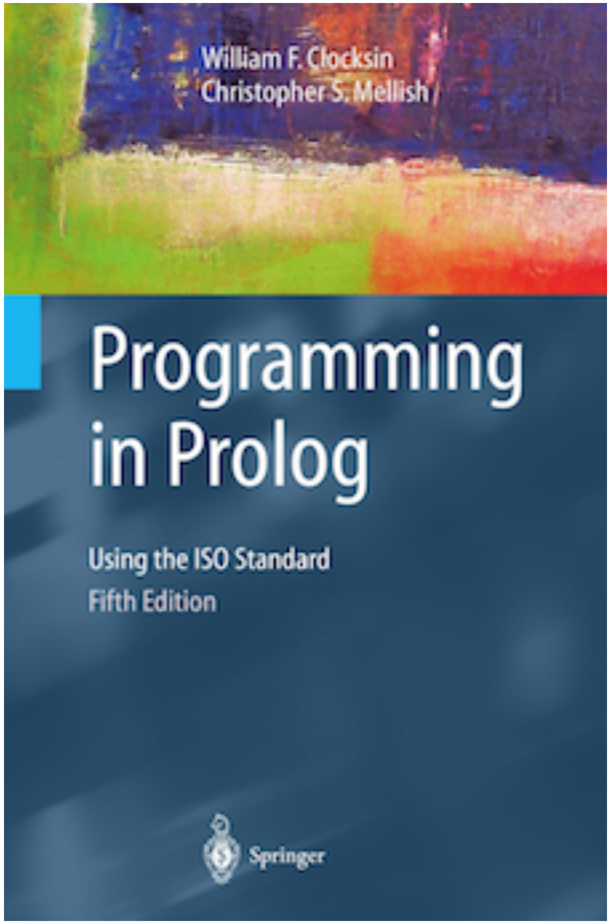
PARADYGMATY I JĘZYKI PROGRAMOWANIA

Programowanie w Logice – Prolog (w12)

Książka (Springer)

Clocksin, Mellish: Programming In Prolog. Springer Verlag, 2003.,



Treść

- Programowanie w logice – wstęp
- Rachunek predykatów – wstęp
- Prolog
  - Historia
  - Proste programy w j. prolog – przykłady
  - Praca z listami
  - Wyrażenia arytmetyczne
  - Operatory
  - Metoda wycofywania – backtracking
  - Rekurencja

Programowanie w Prologu

- Specyfikacja *faktów* o *obiektach* i *związkach* (relacjach) między obiektami
- Definiowanie *reguł* miedzy obiektami i związkami między obiektami
- Zadawanie *pytań* o obiekty i związki między nimi

Przykład programu


```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
```

FAKTY

```
sister_of(X, Y) :-
female(X),
parents(X, M, F),
parents(Y, M, F).
sister_of(X, Y) :- female(X), parents(X, M, F), parents(Y, M, F).
```

REGUŁY

```
% PYTANIA
?- sister_of(alice, edward).    -> yes.
?- sister_of(alice, X).         -> X = edward.
```

Programowanie w logice

- Programowanie w logIce – programowanie deklaratywne.
  - Programowanie polega na wypisaniu stwierdzeń z użyciem rachunku *predykatów* pierwszego rzędu.
  - Wyniki otrzymuje się wg. ogólnych reguł wnioskowania. Programowanie nie polega, jak w innych językach, na przypisaniach i sterowaniu przepływem obliczeń, lecz na *deklaracjach*.
  - Nie podaje się reguł obliczeń, lecz opis rozwiązania.
- Podstawowe pojęcia
  - *term* – funktor (symbol) wraz z listą parametrów, atomów
  - *stała* – term bezparametrowy, np. narty, morze
  - *stwierdzenie* – jeden lub wiele termów połączonych spójnikami
    ~ negacja, ∨ alternatywa, ∧ koniunkcja, = równoważność i => implikacja
  - *zmienne* - stwierdzenia mogą zawierać zmienne (X, Y, Co,To, ...) *związane* przez kwantyfikatory (istnieje) i (dla każdego)
    Zgodność stwierdzeń z podanymi aksjomatami jest sprawdzana przez daną implementację języka.

Rachunek predykatów

- *Funktor*
  - Wyrażenie nie będące zdaniem ani nazwą, służące do konstruowania zdań lub nazw lub innych funktorów.
  - Jeśli *funktor* wraz ze swoimi argumentami tworzy zdanie wówczas nazywa się go *funktorem zdaniotwórczym* („*grzeje mocno”* jest f.z. ponieważ po dołączeniu do niego wyrazu np. „*Słońce”* otrzymamy zdanie „*Słońce grzeje mocno”*); np. *„i”* w wyrażeniu „*Słońce i Księżyc*”
- *Predykat*
  - Funktor zdaniotwórczy (np. słowo *grzeje* w argumentach nazwowych *„słońce grzeje”, „x grzeje”*);
  - Wyrażenie złożone z funktora zdaniotwórczego od argumentów nazwowych; funkcja zdaniowa
  - Wyrażenie, które opisuje pewną własność lub relację
- Dział logiki dotyczący predykatów nazywa się *rachunkiem predykatów* lub *rachunkiem kwantyfikatorów*

Rachunek predykatów

- *Kwantyfikatory* – symbole określające ilość (łac. *quantum*; wszystkie przedmioty, obiekty; niektóre przedmioty, jeden obiekt)
  - k. uniwersalny, duży: $\forall$, $\wedge$; $\forall x F(x)$
  - k. egzystencjalny, szczegółowy: $\exists$, $\vee$; $\exists x F(x)$
- *Rachunek predykatów* = rachunek kwantyfikatorów, rachunek funkcyjny;
  - *rachunek pierwszego rzędu* – kwantyfikatory wiążą tylko zmienne indywiduowe.

Stwierdzenia

- Indywidua reprezentowane są przez symbole (termy), które są
  - *stałymi*, reprezentują konkretne obiekty
  - *zmiennymi* – symbolami, które mogą reprezentować różne obiekty, indywidua
- Z prostych stwierdzeń *atomowych*, tworzy się termy złożone – relacje matematyczne, zapisane w postaci funkcji
  - funkcja jest odwzorowaniem
  - może być zapisana w postaci tablicy

Termy złożone

- Termy złożone są zbudowane z dwóch części
  - funktora – symbolu funkcji określającej relację
  - ciągu argumentów

- Przykłady

```
student(jan)
lubi(jan, osx)
lubi(ala, windows)
lubi(X, linux)
```

Postać stwierdzń

- Stwierdzenia można zapisać jako
  - *fakty* – zakłada się wtedy o nich, że są prawdziwe
  - *zapytania* – należy udowodnić, że są prawdziwe
- Stwierdzenia złożone
  - składają się z jednego lub więcej stwierdzeń atomowych
  - stwierdzenia połączone są operatorami

Operatory i kwantyfikatory RP

Symbole logiczne (operatory)

NAZWA	SYMBOL	PRZYKŁAD	ZNACZENIE
negacja	$\neg$	$\neg a$	nie **a**
koniunkcja	$\wedge$	$a \wedge b$	**a** i **b**
alternatywa	$\vee$	$a \vee b$	**a** lub **b**
równoważność	$\equiv$	$a \equiv b$	**a** jest równoważne **b**
implikacja	$\Rightarrow$	$a \Rightarrow b$	**a** implikuje **b**

Kwantyfikatory

- uniwersalny, duży: $\forall$, $\bigwedge$
- szczegółowy, egzystencjalny: $\exists$, $\bigvee$

Przykłady

- Stwierdzenia złożone
  $a \wedge b \Rightarrow c$
  $a \wedge \neg b \Rightarrow d$
- Kwantyfikatory
  $\forall$**XP** – dla wszystkich X, P jest prawdą
  $\exists$**XP** – istnieje X takie, że P jest prawdą

  $\forall$`X(kot(X) ⇒ zwierze(X))`
  $\exists$`X(matka(alicja, X) ⇒ chłopiec(X))`

  Pierwsze z powyższych stwierdzeń:
  *Dla każdego **X**, jeżeli **X** jest kotem to **X** jest zwierzęciem.*

  Drugie stwierdzenie:
  *Istnieje takie **X**, że jeśli **alicja** jest matką **X** to **X** jest chłopcem.*

Klauzule Horna

- Wiele sposobów wyrażania tych samych stwierdzeń
- Stwierdzenia zapisuje sią w standardowej postaci tzw. klauzul Horna (Alfred Horn, 1918-2001)
- *Klauzule Horna*
  - głowa *(head)* + ciało *(body)*
  - Semantyka (przecinek oznacza koniunkcję $\wedge$)
    $H \Leftarrow B_1, B_2, \ldots, B_n$
    Jeśli wszystkie B_i są prawdziwe to H jest prawdziwe
  - Klauzule typu:
    $A_1 \wedge A_2 \wedge \ldots \wedge A_m \Rightarrow B_1$
    nazywa się *regułami*
  - Klauzule typu B_1 nazywamy *faktami*
  - Klauzule $A_1 \wedge A_2 \wedge \ldots \wedge A_m$ nazywamy *celami*, czyli tym czego chcemy dowieść

RK i dowodzenie twierdzeń

- Stwierdzeń używa sie do dowodzenia nowych twierdzeń na podstawie aksjomatów i innych, znanych twierdzeń
- Metodą dowodzenia jest *rezolucja,* zasada wnioskowania, która pozwala obliczyć wnioskowane stwierdzenia ze stwierdzeń przyjętych jako prawdziwe

Rezolucja, unifikacja

- Rezolucja to metoda wnioskowania.
  - Jeśli

    $\mathbf{A} \Rightarrow \mathbf{B}$ i $\mathbf{C} \Rightarrow \mathbf{D}$

    to chcemy pokazać przy jakich warunkach zachodzi

    $\mathbf{A} \Rightarrow \mathbf{D}$

    Jest to możliwe jeśli można dokonać *unifikacji* zmiennych **B** i **C**:

    $\mathbf{B} = \mathbf{C}$

    Rezolucja polega na obliczeniu $\mathbf{A \wedge C}$ oraz $\mathbf{B \wedge D}$.
    Rezolucja „podstawia” (*unifikuje*) za zmienne ich możliwe wartości i pozwala na ich dopasowanie do sytuacji na podstawie aksjomatów. Odbywa się to przez tzw. nawroty (*backtracking*) – kolejne próby aż do wyczerpania możliwości.

Dowód przez zaprzeczenie

- *Hipoteza*: zdanie lub zbiór zdań nie poddanych wystarczającemu sprawdzeniu, przyjętych prowizorycznie
- *Cel*: zanegowana hipoteza
- Twierdzeń dowodzi się przez znalezienie niezgodności

Dowodzenie

- Podstawa programowania w logice
- W dowodzeniu metodą rezolucji można używać tylko uproszczonych klauzul Horna
  - *klauzule z głową*: pojedyncze, atomowe stwierdzenia po stronie lewej
  - *klauzule bez głowy*: pusta lewa strona; używa się jej do podania faktów
  - większość (nie wszystkie) stwierdzenia dają się zapisać jako klauzule Horna

Przegląd programowania logicznego

- Semantyka deklaratywna
  - istnieją proste sposoby określenia znaczenia każdego zdania
  - prostsza niż semantyka języków imperatywnych
- Programowanie jest p. nieproceduralnym
  - w programach nie podaje się reguł (algorytmów) obliczeń lecz opisuje się wynik; jak ma wyglądać

Przykład: sortowanie listy

Opisać jak wygląda lista posortowana,

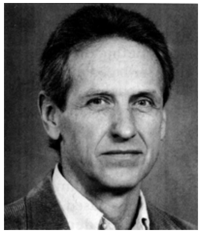
a nie sam proces sortowania

```
sort(old_list, new_list)
    ⇐ permute (old_list, new_list)
      ∧ sorted (new_list)

sorted (list) ⇐ ∀j such that 1≤j< n,
                    list(j) ≤ list (j+1)
```

Historia PROLOGu

- PROLOG = *PROgrammation en LOGique; PRO*gramming in *LOG*ic; *PROgramowanie w LOGice*
- 1972, lato
- Alain Colmerauer
  Phillipe Roussel – Uniwersytet Aix-Marseille
  Robert Kowalski – University of Edingburgh
- Pierwszy interpreter: Roussel
- Pierwszy kompilator: David Warren, Edinburgh
- 1980 – TurboProlog firmy Borland
- ISO Prolog standardisation (1995)
- Literatura: *The birth of Prolog*. Alain Colmerauer and Philippe Roussel (1992).



Dostępne środowisko:

- Autor: Daniel Diaz
- GNU Prolog web site:
  - http://www.gprolog.org/
- Manual (vers. 1.4.0):
  - http://www.gprolog.org/manual/gprolog.pdf

~~~~~~

```
GNU Prolog 1.4.4 (64 bits)
Compiled Apr 23 2013, 17:26:17 with /opt/local/bin/gcc-apple-4.2
By Daniel Diaz
Copyright (C) 1999-2013 Daniel Diaz
| ?-
```
~~~~~~

gproplog

- Polecenie
  `[user].`
  – pozwala wpisać „program”, który następnie można wykonać
- Polecenie
  `consult(„plik”).`
  – wczytanie programu z pliku

Prolog

- Prolog pracuje w oparciu o fakty zgromadzone w bazie wiedzy (klauzule Horna), o której zakłada się, że jest prawdziwa
- Stałe są atomami lub liczbami
- Struktury są predykatami logicznymi albo strukturami danych
- Atomy są podobne do atomów Lispa
  - `nic,      moja_stala,    +,    'reszta'`
- Zmienne (duża pierwsza litera)
  - `Nic,      Jakas_zmienna,        X`
- W wyniku unifikacji (rodzaj obliczania) zmienne mogą przyjmować określone wartości).
- Zakresy zmiennych są ograniczone do klauzul, w których występują
- Nie ma deklaracji zmiennych
- Sprawdzanie typów odbywa się w czasie wykonania programu

Prolog

- Struktury składają się z atomów zwanych *funktorami* i listy argumentów
  `pogodny(lublin).`
  `drzewo_bin(costam, drzewo_bin(lewy, prawy)).`
- Terminu *predykat* używa się dla kombinacji funktora i liczby argumentów, np. predykat **`pogodny/1`** – posiada 1 argument.
- Klauzule klasyfikowane są jako fakty lub reguły. Kończy je kropka.
- Fakt jest klauzulą Horna bez podanej prawej strony, np. **`pogodny(lublin).`**

Prolog

- Reguła posiada prawą stronę
  `mokry(X):- deszczowy(X), zachmurzony(X).`
  `:-` symbol implementacyjny; **`,`** należy rozumieć jako *i (and)*

- Ostatnią klauzulę (regułę) czytamy jako: Dla każdego **`X`**, **`X`** jest **`mokry`** jeśli **`X`** jest **`deszczowy`** i **`zachmurzony`**.

- Zapytania. Jeśli w bazie wiedzy mamy fakty :

  `deszczowy(lublin). deszczowy(chełm).`

  to możemy zapytać co jest deszczowe:

```
?- deszczowy(C).
C=lublin ;                <- ODPOWIEDZI
C=chełm ;
No
```

Unifikacja

- Rozwiązywanie (rezolucja) i unifikacja.
  Reguła: Jeżeli C_1 i C_2 są klauzulami Horna i głowa C_1 zgadza się z jednym ze składników (*term*) w ciele C_2, to można zastąpić term w C_2 przez ciało C_1.
- Przykład.

```
lubi(jan, windows).   lubi(jan, paradygmaty).
lubi(ala, linux).     lubi(ala, język_c).
lubi(ola, linux).     lubi(ela, windows).

lubiToSamo(X, Y) :- lubi(X, Z), lubi(Y, Z).
```

  Jeśli przyjmiemy, że `X` to `jan`, a `Z` to `windows`, to możemy zastąpić w ostatniej klauzuli ...
  `lubiToSamo(jan, Y):-lubi(Y, windows).`
  Mówiąc inaczej, `Y lubiToSamo` jeśli lubi `windows`.

- '`Z`' z prawej strony klauzuli:
  `lubiToSamo`: dla wszystkich `X` i `Y`, `X` i `Y lubiToSamo` jeśli isnieje system operacyjny, który lubią wspólnie
- Dopasowanie wzorca użytego do związania `X` z jan i `Z` z `windows` nosi nazwę unifikacji

Reguły unifikacji

- Stała unifikuje się tylko ze sobą
- Dwie struktury unifikują się wtedy i tylko wtedy jeśli mają ten sam funktor i tę samą liczbę argumentów oraz aargumenty unifikują się rekurencyjnie
- Zmienna unifikuje się ze wszystkim. Jeśli to coś posiada wartość to zmienna ją przyjmuje (instantiation)

Osiągnięcie celu

- Cel `=(A, B)` zostaje osiągnięty jeśli **A** i **B** można zunifikować (`=(A, B)` jest równoważne zapisowi **A=B**)
- To samo w języku PROLOG

```
|?- a = a
Yes
|?- a = b
No
|?- byleco(a, b).
byleco(a, b)
Yes
|?- X = a
X=a
|?- byleco(a,b) = byleco(X, b).
X=a ;
No
|?-
```

Listy

- Listy są podstawowymi strukturami Prologu
  - `[a, b, c]`
  - `.(a .(b .(c, [])))`
  - `[]` == lista pusta
- Notacja `|`
  - `[a, b, c]` ⇔ `[a | [b, c]]`
    ⇔ `[a, b | [c]]`
    ⇔ `[a, b, c | []]`

  równoważność
- Wygodny zapis gdy ogon listy jest zmienną

Przykłady


- ```
  member(X, [X | _]).
  member(X, [_ | T]):- member(X, T).
  ```

- ```
  sorted([]).      % lista pusta jest posortowana
  sorted([_]).     % singleton jest posortowany
  sorted([A, B | T]):- A=<B, sorted([B | T]).
     % złożona lista jest posortowana jeśli
     % dwa pierwsze elementy są posortowane
     % i reszta (po elemencie pierwszym)
     % jest posortowana
  ```

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

Argumenty – nierozróżnialność we/wy

- Nie rozróżnia się argumentów *wejściowych* i *wyjściowych*
- Możemy napisać:

```
?- append([a, b, c], [d,
e], L).

L=[a,b,c,d,e]

?- append(X, [d, e], [a,
b, c, d, e]).

X=[a, b, c]

?- append([a, b, c], Y,
[a, b, c, d, e]).

Y=[d, e]
```

```
?- append(X,Y,[a,b,c]).

X = []
Y = [a,b,c] ? a

X = [a]
Y = [b,c]

X = [a,b]
Y = [c]

X = [a,b,c]
Y = []
```

Arytmetyka w PROLOGu

- `+, -, *, /, ...`
- pierwszeństwo, kolejność działań; 0-1200
- łączność
- sprawdzanie:

```
current_op(Precedence, Associativity, *)
Precedence = 400
Associativity = yfx
Yes

current_op(Precedence, Associativity, **)
Precedence = 200
Associativity = xfx
No
```

Arytmetyka w PROLOGu


- ```
  current_op(Precedence, Associativity, :-)
  Precedence = 1200
  Associativity = xfx ;
  No
  ```
- Uwaga: operatorami są też `=`, `:-`, ...
- Operatory i ich własności
  - yfx – infiksowy, lewostronnie łączny (`+`, `-`, `*`)
  - xfy – infiksowy, prawostronnie łączny (`,`)
  - xfx – infiksowy, nie jest łączny (`=`, `is`, `<`)
  - fy – prefiksowy, łączny
  - fx – prefiksowy, nie jest łączny (`-`; `--5` niedozwolone)
  - yf – postfiksowy, nie jest łączny
- :- może być też operatorem prefiksowym
  ```

Arytmetyka w PROLOGu

- Zamiast pisać `jest_wiekszy_niz(słoń, koń) –` (oznacza to: słoń jest większy niż koń) chcemy pisać

  ```
  słoń jest_wiekszy_niz koń
  ```

- Operatory deklaruje się za pomocą predykatu `op/3.` Przykład:

  ```
  ?- op(300, xfx, jest_wiekszy_niz).
  Yes
  ```

  Po tej deklaracji operator `jest_wiekszy_niz` może być używany jako operator infiksowy:

  ```
  ?- słoń jest_wiekszy_niz koń.
  Yes
  ```

Arytmetyka—przykład


```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Zapytanie: `distance(chevy, Chevy_Distance).`

Metoda nawracania (*backtracking*)

- Jeśli istnieje więcej niż jedna możliwość unifikacji zmiennych PROLOG próbuje kolejno możliwych kombinacji. Jeśli któraś z unifikacji nie powiedzie się, PROLOG wraca do miejsca, w którym unifikacja została dokonana i dokonuje następnej możliwej próby. Nosi to nazwę nawracania (*backtracking*).

- Przykład.

```
permutatacja([], [])
permutatacja(Lista, [Element|Permutacja]) :-
      select(Element, Lista, Reszta),
      permutacja(Reszta, Permutacja).
```

Uwaga na duże litery!

Metoda nawracania

- Wykonanie polecenia

```
?- permutacja([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
No
```

- Problemy z nawracaniem. *Przycinanie*, **!**, pozwala ograniczyć nawracanie

Rekurencja

- Wiele stwierdzeń w regułach programów wymusza rekurencję.

  Przykłady na ćwiczeniach.

Negacja i niepowodzenie

- Zamknięty świat PROLOGu
  - `Yes` oznacza nie tylko prawdziwość postawionej tezy, czy zapytania, ale również ich dowodliwość na podstawie dostępnych założeń.
  - Odpowiedź `No` nie mówi iż teza jest koniecznie fałszem, lecz że nie została udowodniona jej prawdziwość.
  - Program w PROLOGu jest z założenia całym dostępnym do analizy zbiorem faktów (światem Prologu)
  - jeśli `kogut` nie występuje w bazie wiedzy programu o `ptak`ach, to odpowiedź na pytanie
    `ptak(kogut).  -> No`
    nie oznacza oczywiście, że `kogut` to nie `ptak`. Oznacza to niezupełność bazy wiedzy dostępnej dla Prologu. Tylko wtedy gdy baza wiedzy jest zupełna `No` i `not true (false)` oznaczają to samo.

Operator `\+`

- Jeśli chcemy zapytać czy dany cel nie jest spełniony (zazwyczaj pytamy czy jest spełniony), to znaczy czy spełniony jest zanegowany cel, wówczas używamy operatora `\+`.
- Operator `\+` można zastosować do dowolnego poprawnie sformułowanego celu.
- Dowiedliwość celu `\+Cel` oznacza niepowodzenie w dowodzie stwierdzenia `Cel` (tzn. `Cel` nie jest dowiedzioną prawdą).
- Semantyka operatora negacji oznacza negację niepowodzenia.
- W życiu codziennym pojęcie to nie ma raczej odpowiedników (ale w prawie: dowodzi się winy a nie niewinności: *dopóty ktoś jest niewinny, dopóki nie udowodni się jego winy*).

Alternatywa

- Przecinek w ciele zapytania lub reguły oznacza koniunkcję. Sukces oznacza prawdziwość wszystkich zdań w koniunkcji
- Jeśli dwie reguły maja tę samą *głowę* mamy do czynienia z wyborem (alternatywa). Prolog, w przypadku niepowodzenia pierwszej reguły lub w przypadku gdy użytkownik żąda alternatywnych rozwiązań, przystepuje do próby z regułą drugą
- Zapis reguł można wówczas uprościć, rozdzielając reguły średnikiem (;)

Alternatywa

- Przykład
  Zamiast reguł:
  ```
  rodzic(X, Y) :- ojciec(X, Y).
  rodzic(X, Y) :- matka(X, Y).
  ```
  Piszemy:
  ```
  rodzic(X, Y) :- ojciec(X, Y);
                  matka(X, Y).
  ```
- Pierwszeństo operatora `;` przed `,`.
  W złożeniach należy o tym pamiętać!

Przykład—formuły logiczne

- Program sprawdzania tabeli prawdy

A	B	A ∧B
T	T	T
T	F	F
F	T	F
F	F	F

- Przykład
  `true and (true and false implies true) and neg false`

  `true` i `false` są atomami PROLOGu.

Operatory koniunkcji i alternatywy to , i ;
Do wywoływania reguł reprezentowanych przez zmienne używamy wbudowanego predykatu `call/1`.

```
and(A, B) :- call(A), call(B).
or(A, B) :- call(A); call(B).
```

Nasz operator negacji `neg` definiujemy regułą:

```
neg(A) :- \+ call(A).
```

Należy zdefiniować implikację $A \Rightarrow B \equiv \neg A \vee B$.
Można również użyć operatora obcinania `!`

```
implies(A, B) :-
      call(A), !, call(B).
implies(_, _).
```

(Działanie. Przypuśćmy, że `A` jest `false`. Pierwsza reguła nie dziala, PROLOG przechodzi do drugiej i mamy sukces niezależnie od `B`. Jeśli call(A) jest prawdą to mamy obcięcie i sukces jest osiągnięty jeśli `call(B)` jest `true`)

Korespondencja PROLOG-RK

- Korespondencja między PROLOGiem, a logiką (rachunkiem kwantyfikatorów pierwszego rzędu, RK)
  - PROLOG:
    ```
    wiekszy(słoń, koń).   większy(koń, osioł).
    jest_wiekszy_niz(X, Y) :-  większy(X, Y).
    jest_wiekszy_niz(X, Y) :-  większy(X, Z),
                               jest_wiekszy_niz(Z, Y).
    ```
  - Logika:
    ```
    {wiekszy(słoń, koń), wiekszy(koń, osioł),
    ∀x.∀y(wiekszy(x,y) ⇒ jest_wiekszy_niz(x,y)),
    ∀x.∀y(wiekszy(x,z)∧jest_wiekszy_niz(z,y)
                        ⇒ jest_wiekszy_niz(x,y))  }
    ```
  - Pojedyncze stwierdzenia
    ```
    ?- jest_wiekszy_niz(słoń,X), jest_wiekszy_niż(X, osioł).

    ∀x.(jest_wiekszy_niz(słoń,x)∧jest_wiekszy_niz(x,osioł)
    ```

Podsumowanie

- Reguły translacji
  - Każdy predykat PROLOGu odwzorowujemy na regułę atomową pierwszego rzędu w logice
  - Przecinek oznacza koniunkcję
  - Reguły to implikacje, w których ciało stanowi przesłankę, a głowa tezę stwierdzenia (:- oznacza ⇒, zmieniamy też kolejność: głowa – ciało)
  - Zapytania odwzorowuje się w implikacje, w których ciało jest przesłanką, a teza jest fałszem, *falsum* ⊥ (`:-` lub `?-` zastępujemy przez ⇒; po ciele stawiamy znak implikacji ⇒ oraz fałszu ⊥: `ciało_reguły ⇒ ⊥`)
  - Każda zmienna w klauzuli jest opatrzona kwantyfikatorem uniwersalnym ∀(np. ∀`x` = dla każdego `x`; `x` – zmienna)

Podsumowanie

- Ponieważ $A \Rightarrow \bot \equiv \neg A$, (równoważność $A \Rightarrow \bot$ oraz $\neg A$), więc każdą regułę Prologuu da się zapisać jako

$$A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow B \equiv$$
$$\neg (A_1 \wedge A_2 \wedge ... \wedge A_n) \vee B \equiv$$
$$\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_n \vee B$$

- Jeśli B jest $\bot$ otrzymamy

$$\neg A_1 \vee \neg A_2 \vee ... \wedge \neg A_n \vee \bot \equiv$$
$$\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_n$$

Podsumowanie

- Każda formuła prologu da się zapisać w języku kwantyfikowanych literałów alternatyw z co najwyżej jednym literałem pozytywnym, a więc w postaci klauzuli Horna (literał pozytywny = pojedynczy atom). Fakty to też klauzule Horna.
- Klauzule Horna utożsamia się z koniunkcjami alternatyw literałów z co najwyżej jednym literałem pozytywnym każda; Tworzą one cały program w PROLOGu

Podsumowanie

- Jest to równoważne dowiedzeniu, że cel wynika ze zbioru reprezentującego program:
  $\mathbf{P},\ (\mathbf{A} \rightarrow \bot) \vdash \bot \Leftrightarrow \mathbf{P} \vdash \mathbf{A}$
  Czyli: dla pokazania, że **A** wynika z **P** (prawa strona) należy pokazać, że dodanie negacji **A** do **P** prowadzi do absurdu (lewa strona), sprzeczności.
- Odpowiednią metodą dowodzenia takich twierdzeń jest rezolucja. Np. w prostym przypadku:
  $\neg A_1 \vee \neg A_2 \vee B_1$
  $\neg B_1 \vee \neg B_2$

  $\neg A_1 \vee \neg A_2 \vee \neg B_2$

PROLOG. Reguła:
b1 :- a1, a2
druga z formuł to zapytanie:
?- b1, b2
Odpowiedź:
?- a1, a2, b2

(w przypadku gdy B_1 jest fałszem musi zachodzić $\neg A_1 \vee \neg A_2$, a gdy B_1 jest prawdą wówczas mamy $\neg B_2$)

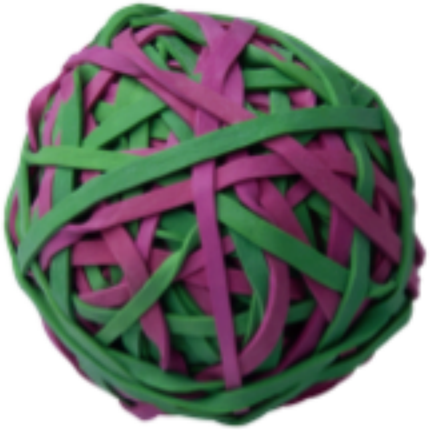
Zastosowania PROLOGu

- Systemy zarządzania relacyjnymi bazami danych
- Systemy ekspertowe
- Przetwarzanie języków naturalnych
- Programowanie gier

Literatura

- Sebesta, rozdział 16
- Scott …
- Clocksin, Mellish: Programming in Prolog, Springer, 2003.
- Internet:
  - http://www.csee.umbc.edu/portal/help/prolog/
  - Ulle Endriss: Lecture notes. An introduction to Prolog programming
  - Manual gprolog-u:
    - http://www.gprolog.org/manual/

Za tydzień: .. współbieżność



Parts of logic not covered...

As noted in Section 11.3, Horn clauses do not capture all of first-order predicate calculus. In particular, they cannot be used to express statements whose clausal form includes a disjunction with more than one nonnegated term. We can sometimes get around this problem in Prolog by using the not predicate, but the semantics are not the same (see Section 11.4.3).

[Scott, R11]

Przykład: trójki Pitagorasa


```
pythag(X, Y, Z) :-
        intriple(X, Y, Z),
        Sumsq is X*X + Y*Y, Sumsq is Z * Z.
intriple(X, Y, Z) :-
        is_integer(Sum),
        minus(Sum, X, Sum1), minus(Sum1, Y, Z).
minus(Sum, Sum, 0).
minus(Sum, D1, D2) :-
        Sum > 0, Sum1 is Sum - 1,
        minus(Sum1, D1, D3), D2 is D3 + 1.
is_integer(0).
is_integer(N) :- is_integer(N1), N is N1 + 1.
```

Symbole

NAZWA	SYMBOL	PRZYKŁAD	ZNACZENIE
negacja	$\neg$	$\neg a$	nie a
koniunkcja	$\wedge$	$a \wedge b$	a i b
alternatywa	$\vee$	$a \vee b$	a lub b
równoważność	$\equiv$	$a \equiv b$	a jest równoważne b
implikacja	$\Rightarrow$	$a \Rightarrow b$	a implikuje b

falsum in uno, falsum in omni