

Prolog - Programming In Logic

Agnieszka Nowak

28 kwietnia 2008

1 Prolog - Programming In Logic

Należy do klasy języków deklaratywnych, gdzie opisuje się rozwiązywany problem, a nie tak jak przy językach imperatywnych (proceduralnych) - w których wskazuje się algorytm rozwiązania problemu. Dzięki zautomatyzowaniu procesu dowodzenia twierdzeń logicznych, a szczególnie dzięki opracowaniu przez Robinsona w 1965 roku tzw. zasady rezolucji - stało się możliwe automatyczne wywnioskowania rozwiązania na podstawie zbioru formuł logicznych opisujących problem.

Programowanie w prologu składa się z:

1. deklarowania Faktów dotyczących obiektów i związków między nimi.
2. Definiowania Reguł dotyczących obiektów i związków między nimi.
3. Zadawania pytań o obiekty i związki między nimi.

2 Budowanie baz wiedzy w Prologu

Jak wiadomo baza wiedzy to zazwyczaj zbiór złożony z faktów i reguł.

2.1 Fakty proste

Przykładem prostego faktu odzwierciedlającego zdanie: Jest słonecznie. jest predykat zapisany następująco:

sunny.

W języku Prolog spytamy o prawdziwość tego stwierdzenie następująco:

? - sunny.

A więc używamy przed nazwą predykatu znaków ?-, które oznaczają pytanie o prawdziwość danego faktu. Oczywiście stwierdzenia (fakty) nie muszą być jedynie funktorami jednoargumentowymi, ale mogą być również wieloargumentowe. Ogólna składnia predykatu wygląda następująco:

relation(< argument1 >, < argument2 >, ..., < argumentN >).

,gdzie *relation* oznacza po prostu nazwę funkcji pełnionej przez dany predykat,

zaś parametry: `< argument1 >, < argument2 >, ..., < argumentN >` oznaczają kolejne argumenty tej funkcji. I tak, w języku Prolog, stwierdzenie: `likes(john, mary)`. oznacza, że *John* lubi *mary*.

2.2 Zadawanie pytań i interpretacja odpowiedzi na pytania ?

Kiedy już wiemy jak zapisywać stwierdzenia (fakty) zawsze możemy pytać o ich prawdziwość. Jeśli przykładowy program wygląda następująco:

```
eats(fred, oranges). /* 'Fred je pomarańcze' */
eats(tony, apple). /* 'Tony je jabłko' */
eats(john, apple). /* 'John je jabłko' */
```

To jeśli teraz zapytamy:

```
? – eats(fred, oranges).
```

```
/* czy to pasuje do zapisów w bazie wiedzy w Prologu? */
```

Widać, iż tak, więc *Prolog* odpowie:

```
yes
```

```
/* yes, ponieważ dokładnie pokrywa się to z 1 zapisem w bazie. */
```

Podobnie zadając pytanie:

```
? – eats(john, apple).
```

Uzyskamy odpowiedź:

```
yes
```

Zaś jeśli spytamy: `? – eats(mike, apple)`.

prolog odpowie:

```
no
```

```
/* ponieważ nie ma żadnej relacji między mike i apple, gdyż w naszej bazie nie ma wcale informacji o mike */
```

2.3 Zmienne w Prologu

W *prologu* nie musimy zadawać konkretnych pytań. Możemy na przykład zawsze zapytać w naszym przykładowym zbiorze Kto je jabłko? albo Co je Fred ?

Jeśli więc dla naszego przykładowego programu:

```
eats(fred, oranges). Jeśli zapytamy:
```

```
? – eats(fred, what).
```

Prolog odpowie ku naszemu zaskoczeniu:

```
no.
```

Powodem tego jest fakt, że *Prolog* będzie szukał dopasowania w swojej bazie *fred* i *what*. Jeśli natomiast chcieliśmy użyć zmiennej w *Prologu*, to są one tutaj wyróżnione przez fakt, że zaczynają się zawsze od wielkiej litery. Przykładowo:

```
X /* pojedyncza litera */
```

```
Variable /* wyraz zaczynający się od wielkiej litery */
```

```
Two_words /* dwa wyrazy oddzielone znakiem podkreślenia */
```

Zatem w naszym przypadku, gdy chcieliśmy spytać Co je Fred? powinniśmy użyć np składni:

```
? – eats(fred, What)
```

Wówczas *Prolog* odpowie:

What = oranges

yes

Podobnie możemy spytać: Kto je pomarańcze? zapisując owo zapytanie tak:

? – eats(Who, oranges).

W tym wypadku, Prolog odpowie:

Who = fred

yes

Jeśli zaś spytamy:

? – eats(Who, apple).

Pierwszą odpowiedzią będzie: [*Who = tony*] ponieważ jest to pierwsze dopasowanie znalezione w bazie. Teraz Prolog będzie czekał aż wcisniemy jakiś znak. Jeśli wybierzesz znak [enter] Prolog będzie oczekiwał na nowe pytanie. Jeśli zaś użyjesz klawisza [;], Prolog będzie dalej przeszukiwał swoją bazę w celu znalezienia kolejnych dopasowań (podstawień za zmienną *Who*). W ten sposób Prolog znajdzie kolejną odpowiedź [*Who = john*]. Jeśli teraz ponownie wcisniemy znak: [;] prolog ponownie będzie chciał szukać kolejnych dopasowań. Wynikiem jednak będzie odpowiedź: [*no*] ponieważ faktycznie w bazie nie ma więcej dopasowań (unifikacji) dla tego stwierdzenia.

? – eats(Who, apple).

Who = tony;

Who = john;

no

2.4 Reguły w Prologu

Rozważmy zdanie: Wszystkie jabłka to owoce. Takie zdanie możemy wyrazić również w Prologu, następująco:

fruit(X) : –apple(X).

Zdanie to można odczytać jako: X jest owocem jeśli X jest jabłkiem.

Teraz rozważmy zdanie (fakt): Gloster jest typem jabłka.

W Prologu zapiszemy to tak:

fruit(X) : –apple(X).

apple(gloster).

Jeśli teraz spytamy w Prologu:

? – fruit(gloster). Użyjemy odpowiedzi:

yes

Prolog użyje zdefiniowanej przez nas reguły, zgodnie z którą X jest owocem, jeśli jest jabłkiem. Zatem dzięki stwierdzeniu: *apple(gloster)*, Prolog wygeneruje nowy fakt: *fruit(gloster)*. W regułach również możemy stosować zmienne. Zatem możemy spytać: Jaki X jest owocem?:

? – fruit(X).

na co Prolog powinien odpowiedzieć:

X = gloster

Oczywiście reguły mogą mieć w części przesłankowej (warunkowej) więcej elementów połączonych operatorem logicznym *and* lub *or*. Czasami bowiem, do tego samego stwierdzenia możemy dojść różnymi drogami. Przykładowo, zda-

nie: Coś jest smaczne jeśli jest owocem i ma słodki smak lub jeśli zawiera cukier.
To zdanie można następująco zapisać w Prologu:

```
tasty(X) : - /* coś jest smaczne jeśli */  
fruit(X), /* jest owocem i */  
is_sweet(X). /* jest słodkie */  
tasty(X) : - /* albo coś jest smaczne jeśli */  
has_sugar(X) /* zawiera cukier */
```

Zatem w takim programie, są 2 drogi, aby dowiedzieć, się czy coś jest smaczne czy nie. Jeśli pierwsza reguła nie pozwoli wykazać prawdziwości tego stwierdzenia, to wówczas Prolog będzie próbował wykazać prawdziwość drugiej reguły. Należy pamiętać, iż tak samo nazwane zmienne w regule są traktowane jako ta sama zmienna. Zatem takie same zmienne w różnych regułach są traktowane jako różne zmienne i są w tym względzie niezależne. Przykładowo w programie:

```
tasty(X) : -fruit(X), is_sweet(X).  
tasty(X) : -has_sugar(X).
```

Prolog będzie traktował nasze zapisy jako:

```
tasty(X1) : -fruit(X1), is_sweet(X1).  
tasty(X2) : -has_sugar(X2).
```

3 Środowisko do programowania w Prolog'u

Krótkie fakty dotyczące Prolog:

- **Prolog** - stworzony w 1971 roku przez Alaina Colmeraurera i Phillipe'a Roussela na Uniwersytecie w Marsylii.
- Podczas pracy nad zastosowaniem logiki predykatów (klauzul Horna) do NLP.
- Pierwszy kompilator Prologu powstał w Algolu.
- Od połowy lat 70-tych współpraca z Robertem Kowalskim na Uniwersytecie w Edynburgu (Szkocja).

3.1 Implementacje - Narzędzia darmowe

- Ciao Prolog
- ECLIPSE
- **GNU Prolog**
- **SWI-Prolog**
- YAP Prolog



Rysunek 1: Konsola GNU Prolog

3.2 Implementacje - Narzędzia komercyjne

- ALS Prolog
- Amzi! Prolog
- IF Prolog
- LPA Prolog
- MINERVA
- SICStus Prolog

3.3 Czym jest GNU Prolog ?

GNU Prolog (gprolog), to otwarte oprogramowanie pod GNU General Public License.

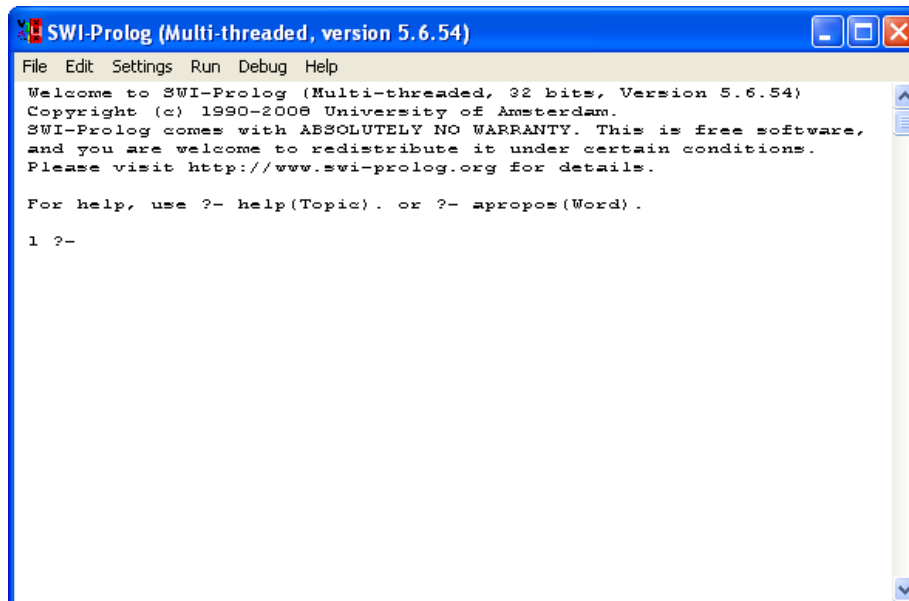
Źródło: [<http://gnu-prolog.inria.fr>].

Jest to narzędzie w pełni darmowe, autorem jest Daniel Diaz, napisane w języku C.

Prócz wbudowanego kompilatora, zawiera także interpreter typu (top-level) i debugger.

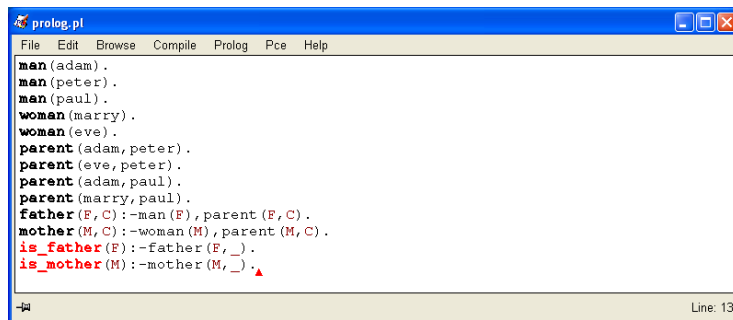
3.4 Czym jest SWI Prolog ?

SWI-Prolog to również samodzielne darmowe środowisko do programowania w Prologu. Źródło: [<http://www.swi-prolog.org>]. Zawiera edytor i program uru-

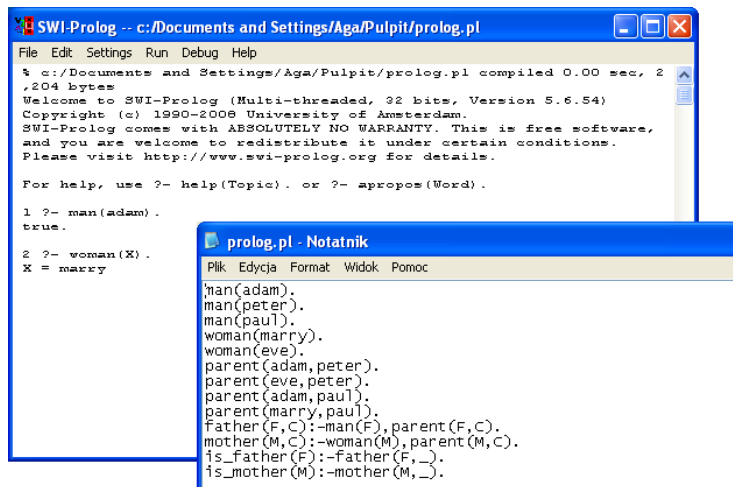


Rysunek 2: Konsola SWI Prolog

chomieniowy. Wraz z narzędziem *XPCE* pozwala na edycję programów w Prologu nie w notatniku, a właśnie w edytorze *XPCE*, pozwalającym m.in. na kolorowanie składni programu.



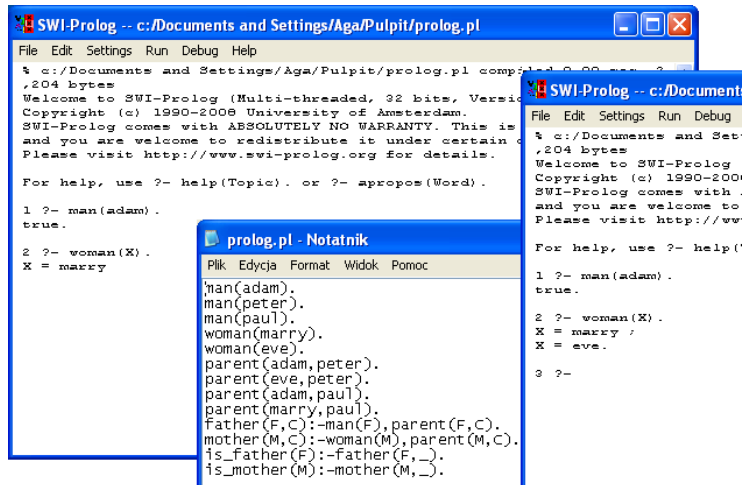
Rysunek 3: Edytor XPCE dla SWI Prolog



4 Prolog w praktyce

4.1 Przykładowa baza: [prolog.pl]

4.2 Wnioskowanie na bazie [prolog.pl]





The image shows a window titled "SWI-Prolog -- c:/Documents and Settings/Agar/Pulpit/prolog.pl". The menu bar includes File, Edit, Settings, Run, Debug, and Help. The text area contains the following content:

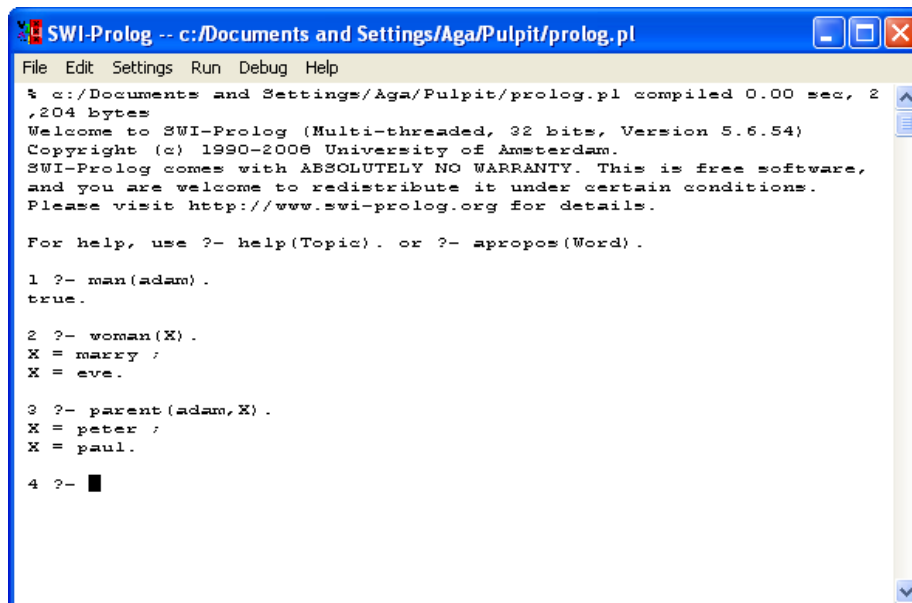
```
% c:/Documents and Settings/Agar/Pulpit/prolog.pl compiled 0.00 sec, 2,204 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.54)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic) . or ?- apropos(Word) .

1 ?- man(adam) .
true.

2 ?- woman(X) .
X = marry ;
X = eve.

3 ?- parent(adam,X) .
X = peter
```



The image shows the same SWI-Prolog window as above, but with additional queries and results:

```
% c:/Documents and Settings/Agar/Pulpit/prolog.pl compiled 0.00 sec, 2,204 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.54)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

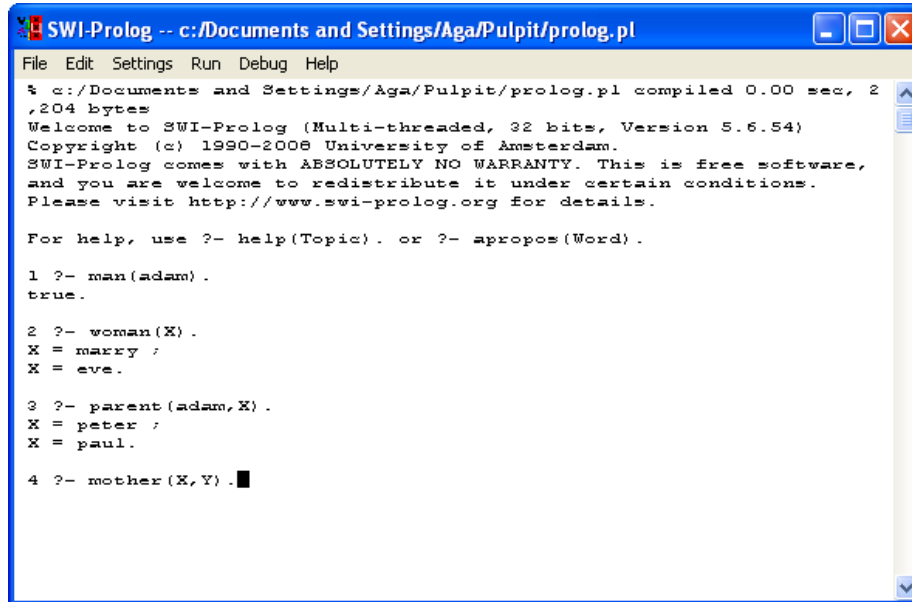
For help, use ?- help(Topic) . or ?- apropos(Word) .

1 ?- man(adam) .
true.

2 ?- woman(X) .
X = marry ;
X = eve.

3 ?- parent(adam,X) .
X = peter ;
X = paul.

4 ?- █
```



```
SWI-Prolog -- c:/Documents and Settings/Aga/Pulpit/prolog.pl
File Edit Settings Run Debug Help
% c:/Documents and Settings/Aga/Pulpit/prolog.pl compiled 0.00 sec, 2
,204 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.54)
Copyright (c) 1990-2006 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

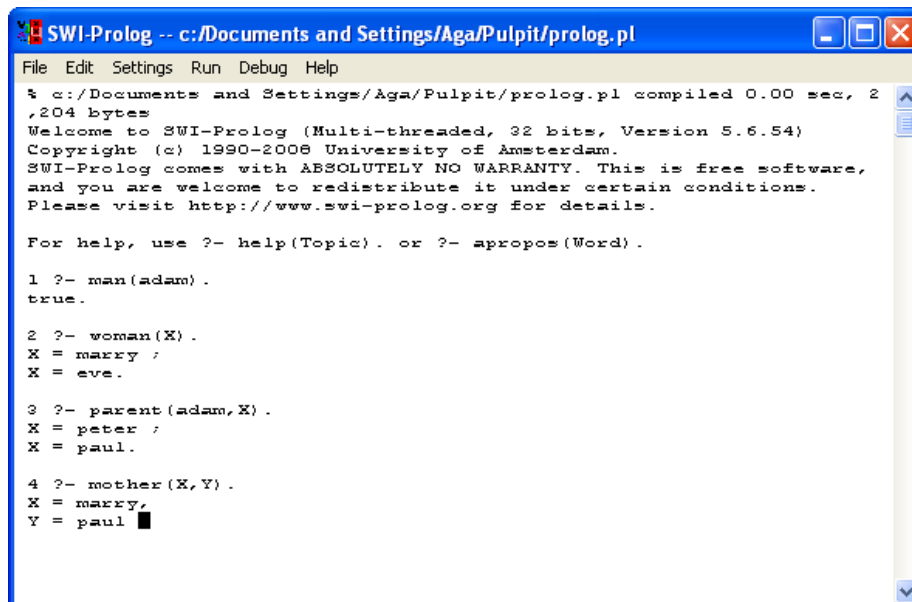
For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- man(adam) .
true.

2 ?- woman(X) .
X = marry ;
X = eve.

3 ?- parent(adam,X) .
X = peter ;
X = paul.

4 ?- mother(X,Y) .
```



```
SWI-Prolog -- c:/Documents and Settings/Aga/Pulpit/prolog.pl
File Edit Settings Run Debug Help
% c:/Documents and Settings/Aga/Pulpit/prolog.pl compiled 0.00 sec, 2
,204 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.54)
Copyright (c) 1990-2006 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- man(adam) .
true.

2 ?- woman(X) .
X = marry ;
X = eve.

3 ?- parent(adam,X) .
X = peter ;
X = paul.

4 ?- mother(X,Y) .
X = marry,
Y = paul
```



```
SWI-Prolog -- c:/Documents and Settings/Ag/Pulpit/prolog.pl
File Edit Settings Run Debug Help
% c:/Documents and Settings/Ag/Pulpit/prolog.pl compiled 0.00 sec, 2
,204 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.6.54)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- man(adam) .
true.

2 ?- woman(X) .
X = marry ;
X = eve.

3 ?- parent(adam,X) .
X = peter ;
X = paul.

4 ?- mother(X,Y) .
X = marry,
Y = paul ;
X = eve,
Y = peter.

5 ?- █
```

5 Prolog w przykładach

5.1 Przykład nr 1: wprowadzenie do Prologu

Przykład nr 1:

Lubi(Jarek, ryby)
Lubi(Jarek, Maria)
Lubi(Maria, ksiazka)
Lubi(Jan, ksiazka)
Lubi(Jan, Francja)

Teraz gdybyśmy chcieli wywnioskować z tych faktów czy *Jarek* lubi *pieniadze*:

? – *lubi(Jarek, pieniadze)*

Odp Prologu: *no*

? – *lubi(Maria, Jarek)*

Odp Prologu: *no*

? – *lubi(Jarek, Maria)*

Odp Prologu: *yes*

Przykład :

Lubi(Jan, kwiaty)
Lubi(Jan, Maria)
Lubi(Pawel, Maria)

Teraz gdybyśmy chcieli wywnioskować z tych faktów co lubi Jan?

PYTANIE:

? – *lubi(Jan, X)*

Odp Prologu:

$X = \textit{kwiaty}$

(zmienna X jest odtąd UKONKRETNIONA !)

PYTANIE:

? – $\textit{lubi}(x, \textit{Maria})$

Odp Prologu:

$X = \textit{Jan};$

$X = \textit{Pawel};$

$\textit{No};$

co oznacza że nie ma już więcej odpowiedzi możliwych !

KONIUNKCJE:

Przykład :

$\textit{Lubi}(\textit{Maria}, \textit{czekolada})$

$\textit{Lubi}(\textit{Maria}, \textit{wino})$

$\textit{Lubi}(\textit{Jan}, \textit{wino})$

$\textit{Lubi}(\textit{Jan}, \textit{Maria})$

Wówczas:

Pytanie:

? – $\textit{lubi}(\textit{Jan}, \textit{Maria}), \textit{lubi}(\textit{Maria}, \textit{Jan})$

Odp Prologu: $\textit{no};$

Pytanie:

? – $\textit{lubi}(\textit{Maria}, X), \textit{lubi}(\textit{Jan}, X)$

czyli czy istnieje coś co jednocześnie lubią i Maria i Jan

Odp Prologu:

$X = \textit{wino};$

$\textit{no};$

REGUŁY:

Jan lubi każdego kto lubi wino

$\textit{Lubi}(\textit{Jan}, X) : \neg \textit{Lubi}(X, \textit{wino}).$

Jan lubi kobiety, które lubią wino

$\textit{Lubi}(\textit{Jan}, X) : \neg \textit{Kobieta}(X), \textit{Lubi}(X, \textit{wino}).$

5.2 Przykład nr 2: "Rozkład lotów"

2 predykaty są dane:

$\textit{Rejsy}(\textit{skąd}, \textit{dokąd}, \textit{odlot}, \textit{przylot})$

(wyraża dostępne loty między miastami USA)

$\textit{Połączenie}(\textit{skąd}, \textit{dokąd}, \textit{odlot}, \textit{przylot})$

(znajduje połączenia pośrednie i bezpośrednie, ale na każdą przesiadkę rezerwuje minimum godzinę między przylotem a odlotem.)

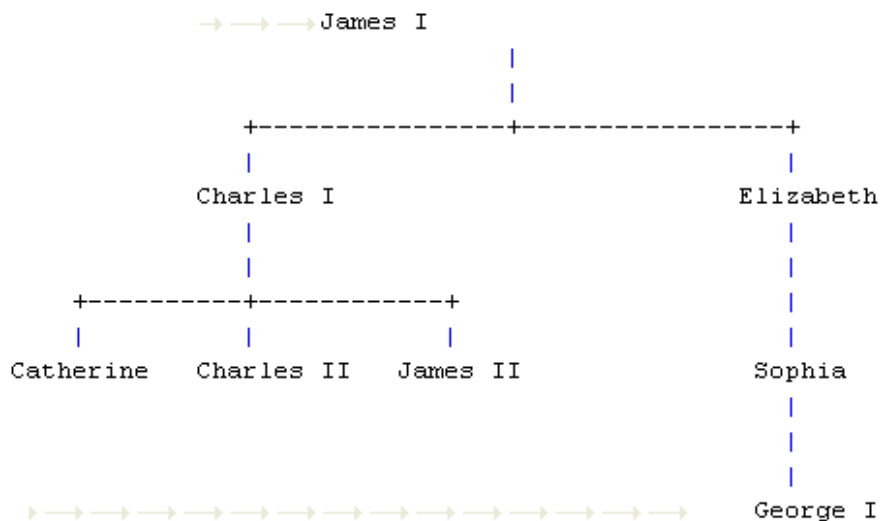
Dane są:

$\textit{Rejsy}(\textit{SF}, \textit{DEN}, 930, 1230)$

$\textit{Rejsy}(\textit{SF}, \textit{DAL}, 900, 1430)$

$\textit{Rejsy}(\textit{DEN}, \textit{CHI}, 1500, 1800)$

$\textit{Rejsy}(\textit{DEN}, \textit{DAL}, 1400, 1700)$



Rejsy(DAL, CHI, 1530, 1730)

Rejsy(CHI, NY, 1500, 1930)

Rejsy(CHI, NY, 1900, 2200)

Rejsy(CHI, NY, 1830, 2130)

Polaczenie(X, Y, O, P) : -rejsy(X, Y, O, P)

Polaczenie(X, Y, O, P) : -rejsy(X, Z, O, T1), Polaczenie(Z, Y, T2, P), T2 >= T1 + 100.

Teraz w celu znalezienia połączeń z San Francisco (SF) do Chicago (CHI) zadajemy następujące pytanie:

? - Polaczenie(SF, CHI, Odlot, Przylot).

A prolog odpowiada:

Odlot = 930, Przylot = 1800;

Odlot = 900, Przylot = 1730;

No.

Natomiast jeśli chcemy wylecieć z SF do NY po godzinie 900, to możemy znaleźć odpowiednie połączenie zadając pytanie:

? - polaczenie(SF, NY, Odlot, Przylot), Odlot > 900.

Wówczas Prolog odpowie:

Odlot = 930, Przylot = 2200;

No.

5.3 Przykład nr 3 "Windsor"

W Prolog'u zapiszemy to następująco:

male(james1).

male(charles1).

male(charles2).

```

male(james2).
male(george1).
/* female(P) is true when P is female */
female(catherine).
female(elizabeth).
female(sophia).
/* parent(C,P) is true when C has a parent called P */
parent(charles1, james1).
parent(elizabeth, james1).
parent(charles2, charles1).
parent(catherine, charles1).
parent(james2, charles1).
parent(sophia, elizabeth).
parent(george1, sophia).

```

Teraz możliwe jest zdefiniowanie kilku pytań:

Was George I the parent of Charles I?
parent(charles1, george1).

Who was Charles I's parent?
parent(charles1, Parent).

who were the children of Charles I?
parent(Child, charles1).

5.4 Przykład nr 4 - "student"

Przypuśćmy, że mamy następujące zdania w języku PROLOG:

```

teaches(dr_fred, history)..
teaches(dr_fred, english)..
teaches(dr_fred, drama).
teaches(dr_fiona, physics).
studies(alice, english).
studies(angus, english).
studies(amelia, drama).
studies(alex, physics).

```

Roważmy następujące pytania i odpowiedzi na nie:

? – teaches(dr_fred, Course), studies(Student, Course).

odpowiedź:

```

Course = english
Student = alice ;
Course = english
Student = angus ;
Course = drama
Student = amelia ;
No

```

5.5 Przykład nr 5 - "Windsor cd."

Rozważmy zdania:

born(charles, elizabeth2, philip).

born(anne, elizabeth2, philip).

born(andrew, elizabeth2, philip).

born(edward, elizabeth2, philip).

born(diana, frances, edwardSpencer).

born(william, diana, charles).

born(henry, diana, charles).

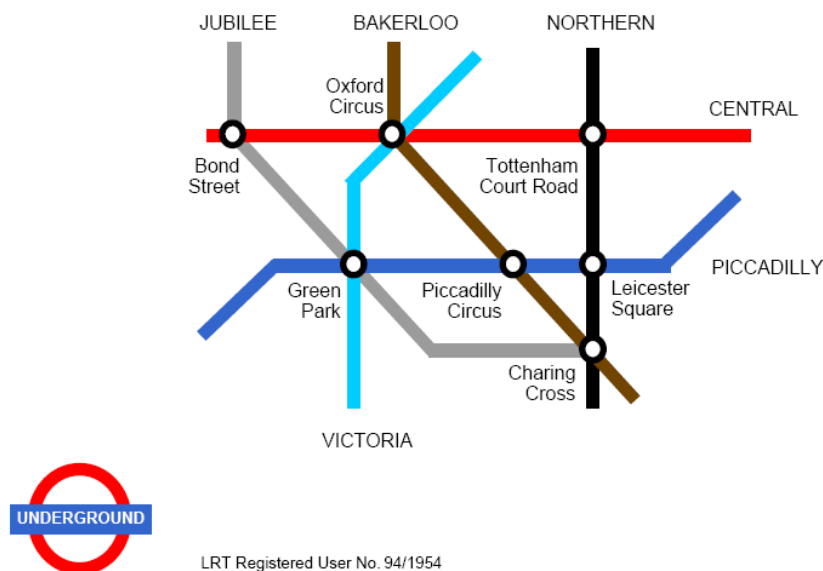
Rozważmy więc zapytanie:

? – *born(S, elizabeth2, Y) and born(G, M, S).*

Jaka będzie odpowiedź?

5.6 Przykład nr 6 - "Metro w Londynie"

Schemat metra jest zamieszczony na rysunku.



Zaś baza zapisana w Prologu wygląda następująco:

connected(bond_street, oxford_circus, central).

connected(oxford_circus, tottenham_court_road, central).

connected(bond_street, green_park, jubilee).

connected(green_park, charing_cross, jubilee).

connected(green_park, piccadilly_circus, piccadilly).

connected(piccadilly_circus, leicester_square, piccadilly).

connected(green_park,oxford_circus,victoria).
connected(oxford_circus,piccadilly_circus,bakerloo).
connected(piccadilly_circus,charging_cross,bakerloo).
connected(tottenham_court_road,leicester_square,northern).
connected(leicester_square,charging_cross,northern).
nearby(bond_street,oxford_circus).
nearby(oxford_circus,tottenham_court_road).
nearby(bond_street,tottenham_court_road).
nearby(bond_street,green_park).
nearby(green_park,charging_cross).
nearby(bond_street,charging_cross).
nearby(green_park,piccadilly_circus).
nearby(piccadilly_circus,leicester_square).
nearby(green_park,leicester_square).
nearby(green_park,oxford_circus).
nearby(oxford_circus,piccadilly_circus).
nearby(piccadilly_circus,charging_cross).
nearby(oxford_circus,charging_cross).
nearby(tottenham_court_road,leicester_square).
nearby(leicester_square,charging_cross).
nearby(tottenham_court_road,charging_cross).

Wykonaj teraz zapytania:

- ? – *nearby(tottenham_court_road, W)*
oraz
- ? – *connected(tottenham_court_road, W, L).*

Czy zapisy:

nearby(X, Y) : –connected(X, Y, L).

nearby(X, Y) : –connected(X, Z, L), connected(Z, Y, L).

dokładnie odpowiadają zawartości całej bazy ?

5.7 Ćwiczenia

Założmy, że przykładowa baza wygląda następująco:

- *incumbent(csp Professor, davis).*
- *incumbent(csp Professor, rowe).*
- *incumbent(csp Professor, wu).*
- *incumbent(csp Professor, zyda).*
- *incumbent(cschairman, lum).*
- *incumbent(dean_ips, marshall).*
- *incumbent(provost, shrad y).*

- *incumbent(superintendent, shumaker).*
- *incumbent(director_milops, bley).*
- *bossed_by(csprofessor, cschairman).*
- *bossed_by(cschairman, dean_ips).*
- *bossed_by(orchairman, dean_ips).*
- *bossed_by(dean_ips, provost).*
- *bossed_by(provost, superintendent).*
- *bossed_by(director_milops, superintendent).*

gdzie *incumbent* oznacza, że osoba będąca drugim argumentem ma pracę opisaną jako pierwszy argument. *bossed_by* zaś oznacza, że szefem (*boss*) osoby będącej pierwszym argumentem, jest osoba będąca drugim argumentem. predicate means that the boss of the first argument is the second argument.

Sformułuj poniższe pytania w języku angielskim:

- ? – *bossed_by(csprofessor, X), bossed_by(X, Y).*
- ? – *bossed_by(X, Y), incumbent(X, rowe), incumbent(Y, Z).*
- ? – *incumbent(dean_ip, X); incumbent(dean_ips, X).*
- ? – *incumbent(J, P), (bossed_by(J, provost); bossed_by(J, dean_ips)).*
- ? – *bossed_by(P, superintendent), not(incumbent(P, shrady)).*

6 Dodatki

6.1 Wybór reguł i faktów - kolejność reguł i faktów w bazie

Reguły mogą być składowymi bazy Prologowej tak samo jak fakty. Wówczas interpreter wykorzystuje zarówno reguły jak i fakty do odpowiedzi na pytania. Ważna jest jednak kolejność (porządek) obu tych zbiorów. Aby to uzasadnić, rozważmy kilka różnych kombinacji metod rozumowania (wnioskowania) dla tej samej reguły, trzema różnymi drogami ustalającymi kolor czegoś:

*color_object(X, C) : -color(X, C); (part_of(X, Y), color(Y, C));
(part_of(X, Y), part_of(Y, Z), color(Z, C)).*

Oczywiście nie jest problemem zajęcie przez dane pytanie czy regułę więcej niż jednej linii zapisu. Długie zapisy są jednak mało czytelne, dlatego też dla złożonych reguł lepiej jest stosować kilka oddzielnych reguł (3) z taką samą formą lewej ich strony:

color_object(X, C) : -color(X, C).

color_object(X, C) : -part_of(X, Y), color(Y, C).

color_object(X, C) : -part_of(X, Y), part_of(Y, Z), color(Z, C). Teraz prawa strona każdej reguły (tzw. warunki, przesłanki reguły) przedstawie odpowiednie warunki do tego aby lewa strona reguły została uznana za prawdziwą. Każda z reguł przedstawia inną drogę ustalenia koloru jakiegoś obiektu, ale nie wszystkie reguły muszą być spełnione. Innymi słowy: nie dla wszystkich przypadków *color_object* musi być zostać pokryte faktami w bazie. Oczywiście ma znaczenie kolejność tych reguł względem siebie. Zawsze, gdy pytanie dotyczące predykatu *color_object* z dwoma argumentami zostanie zadane, reguły będą usiłowały odpowiedzieć na pytanie, przeszukując kolejno bazę. Porządek przedstawiony tutaj zdaje się być optymalnym, gdyż jako pierwsza zostanie uaktywniona najprostsza reguła, potem dopiero kolejne, bardziej złożone. Oczywiście fakty powinny być umieszczone w bazie przez regułami. Dzieje się tak dlatego, iż fakty zwykle wymagają mniej zadania dopasowywania niż reguły.

6.2 Nawroty w regułach

Nawroty są stosowane zawsze wtedy, gdy następuje próba dopasowania poszczególnych wyrażeń w bazie do pytania. Jeśli predykat w pytaniu ma regułę, wówczas interpreter zachowuje się tak, jakby prawa strona definicji była wprowadzona jak część pytania zamiast pojedynczego predykatu z lewej strony. Stosowanie nawrotów oznacza powtórne wykonywanie tych samych reguł. Jest to proces trudny w wykonaniu, gdyż wiąże się z wielokrotnym przesuwaniem w lewą lub prawą stronę dowodu. Często też powoduje skoki w dół lub górę drzewa wywodu. To jest niewątpliwie spory problem w sensie mocy obliczeniowej takich języków jak Prolog. Rozważmy jednak następujący przykład:

Zakładając, że mamy dwa rodzaje faktów dotyczących organizacji: wydział zatrudniający pracowników, i zarządzający tym wydziałem. Założmy, że definiujemy szefa używając dwóch argumentów: *B* oraz *E*, które mówią, że *B* jest szefem *E* jeśli *B* zarządza wydziałem *D* w którym *E* jest zatrudniony. *D* będzie

zmienną lokalną. Zakładamy też, że *Tom* pracuje w wydziale sprzedaży, *Harry* w wydziale produkcji, zaś *Dick* zarządza wydziałem sprzedaży a *Mary* zarządza wydziałem produkcji. Wówczas w baza w Prologu ujmującą tą wiedzę wyglądałaby następująco:

department(tom, sales).

department(harry, production).

manager(dick, sales).

manager(mary, production).

boss(B, E) : -department(E, D), manager(B, D).

Teraz, załóżmy, że chcemy znaleźć szefa innego niż *Tom*. Stworzymy więc pytanie ze zmienną *X*:

? - boss(X, Y), not(boss(X, tom)).

Jak widać, nie ma faktów w bazie pasujących wprost do pytania. Jest natomiast reguła. Należy zatem znaleźć dopasowanie pierwszego wyrażenia prawej strony reguły: *department(E, D)*.

Pasuje to do pierwszego faktu, przez co *E = tom* i *D = sales*. Teraz, przesuwając się do drugiego wyrażenia w regule: *manager(B, D)*, interpreter znajdzie dopasowanie trzeciego faktu w bazie: *B = dick*, przez co cała reguła zostanie pomyślnie uaktywniona, i pierwsze wyrażenie w pytaniu zostanie spełnione, przez co uzyskamy odpowiedź: *X = dick* i *Y = tom*.

Teraz, przesuwając się do drugiego (ostatniego) wyrażenia w pytaniu: *not(boss(X, tom))*, interpreter próbuje sprawdzić, czy *Dick* jest szefem (*boss*) *Toma*.

Nie ma jednak żadnych zapisów tego dotyczących więc następuje powrót do reguły.

Ponownie, oba predykaty prawej strony reguły mogą być dopasowywane do faktów, przez co reguła zostanie uaktywniona. Ale warunek w pytaniu oryginalnym jest negacją (zaprzeczeniem) tego, przez co druga część oryginalnego pytania zawodzi (kończy się porażką), i interpreter wraca do pierwszego wyrażenia. Nawroty kończą się sukcesem tylko wtedy, gdy poprzednio znaleziono jakieś nowe dopasowania. Skoro więc nie dodano żadnego nowego szefa w dziale sprzedaży, musimy więc wrócić do pierwszej części reguły: *department(E, D)* z nie zadanymi *B* i *E*.

Na szczęście mamy inny wybór: *E = harry* i *D = production*.

Teraz interpreter może powtórnie uruchomić prawą stronę reguły. Polegać to będzie na ponownym rozważaniu reguły. Teraz możemy znaleźć dopasowanie *B = mary* (pamiętając że teraz *D = production*). Reguła zostanie uaktywniona, a pierwsze wyrażenie wyprowadzi: *X = mary* i *Y = harry*.

W drugiej części pytania, musimy sprawdzić, czy nieprawdą jest, że *Mary* jest szefem *Toma*? Aby tego dokonać, musimy próbować sprawdzić, czy prawdziwa jest reguła *boss* dla *B = mary* i *Toma* jako drugiego argumentu. Z pierwszej części prawej strony reguły, możemy wywnioskować, że można dopasować *D* do sprzedaży, ale nie ma faktu, że *Mary* zarządza tym działem. Dlatego reguła zakończy się porażką, a że nie ma więcej reguł na *boss*, cały dowód zakończy się porażką. Jednak odkąd mamy negację z przodu wyrażenia, całe pytanie pasuje, więc: *X = mary* jest odpowiedzią, której szukaliśmy.

6.3 Przechodność wnioskowań

Formy pewnych reguł szczególnie często pojawiają się w sztucznej inteligencji. Bardzo ważną formę określa przechodność predykatów dwuargumentowych. Dla przykładu rozważmy przykład "szefów" w organizacji. Jeśli Twój szef ma szefa kolejnego, ten duży szef jest również Twoim szefem. Jeśli i ten szef ma jakiegoś szefa, to nawet ten jest po części i Twoim szefem. Stąd relacja szefostwa tworzy łańcuchy, i to nazywamy właśnie przechodnością.

Formalnie, predykat relacji r jest przechodni jeśli następująca reguła jest poprawna:

$r(X, Y) : \neg r(X, Z), r(Z, Y).$

Oznacza to mniej więcej, że jeśli predykat r działa od jakiegoś X do jakiegoś Z , i również od Z do jakiegoś Y , wówczas mówimy, że predykat ten również przechodzi od jakiegoś X do Y . Można tej reguły używać również rekurencyjnie, również dla reguł a nie tylko faktów. Dla przykładu, rozważmy fakty przed regułami w bazie Prolog:

$r(a, b).$

$r(b, c).$

$r(c, d).$

Teraz, jeśli zadamy następujące pytanie:

$? - r(a, d),$

na które nie ma dopasowania, interpreter użyje reguł, i najpierw zapyta:

$? - r(a, Z).$

Teraz, Z pasuje do b w pierwszym fakcie. Interpreter zapyta więc kolejno:

$? - r(b, d).$

Na to nie ma niestety żadnego dopasowania, stąd reguła musi być użyta powtórnie. Dla nowego wywołania reguły: $X = b$ i $Y = d$, zatem kolejne pytanie będzie miało formę:

$? - r(b, Z).$

To nowe Z różni się od poprzedniego Z , odkąd każda powtórne wywołanie posiada własne zmienne. Teraz interpreter potrafi dopasować Z do c ponieważ istnieje fakt $r(b, c)$, i teraz druga część reguły staje się pytaniem: $? - r(c, d).$

To jest fakt w bazie. Stąd reguła została pomyślnie uaktywniona i dowiodła $r(b, d)$ - a więc oryginalne pytanie. Wiele relacji jest przechodnich: *a_kind_of*, *part_of*, *right_of*. Przykłady:

- If the Vinson is a kind of carrier, and the carrier is a kind of ship, then the Vinson is a kind of ship.
- If the electrical system is part of the car, and the battery is part of the electrical system, then the battery is part of the car.
- If the Vinson is north of the Enterprise, and the Enterprise is north of the Kennedy, then the Vinson is north of the Kennedy.
- If during the day Monday you had a meeting with your boss, and during the meeting you found out you got promoted, then during the day Monday

you found out you got promoted.

- If a number X is greater than a number Y , and Y is greater than a number Z , then X is greater than Z .

6.3.1 Ćwiczenia: klocki

Rozważmy przykład faktów dotyczących sterty klocków na stole. Kłosek jest nad innym klockiem jeśli na nim spoczywa. Kłosek b jest nad klockiem a a kłosek c nad klockiem b , zaś kłosek d nad klockiem c . Przez relację przechodniości powiemy, że kłosek d jest nad klockiem a .

6.4 Dziedziczenie wnioskowań

Dziedziczenie jest jeszcze bardziej istotne niż przechodniość. Rozważmy przykład organizacji biurowej. Jeśli ma ona tylko jeden adres biura, jest to wówczas adres wszystkich zatrudnionych w tym biurze. Nie ma więc sensu dla komputera trzymanie oddzielnie adresów (faktów o adresie) dla każdego pracownika. Zamiast tego bardziej wskazane jest pamiętanie jednego adresu z nazwą organizacji. Teraz adres dziedziczony jest z organizacji na pracowników.

Dziedziczenie zawsze wymaga dwóch predykatów, właściwości i relacji. Formalnie powiemy, że właściwość predykatu p dziedziczy w odniesieniu do relacji predykatu r jeśli ta reguła jest poprawna:

$p(X, Value) : \neg r(X, Y), p(Y, Value)$.

Zatem, potrafimy wykazać, że właściwość p dla obiektu X posiada wartość $Value$ jeśli potrafimy wykazać, że Y jest związany z X predykatem r , i, że Y posiada wartość $Value$ dla właściwości p . To uogólnia regułę dla predykatu *color_object*.

6.4.1 Ćwiczenia: drzewo genealogiczne

Napisz reguły dla wnioskowania o genealogii czy drzewie rodzinnym. Załóżmy, że genealogia jest prezentowana za pomocą faktów w jednej z czterech form: *child(< name_of_father >, < name_of_mother >, < name_of_child >, < sex >)*.

Zdefiniuj następujące predykaty:

- *father*(X, Y), znaczy, że X jest ojcem Y
- *mother*(X, Y), znaczy, że X jest matką Y
- *son*(X, Y), znaczy, że X jest synem Y
- *grandfather*(X, Y), znaczy, że X jest dziadkiem Y
- *sister*(X, Y), znaczy, że X jest siostrą Y
- *uncle*(X, Y), znaczy, że X jest wujkiem Y
- *ancestor*(X, Y), znaczy, że X jest przodkiem Y
- *half_sister*(X, Y), znaczy, że X jest siostrą przyrodnią Y

7 Literatura

- Pawlak Z., (1983) Information Systems - theoretical foundations [polish], WNT, W-wa.
- Pogorzelski W.A., (1973), Klasyczny rachunek zdań. Zarys teorii, PWN, Warszawa, Poland
- Cholewa W., Pedrycz W., (1987), Systemy doradcze, skrypt Politechniki Śląskiej nr 1447, Gliwice, Poland
- Cichosz P.,(2001), Systemy uczące się, WNT, Warszawa, Poland
- Grzegorzczak A., (1969), Zarys logiki matematycznej, PWN, Warszawa, Poland
- Paprzycka K., Samouczek logiki zdań i logiki kwantyfikatorów - dostępny na stronie: <http://www.filozofia.uw.edu.pl/kpaprzycka/Publ/xSamouczek.html>
- Nilsson U., Małuszyński J.,(2000), Logic, programming and Prolog (2ED), <http://www.ida.liu.se/~ulfni/lpp>
- Flach P., (1994), University of Bristol, United Kingdom Simply Logical - Intelligent Reasoning by Example