystości kodu w Prologu poprzez porównanie z innymi językami – umieszczam na końcu tego dokumentu (4 ostatnie strony) listing programu w C++ (wersja zoptymalizowana). (AK)

Prolog

Zalety programowania w logice

W książce „2001: Odyseja kosmiczna" Arthura C. Clarke'a zainstalowany na statku „Discovery 1" komputer pokładowy HAL 9000 sprawuje nadzór nad przebiegiem misji. Zarządza wszelkimi aspektami lotu, a także jako jedyny zna prawdziwy cel wyprawy. Nie tylko przetwarza dane, aby udostępnić je załodze, lecz przede wszystkim samodzielnie podejmuje decyzje. Wyprowadza wnioski z przesłanek. Dzięki temu staje się prawie równie inteligentny jak ludzie.

Zagadnienia sztucznej inteligencji należą niewątpliwie do najbardziej fascynujących zagadnień współczesnej informatyki. Osobiście gdyby przyszło mi kiedyś programować komputer pokroju HAL-a wykorzystałbym do tego celu język Prolog. Jest to jeden z niewielu języków programowania, o którym można powiedzieć, że wystarczy w nim opisać problem aby maszyna sama znalazła jego rozwiązanie.

Prolog został stworzony w 1971 roku jako język programowania służący do automatycznej analizy języków naturalnych. Jego autorami są Alain Colmeraurer i Phillipe Roussel. Prolog oparty jest o rachunek predykatów pierwszego rzędu.

Dzisiaj najwięcej uwagi poświęca się obiektowym językom programowania. Obecnych jest wiele języków i ich specyfikacji. Cechą wspólną języków takich jak Java, C++, C# a nawet PHP, Pyton czy Ruby on Raili jest to, że nadają się one dość dobrze do pewnych konkretnych zastosowań. Języki obiektowe robią dziś furorę jako narzędzie projektowania szeroko rozumianych aplikacji biznesowych, portali Web i oprogramowania użytkowego.

Są jednak takie dziedziny, w których paradygmat programowania obiektowego sprawdza się znacznie gorzej. Warto pomyśleć nad innymi dostępnymi rozwiązaniami. Często zaoszczędzimy bowiem zarówno na konieczności pisania długiego kodu, jak i nieefektywności obliczeniowej rozwiązania.

Jeśli zatem zajmujemy się zagadnieniami związanymi z analizą i rozumieniem języka naturalnego, rozwiązywaniem równań symbolicznych, logiką matematyczną, automatycznym projektowaniem, analizą struktur biochemicznych lub też innymi problemami z zakresu sztucznej inteligencji winniśmy zastanowić się, czy zastosowanie paradygmatu programowania w logice nie będzie dobrym wyborem.

Programowanie w Prologu znacząco różni się od programowania w językach obiektowych czy proceduralnych. Zamiast pisać algorytm, ukazujący kolejne kroki do rozwiązania problemu programista zobowiązany jest do dokładnego, sformalizowanego opisu na czym polega sam problem. Prolog sam wykona resztę. Programowanie w logice jest opisywaniem obiektów i łączących je zależności (relacji). Gdy już tego dokonamy, będziemy mogli zwrócić się do naszego programu z zapytaniem o relacje i powiązania, których co prawda nie podaliśmy, jednak system jest w stanie wywnioskować je z wprowadzonych już danych.

Dostępne oprogramowanie i jego instalacja

Język Prolog doczekał się wielu implementacji. Obecnie istnieją wersje działające między innymi w systemach Linux, Windows, MS-DOS i Macos X. Z bardziej znanych można wymienić GNU Prolog, SWI-Prolog, Visual Prolog będący produktem komercyjnym oraz tuProlog rozwijany jako implementacja działająca na wirtualnej maszynie Javy.

Na potrzeby tego artykułu skorzystamy ze SWI-Prolog, który jest dostępny dla większości dystrybucji Linuxa w postaci prekompilowanych binariów. Chcąc zainstalować SWI-Prolog w systemie Linux Ubuntu wystarczy wydać w konsoli polecenie: `sudo apt-get install swi-prolog`.

Aby uruchomić przykładowe programy zamieszczone poniżej w postaci listingów należy zapisać je wpierw w postaci plików tekstowych z rozszerzeniem *.pl* a następnie wywołać poleceniem `swipl nazwapliku.pl`. Uruchomiony zostanie interaktywny interpreter Prologu, w którym będziemy mogli wprowadzać zapytania. Ten sam efekt możemy uzyskać także wywołując polecenie swipl, a następnie po uruchomieniu się interpretera wprowadzając na-

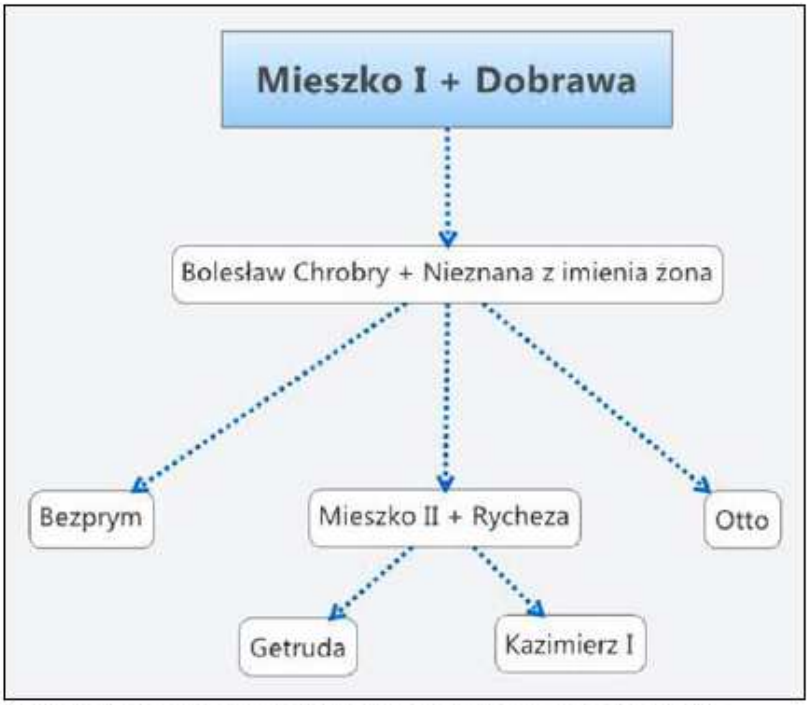
zwę pliku źródłowego, bez rozszerzenia, ujętą w nawiasach kwadratowych i zakończoną kropką. Spowoduje to wczytanie kodu programu do interpretera. Warto jeszcze dodać, że wprowadzenie `halt.` spowoduje wyjście z trybu interaktywnego do linii poleceń powłoki Linuxa.

Obiekty i relacje

Jak wspomnieliśmy uprzednio programowanie w Prologu polega na opisaniu problemu w sposób sformalizowany. To co musimy wykonać to: opisanie pewnych faktów o obiektach i łączących je wzajemnych relacjach oraz ewentualne podanie reguł odnośnie wspomnianych. Następnie możemy zapytać system o obiekty i ich wzajemne relacje.

Przypuścimy, że tworzymy program, który będzie służył analizom drzew genealogicznych sławnych rodów. Pragniemy aby nasz system „rozumiał" takie pojęcia jak ojciec, matka, brat, siostra, dziadek i tym podobne. Projektowany system miałby umożliwiać ustalenie wzajemnych koligacji rodzinnych po wprowadzeniu danych. Jak zaprogramować tego problem w Prologu? Posłużmy się praktycznym przykładem.

Analizie poddamy drzewo genealogiczne Piastów polskich (patrz Rysunek 1). Jako pierwszy będziemy chcieli zapisać fakt, że Mieszko I jest mężem Dobrawy. W Prologu możemy użyć do tego następującej składni: `maz(mieszkoI, dobrawa).` Warto już w tym miejscu zaznaczyć dwie rzeczy. W Prologu stałe zapisujemy rozpoczynając od małej litery. W ten oto sposób musimy też zapisać mieszkoI i dobrawa. Relacja bycia mężem nie jest relacją symetryczną. Fakty `maz(mieszkoI, dobrawa).` i `maz(dobrawa, mieszkoI).` oznaczają co innego i Prolog traktuje je jako inne. Oczywiście od nas zależy jak będziemy czytać daną relację. Musimy jednak zwrócić uwagę, że kolejność stałych ma istotne znaczenie. Za-



Rysunek 1. *Wybrany fragment drzewa genealogicznego rodu Piastów.*

Listing 1. *Pierwszy program w Prologu – Genealogia rodu Piastów*

```
maz(mieszkoI, dobrawa).
syn(boleslawChrobry, mieszkoI).
syn(bezprym, boleslawChrobr).
syn(mieszkoII, boleslawChrobry).
syn(otto, boleslawChrobry).
zona(rycheza, mieszkoII).
corka(gertruda, mieszkoII).
brat(kazimierzI, getruda).

ojciec(X,Y) :- syn(Y,X).
maz(X,Y) :- zona(Y,X).
```

notujmy jeszcze kilka faktów odnoszących się do naszego drzewa genealogicznego.

Synem Mieszka I jest Bolesław Chrobry – `syn(boleslawChrobry, mieszkoI).` Bolesław Chrobry ma synów Bezpryma, Mieszka II i Otta – `syn(bezprym, boleslawChrobry). syn(mieszkowi, boleslawChrobry). syn(otto, boleslawChrobry).` Żoną Mieszka II jest Rycheza – `zona(rycheza, mieszkoII),` Córką Mieszka II jest Gertruda – `corka(Gertruda, mieszkoII).` Bratem Gertrudy jest Kazimierz I – `brat(kazimierzI, gertruda).`

Poza zbiorem faktów program w Prologu powinien zawierać jakieś reguły. Posłużą one do wyprowadzania nowych faktów poprzez wnioskowanie. Co prawda można by sobie wyobrazić program bez reguł, jednak z punktu widzenia naszych rozważań byłby on mało interesujący.

Dodajmy na początek dwie bardzo proste reguły. `ojciec(X,Y) :- syn(Y,X).` oraz `maz(X,Y) :- zona(Y,X).` Duże litery X i Y reprezentują zmienne. Reguły odnoszą się bowiem do wielu obiektów i jak już mówiliśmy pozwalają wyprowadzać nowe fakty. Znak „:-" czytać będziemy „jeśli", reprezentuje on tak zwany funktor odwrotnej implikacji wyrażanej zazwyczaj w logice znakiem „→". Wprowadzone przez nas reguły przeczytamy odpowiednio „X jest ojcem Y jeśli Y jest synem X" oraz „X jest mężem Y jeśli Y jest żoną X".

Zbiór opisanych dotąd faktów oraz reguł możemy skomponować w program przedstawiony na Listingu 1.

Tak przygotowany program zapisujemy w pliku tekstowym nadając mu nazwę listing1.pl. Możemy teraz uruchomić interpreter języka prolog podając jako parametr nazwę pliku źródłowego. Wpisujemy w linii poleceń `swipl listing1.pl.` Interpreter po załadowaniu wyświetla znak zachęty. W tym momencie możemy sprawdzić działanie naszego prostego programu wysyłając zapytania do systemu.

Zapytanie w Prologu przybiera postać wyrażenia, w którym występuje co najmniej jedna

zmienna. Interpreter po otrzymaniu takiego zapytanie stara się ukonkretnić zmienne występujące w wyrażeniu poprzez stałe. W przypadku, gdy istnieje wiele rozwiązań danego zadania, Prolog wyświetla pierwsze z nich, a następnie czeka na reakcję ze strony użytkownika.

Na początek zadajmy pytanie dotyczące jakiegoś prostego faktu, który sami wprowadziliśmy.

Przykładowo: „Kto jest synem Mieszka I ?" – `syn(X, mieszkoI)`. System nie posiada informacji o innych synach pierwszego władcy Polski. Po wciśnięciu klawisza [enter] otrzymujemy jednoznaczną odpowiedź: `X=boleslawChrobry`.

Spróbujmy teraz innego zapytania `ojciec(X, boleslawChrobry)`. Otrzymamy odpowiedź `X=mieszkoI`. Zwróćmy uwagę, że fakt ten został wywnioskowany z użyciem reguły ojciec(X,Y) :- syn(Y,X) ukazującej relację jaka zachodzi pomiędzy ojcem a synem.

Możemy dopisać do naszego pliku źródłowego następującą regułę `dziadek(X,Z) :- ojciec(X,Y), ojciec(Y,Z)`. Znak przecinka używany jest w Prologu dla oznaczenia koniunkcji. Jest to odpowiednik operatora && w C++ czy „and" w Turbo Pascalu. Naszą nową regułę przeczytamy zatem „X jest dziadkiem Z jeśli X jest ojcem Y i jednocześnie Y jest ojcem Z".

W tym momencie możemy zapisać plik źródłowy i ponownie uruchomić interpreter. Myślę, że spodziewacie się już jaką odpowiedź uzyskamy wprowadzając zapytanie `dziadek(X, otto) lub dziadek(X, bezprym)`.

Listing 2. *Genealogia rodu Piastów – ciąg dalszy*

```
kobieta(dobrawa).
kobieta(rycheza).
Kobieta(gertruda).
mezczyzna(mieszkoI).
mezczyzna(boleslawChrobry).
mezczyzna(mieszkoII).
mezczyzna(kazimierzI).
maz(mieszkoI, dobrawa).
maz(mieszkoI, dobrawa).
syn(boleslawChrobry, mieszkoI).
syn(bezprym, boleslawChrobr).
syn(mieszkoII, boleslawChrobry).
syn(otto, boleslawChrobry).
zona(rycheza, mieszkoII).
corka(gertruda, mieszkoII).
brat(kazimierzI, getruda).

ojciec(X,Y) :- syn(Y,X), mezczyzna(X).
matka(X,Y) :- syn(Y,X), kobieta(X).
ojciec(X,Y) :- corka(Y,X), mezczyzna(X).
matka(X,Y) :- corka(Y,X), kobieta(X).
maz(X,Y) :- zona(Y,X).
dziadek(X,Z) :- ojciec(X,Y), ojciec(Y,Z).
```

Aby dodać reguły pozwalające nam zdefiniować relacje takie jak bycie siostrą, bratem czy też bycie matką potrzebne nam będą dodatkowe informacje w naszej logicznej bazie danych. Musimy mianowicie określić płeć każdej z osób. Zrobimy to wykorzystując jednoargumentowe predykaty. Fakt, że Mieszko I jest mężczyzną zapiszemy jako `mezczyzna(mieszkoI)`. Pozwoli to dokonać rozróżnienia między stosunkiem ojcostwa i macierzyństwa. Zmodyfikowany program wraz z kolejnymi dołączonymi faktami i regułami został podany w Listingu 2.

Możemy teraz zmodyfikować regułę `ojciec(X,Y) :- syn(Y,X), mezczyzna(X)`. Przeczytamy ją w następujący sposób X jest ojcem Y jeśli Y jest synem X i zarazem X jest mężczyzną. Jak zdefiniować stosunek macierzyństwa? X jest matką Y jeśli Y jest synem X (lub córką) i zarazem X jest kobietą. Wykorzystujemy w celu zapisu dwie reguły `matka(X,Y) :- syn(Y,X), kobieta(X).` oraz `matka(X,Y) :- corka(Y,X), kobieta(X).`

Jeśli chcemy sprawdzić działanie zmodyfikowanego programu należy uruchomić ponownie interpreter Prologu i wczytać kod zamieszczony jako Listing 2. Możemy sprawdzić teraz działanie takich zapytań jak `matka(X, boleslawChrobry)`. lub `ojciec(X, gertruda)`. Zadanie jak zdefiniować relacje stosunku pokrewieństwa takie jak bycie bratem lub siostrą zostawiamy jako pracę domową uważnym czytelnikom. Warto także poeksperymentować z programem dopisując własne fakty i reguły. Cennym może okazać się zaczerpnięcie dodatkowej wiedzy z pozycji książkowych polecanych w ramce z odnośnikami do literatury omawiającej programowanie w Prologu.

Zagadka Einsteina

Pragnąc ukazać wyższość Prologu nad innymi językami programowania warto powołać się na przykład zadania nazywany „zagadką Einsteina". Jest to znane zagadnienie programistyczne. Rozwiązanie tej zagadki można znaleźć implementując odpowiedni algorytm w języku programowania obiektowego lub proceduralnego. Problem jednak leży w tym, że złożoność obliczeniowa zagadnienia powoduje, że dobrze napisany program w C++ znajduje rozwiązanie dopiero po wielotygodniowych obliczeniach. Prolog radzi sobie z zagadką w mniej niż sekundę.

Autorstwo zagadki przypisuje się Albertowi Einsteinowi. Podobno miał on powiedzieć kiedyś, że tylko 2% ludzi umie ją rozwiązać (mowa była zapewne o rozwiązaniu bez użycia komputera). Niektórzy uważają natomiast, iż to nie Einstein lecz pisarz brytyjski, autor „Alicji w Krainie Czarów" Lewis Caroll pozostawił ludzkości ów problem do rozwiązania.

Obecnie istnieje wiele możliwych sformułowań zagadki Einsteina. Jedno z nich przytaczamy w Ramce 1. Stworzenie efektywnego algorytmu rozwiązującego tak postawiony problem jest nie lada wyzwaniem. Warto uświadomić sobie złożoność obliczeniową zagadnienia. Jeśli zdecydowalibyśmy się używać nieoptymalnego algoryt-

mu typu brute force generując wszystkie możliwe permutacje danych ich liczba wyniosłaby 24 miliardy.

Liczba możliwych kombinacji tworzy tablicę o pięciu rzędach odpowiednio (narodowość mieszkańca, kolor domu, preferowany napój, rodzaj palonych papierosów, hodowane zwierzę) i pięciu kolumnach (po jednej kolumnie dla każdego domu). Tablicę taką można wypełnić na $5!^5 = (1*2*3*4*5)^5 = 120^5 = 24\ 883\ 200\ 000$ sposobów. Następnie należałoby wybrać jedną permutację odpowiadającą rozwiązaniu zadania.

Implementacja programu rozwiązującego omawianą zagadkę w Prologu zajmuje 21 linijek kodu. Na komputerze autora program wykonał się w niecałą sekundę. Nie będziemy tutaj przytaczać rozwiązania zagadki. Jeśli ktoś z czytelników jest ciekawy, który z mężczyzn hoduje zebrę zachęcamy do przepisania programu z Listingu 3 do pliku i samodzielnego sprawdzenia. Po załadowaniu interpretera Prologu z parametrem w postaci nazwy pliku źródłowego należy w tym celu jako zapytanie wprowadzić `ma_zebre(Ulica, Kto).`

Listing programu prologowego rozwiązującego zagadkę Einsteina może wydawać się na początku niezrozumiały. Tak naprawdę jego dokładne wytłumaczenie wiersz po wierszu wymagałoby więcej miejsca niż przewidziano w tym artykule. Pod-

Listing 3. *Program rozwiązujący zagadkę Einsteina*

```
obok(X, Y, List) :- po_prawej(X, Y, List).
obok(X, Y, List) :- po_prawej(Y, X, List).
po_prawej(L, R, [L | [R | _]]).
po_prawej(L, R, [_ | Rest]) :- po_prawej(L, R, Rest).
ma_zebre(Ulica, Kto) :-
Ulica = [_Dom1, _Dom2, _Dom3, _Dom4, _Dom5],
member(dom(czerwony, anglik, _, _, _), Ulica),
member(dom(_, szwed, pies, _, _), Ulica),
member(dom(zielony, _, _, kawa, _), Ulica),
member(dom(_, dunczyk, _, herbata, _), Ulica),
po_prawej(dom(zielony,_,_,_,_), dom(bialy,_,_,_,_), Ulica),
member(dom(_, _, slimak, _, old_gold), Ulica),
member(dom(zolty, _, _, _, kools), Ulica),
[_, _, dom(_, _, _, mleko, _), _, _] = Ulica,
[dom(_, norweg, _, _, _) | _] = Ulica,
obok(dom(_, _, _, _, chesterfields), dom(_, _, lis, _, _), Ulica),
obok(dom(_, _, _, _, kools), dom(_, _, kon, _, _), Ulica),
member(dom(_, _, _, sok_pomaranczowy, lucky_strike), Ulica),
member(dom(_, japonczyk, _, _, parliaments), Ulica),
obok(dom(_, norweg, _, _, _), dom(niebieski, _, _, _, _), Ulica),
member(dom(_, Kto, zebra, _, _), Ulica).
```

stawowa idea programu jest jednak bardzo prosta. Na początku wprowadzono dwie reguły definiujące trójargumentowy predykat `obok(X,Y,Lista)`. Dany dom X znajduje się obok domu Y, jeśli dom X jest po prawej stronie domu Y (pierwsza reguła) lub też gdy dom Y jest po prawej stronie domu X (druga reguła). Alternatywa możliwości wyrażana jest tutaj poprzez wprowadzenie dwu odrębnych reguł.

Predykat `po_prawej` zdefiniowano przy użyciu operacji na listach. Jego definicja składa się z dwu reguł i jest oparta na rekurencji.

Na cały dalszy kod programu składa się definicja predykatu `ma_zebre(Ulica, Kto)` kolejne zdania zagadki stanowią kolejne elementy koniunkcji, które muszą być spełnione aby wspomniany predykat został ukonkretniony i znalezione rozwiązanie.

Rysunek 2. *Interpreter SWI-Prolog uruchomiony w konsoli Linuxa.*

Podsumowanie

Wiele osób będących doświadczonymi programistami zauważy z pewnością, że Prolog łączy w sobie cechy nie tylko języka programowania lecz również systemu automatycznego dowodzenia twierdzeń oraz narzędzia do tworzenia systemów eksperckich.

Z punktu widzenia logika, Prolog nie stanowi systemu automatycznego dowodzenia twierdzeń. Ograniczenie zbioru rozpatrywanych formuł do klauzul Horna nie pozwala na zapisywanie w postaci programów w Prologu bardziej skomplikowanych teorii. Dotyczy to w szczególności zdań asertywnych zawierających funktor negacji, czyli wyrażających, że jakaś relacja nie zachodzi.

W Prologu istnieje co prawda mechanizm stanowiący namiastkę negacji. Możliwe jest wprowadzenie funktora negacji interpretowanego w następujący sposób: jeśli system nie jest w stanie udowodnić danej formuły, można wtedy przyjąć, że zachodzi jej zaprzeczenie. Takie rozwiązanie nie jest jednak uniwersalne i stosowanie go w pewnych przypadkach może być przyczyną pojawienia się zapętleń w trakcie ukonkretniania zapytania.

Pomimo wspomnianych niedoskonałości Prolog ma jednak zaletę, której nie sposób pominąć. Jest on rewelacyjnie szybki. Cecha ta powoduje, że warto rozważyć użycie tego języka podczas rozwiązywania pewnych zadań, być może nie tylko związanych z problematyką sztucznej inteligencji. Wiele typowych algorytmów, które znane są z akademickich kursów programowania obiektowego można rozwiązać bardzo efektywnie także w Prologu.

Czy warto uczyć się Prologu jako języka programowania? Niewątpliwie tak, po pierwsze dlatego, że paradygmat programowania w logice zupełnie różni się od paradygmatu projektowania obiektowego. Z tego też powodu podejmując się nauki, wyrobimy w sobie zupełnie nowe formy abstrakcyjnego myślenia obce wielu programistom.

Zagadka Einsteina

Pięciu ludzi różnych narodowości zamieszkuje pięć domów w pięciu różnych kolorach. Każdy z nich pali papierosy jednej z pięciu różnych marek i pija jeden z pięciu różnych napojów. Hodują zwierzęta pięciu różnych gatunków. Pytanie brzmi: który z nich hoduje zebrę?

1. Anglik mieszka w czerwonym domu.
2. Szwed hoduje psa.
3. W zielonym domu pija sie kawę.
4. Duńczyk pija herbatę.
5. Zielony dom stoi bezpośrednio po prawej stronie białego domu.
6. Palacz Old Goldów hoduje ślimaka.
7. Koolsy pali się w żółtym domu.
8. Mieszkaniec środkowego domu pija mleko.
9. Norweg zamieszkuje pierwszy dom.
10. Mężczyzna, który pali Chesterfieldy zamieszkuje dom obok hodowcy lisa.
11. Koolsy pali się w domu znajdującym się obok domu, w którym trzymany jest koń.
12. Palacz papierosów marki Lucky Strike pija sok pomarańczowy.
13. Japończyk pali Parliamentsy.
14. Norweg mieszka obok niebieskiego domu.

Jednocześnie zdobędziemy umiejętność implementacji wielu złożonych problemów.

Wydaje się, że przyszłość informatyki będzie coraz mocniej związana z szeroko rozumianą sztuczną inteligencją. Obecnie komputery służą nam najczęściej jako medium dostępu do informacji. Jednocześnie jednak coraz bardziej widoczne staje się to, że informacji docierającej do nas jest zbyt dużo, jesteśmy nią niemalże zalewani. Stąd prawdopodobnym jest, że w niedalekiej przyszłości preferowane będą te technologie, które pozwolą na dostarczanie użytkownikom tylko wybranych, w pełni spersonalizowanych treści. Wtedy jednak, podstawowa rola oprogramowania, którą aktualnie jest przetwarzanie danych zmieni się na podejmowanie decyzji jakie treści udostępnić końcowemu użytkownikowi, aby zadośćuczynić jego żądaniu. Pojawią się inteligentne wyszukiwarki rozumiejące język naturalny, analizujące kontekst wypowiedzi i udostępniające w postaci wyników tylko te i dokładnie te informacje jakich potrzebujemy. Być może przestrzeń Internetu coraz częściej będą urozmaicać nam „rozumne" czaterboty, z którymi będziemy mogli nie tylko poprowadzić zabawną konwersację, lecz wypytać o szczegóły oferty, poprosić o zarezerwowanie biletu na koncert lub do teatru, jak też automatyczne przygotowanie raportu na wybrany temat.

Zagadnienia, przy których rozwiązywaniu istotna jest umiejętność wnioskowania wydają się mieć jednak tę cechę, że niezbyt dobrze są opisywalne w postaci algorytmicznej. Programista musiałby znać bowiem nie tylko problem lecz również metodę krok po kroku prowadzącą do rozwiązania. Prolog natomiast nie wymaga znajomości już na początku przepisu prowadzącego do odpowiedzi. To czyni z niego język doskonały do omawianych zagadnień.

Polecana literatura

- W. F. Clocksin, C. S. Mellish, *Prolog. Programowanie*, Helion, Gliwice 2006.
- I. Bratko, *Prolog programing for artificial intelligence*, Addison-Wesley, Harlow 2006.

W Sieci

- *http://www.swi-prolog.org/* – Witryna używanej w artykule implementacji Prologu (en);
- *http://www.learnprolognow.org/* – Kurs programowania w Prologu w wersji online (en);

PIOTR ORZESZEK

Piotr Orzeszek ma wykształcenie filozoficzne. Pracuje jako nauczyciel akademicki na Uniwersytecie Kardynała Stefana Wyszyńskiego w Warszawie. Linuksem interesuje się od ponad dziesięciu lat i używa wspomnianego systemu operacyjnego na co dzień. Kontakt z autorem: piotr@orzeszek.com

Rozwiązanie w Prologu:

```
obok(X, Y, List) :- po_prawej(X, Y, List).
obok(X, Y, List) :- po_prawej(Y, X, List).
po_prawej(L, R, [L | [R | _]]).
po_prawej(L, R, [_ | Rest]) :- po_prawej(L, R, Rest).
ma_zebre(Ulica, Kto) :-
   Ulica = [_Dom1, _Dom2, _Dom3, _Dom4, _Dom5],
   member(dom(czerwony, anglik, _, _, _), Ulica),
   member(dom(_, szwed, pies, _, _), Ulica),
   member(dom(zielony, _, _, kawa, _), Ulica),
   member(dom(_, dunczyk, _, herbata, _), Ulica),
   po_prawej(dom(zielony,_,_,_,_), dom(bialy,_,_,_,_), Ulica),
   member(dom(_, _, slimak, _, old_gold), Ulica),
   member(dom(zolty, _, _, _, kools), Ulica),
   [_, _, dom(_, _, _, mleko, _), _, _] = Ulica,
   [dom(_, norweg, _, _, _) | _] = Ulica,
   obok(dom(_, _, _, _, chesterfields), dom(_, _, lis, _, _), Ulica),
   obok(dom(_, _, _, _, kools), dom(_, _, kon, _, _), Ulica),
   member(dom(_, _, _, sok_pomaranczowy, lucky_strike), Ulica),
   member(dom(_, japonczyk, _, _, parliaments), Ulica),
   obok(dom(_, norweg, _, _, _), dom(niebieski, _, _, _, _), Ulica),
   member(dom(_, Kto, zebra, _, _), Ulica).
```

% Uruchomienie poprzez zapytanie *?- ma_zebre(Ulica,Kto).*

Rozwiązanie w C++ (wersja optymalizowana)

```
//  Original file:  http://weitz.de/files/einstein2.cpp
//  einstein.cpp  (c) Klaus Betzler 20011218  Klaus.Betzler@uos.de

//  Einstein's Riddle, the rules:

//  1 The Brit lives in the red house
//  2 The Swede keeps dogs as pets
//  3 The Dane drinks tea
//  4 The green house is on the left of the white house
//  5 The green house's owner drinks coffee
//  6 The person who smokes Pall Mall rears birds
//  7 The owner of the yellow house smokes Dunhill
//  8 The man living in the centre house drinks milk
//  9 The Norwegian lives in the first house
// 10 The person who smokes Marlboro lives next to the one who keeps cats
// 11 The person who keeps horses lives next to the person who smokes Dunhill
// 12 The person who smokes Winfield drinks beer
// 13 The German smokes Rothmans
// 14 The Norwegian lives next to the blue house
// 15 The person who smokes Marlboro has a neigbor who drinks water

#undef WIN32           // #undef for Linux

#include <stdio.h>
#ifdef WIN32
  #include <windows.h>
#endif

inline unsigned long BIT(unsigned n) {return 1<<n;}

const unsigned long
  yellow = BIT(0),
  blue = BIT(1),
  red = BIT(2),
  green = BIT(3),
  white = BIT(4),

  norwegian = BIT(5),
  dane = BIT(6),
  brit = BIT(7),
  german = BIT(8),
  swede = BIT(9),

  water = BIT(10),
  tea = BIT(11),
  milk = BIT(12),
  coffee = BIT(13),
  beer = BIT(14),

  dunhill = BIT(15),
  marlboro = BIT(16),
  pallmall = BIT(17),
  rothmans = BIT(18),
  winfield = BIT(19),

  cat = BIT(20),
  horse = BIT(21),
  bird = BIT(22),
  fish = BIT(23),
  dog = BIT(24);

const char * Label[] = {
  "Yellow","Blue","Red","Green","White",
  "Norwegian","Dane","Brit","German","Swede",
  "Water","Tea","Milk","Coffee","Beer",
  "Dunhill","Marlboro","Pallmall","Rothmans","Winfield",
  "Cat","Horse","Bird","Fish","Dog"
};
```

```
const unsigned long color = yellow+blue+red+green+white;
const unsigned long country = norwegian+dane+brit+german+swede;
const unsigned long drink = water+tea+milk+coffee+beer;
const unsigned long cigar = dunhill+marlboro+pallmall+rothmans+winfield;
const unsigned long animal = cat+horse+bird+fish+dog;

unsigned long house[5] = {norwegian,blue,milk,0,0};  // rules 8,9,14
unsigned long result[5];

const unsigned long comb[] = { // simple rules
  brit+red,                    // 1
  swede+dog,                   // 2
  dane+tea,                    // 3
  green+coffee,                // 5
  pallmall+bird,               // 6
  yellow+dunhill,              // 7
  winfield+beer,               // 12
  german+rothmans              // 13
};

const unsigned long combmask[] = { // corresponding selection masks
  country+color,
  country+animal,
  country+drink,
  color+drink,
  cigar+animal,
  color+cigar,
  cigar+drink,
  country+cigar
};

inline bool SimpleRule(unsigned nr, unsigned which)
{
  if (which<8) {
    if ((house[nr]&combmask[which])>0)
      return false;
    else {
      house[nr]|=comb[which];
      return true;
    }
  }
  else {           // rule 4
    if ((nr==4)||((house[nr]&green)==0))
      return false;
    else
      if ((house[nr+1]&color)>0)
        return false;
      else {
        house[nr+1]|=white;
        return true;
      }
  }
}

inline void RemoveSimple(unsigned nr, unsigned which)
{
  if (which<8)
    house[nr]&=~comb[which];
  else
    house[nr+1]&=~white;
}

inline bool DunhillRule(unsigned nr, int side)  // 11
{
  if (((side==1)&&(nr==4))||((side==-1)&&(nr==0))||((house[nr]&dunhill)==0))
    return false;
  if ((house[nr+side]&animal)>0)
    return false;
  house[nr+side]|=horse;
  return true;
}
```

```
inline void RemoveDunhill(unsigned nr, unsigned side)
{
  house[nr+side]&=~horse;
}

inline bool MarlboroRule(unsigned nr)    // 10 + 15
{
  if ((house[nr]&cigar)>0)
    return false;
  house[nr]|=marlboro;
  if (nr==0) {
    if ((house[1]&(animal+drink))>0)
      return false;
    else {
      house[1]|=(cat+water);
      return true;
    }
  }
  if (nr==4) {
    if ((house[3]&(animal+drink))>0)
      return false;
    else {
      house[3]|=(cat+water);
      return true;
    }
  }
  int i,k;
  for (i=-1; i<2; i+=2) {
    if ((house[nr+i]&animal)==0) {
      house[nr+i]|=cat;
      for (k=-1; k<2; k+=2) {
        if ((house[nr+k]&drink)==0) {
          house[nr+k]|=water;
          return true;
        }
      }
    }
  }
  return false;
}

void RemoveMarlboro(unsigned m)
{
  house[m]&=~marlboro;
  if (m>0)
    house[m-1]&=~(cat+water);
  if (m<4)
    house[m+1]&=~(cat+water);
}

void Recurse(unsigned recdepth)
{
  unsigned n, m;
  for (n=0; n<5; n++) {
    if (recdepth<9) {    // simple rules
      if (SimpleRule(n, recdepth)) {
        Recurse(recdepth+1);
        RemoveSimple(n, recdepth);
      }
    }
    else {               // Dunhill and Marlboro
      for (int side=-1; side<2; side+=2)
        if (DunhillRule(n, side)) {
          for (m=0; m<5; m++)
            if (MarlboroRule(m))
              for (int r=0; r<5; r++)
                result[r] = house[r];
            else
              RemoveMarlboro(m);
          RemoveDunhill(n, side);
```

```
        }
    }
  }
}

int main()
{
  int index, i;
#ifdef WIN32
  LARGE_INTEGER time0, time1, freq;
  QueryPerformanceCounter(&time0);
#endif
  Recurse(0);
#ifdef WIN32
  QueryPerformanceCounter(&time1);
  QueryPerformanceFrequency(&freq);
  printf("\nComputation Time: %ld microsec\n\n",
    (time1.QuadPart-time0.QuadPart)*1000000/freq.QuadPart);
#endif
  if (result[0]==0) {
    printf("No solution found !?!\n");
    return 1;
    }
  for (i=0; i<5; i++)
    if ((result[i]&animal)==0)
      for (index=0; index<25; index++)
        if (((result[i]&country)>>index)==1)
          printf("Fish Owner is the %s !!!\n\n", Label[index]);
  for (i=0; i<5; i++) {
    printf("%d: ",i+1);
    for (index=0; index<25; index++)
      if (((result[i]>>index)&1)==1)
        printf("%-12s",Label[index]);
    printf("\n\n");
    }
  return 0;
}
```