# Języki i Paradygmaty Programowania

## Prolog i programowanie w logice 2


Marcin Benke

MIM UW


16 maja 2016

# Interpretacja zapytań

Załóżmy, ze mamy pewien zbiór aksjomatów (klauzul programu) $\Phi$.

Klauzula negatywna

$$\leftarrow \mathrm{B}(\vec{x})$$

reprezentuje pytanie

> *"Dla jakich wartości $\vec{x}$, w każdym modelu spełniającym aksjomaty $\Phi$ zachodzi B?"*

Jak można znaleźć takie wartościowanie?

# Dowody przez sprzeczność

Mamy pewną teorię (zbiór formuł) $\Phi$ i formułę $B$.

Chcemy się przekonać, czy $\Phi \models B$ (albo: dla jakich wartościowań $\Phi, \theta \models B$)

**Fakt 1:** $\Phi \models B \Leftrightarrow \Phi \cup \neg B \models \bot$

**Fakt 2:** Jeśli $\vdash$ jest poprawnym systemem dowodzenia, to

$$\Phi \cup \neg B \vdash \bot \Rightarrow \Phi \cup \neg B \models \bot$$

**Dowód przez sprzeczność:**

- Zacznij od $\Phi \cup \neg B$
- Stosuj reguły dowodzenia aż uzyskasz sprzeczność
- Jeśli się udało, to wiemy, że $\Phi \models B$

Dokładniej, w trakcie dowodu będziemy konstruować podstawienie $\theta$ takie, że $\Phi \models \theta(B)$ (czyli $\Phi, \theta \models B$)

# Rezolucja

Prolog znajduje rozwiązania przy pomocy mechanizmu rezolucji.

W logice zdaniowej poprawna jest reguła

$$\frac{\neg\alpha \vee \beta \qquad \alpha \vee \gamma}{\beta \vee \gamma}$$

Szczególnymi jej przypadkami są

$$\frac{\neg\alpha \vee \beta,\ \ \alpha}{\beta} \qquad \frac{\neg\alpha,\ \ \alpha}{\bot}$$

# Rezolucja dla klauzul

Reguła rezolucji w logice predykatów (klauzul)

$$\frac{B \leftarrow A_1, \ldots, A_n \qquad D \leftarrow C, C_1, \ldots, C_k}{(D \leftarrow A_1, \ldots, A_n, C_1, \ldots, C_k)\theta}$$

gdzie $\theta = mgu(B, C)$ (czyli $B\theta = C\theta$).

Klauzula programu i klauzula negatywna:

$$\frac{B \leftarrow A \qquad \leftarrow C}{\leftarrow A\theta}$$

Fakt i klauzula negatywna

$$\frac{B \leftarrow \qquad \leftarrow C}{\perp}$$

# Banalny przykład rezolucji w Prologu

Rozważmy banalny przykład programu i zapytania w Prologu:

```
even(z).
```

```
?- even(X)
```

Zapytanie jest klauzulą negatywna, którą nazywamy *celem*.
Możemy znaleźć rozwiązanie używając reguły rezolucji:

$$\frac{even(z) \leftarrow \qquad \leftarrow even(X)}{\bot} \quad \theta = [X := z]$$

```
?- even(X).
X = z.
```

# Prosty przykład rezolucji w Prologu

```
even(z).
even(s(s(X))) :- even(X).

?- even(s(X))
```

$$\frac{even(z) \leftarrow \qquad \dfrac{even(s(s(Y))) \leftarrow even(Y) \qquad \leftarrow even(s(X))}{\leftarrow even(Y)}\;[X := s(Y)]}{\perp}\;[Y := z]$$

```
?- even(s(X)).
X = s(z) .
```

# Mechanizm rezolucji — klauzule ustalone

Rozważmy najpierw wersję uproszczoną — wszystkie klauzule ustalone

Niech $G$ będzie celem (klauzulą negatywną):

$$G = \leftarrow C_1, \dots, C_i, \dots C_n$$

jeśli program zawiera klauzulę

$$K = C \leftarrow B_1, \dots, B_k$$

taką, że $C \equiv C_i$. Wówczas cel $G$ możemy zastapić jego *rezolwentą* $G_1$:

$$G_1 = \leftarrow C_1, \dots, C_{i-1}, B_1, \dots, B_k, C_{i+1}, \dots C_n$$

# Mechanizm rezolucji — klauzule ustalone

Wydawałoby się, że mechanizm rezolucji prowadzi nas do coraz większych celów. Zwróćmy jednak uwagę na przypadki gdy K jest faktem:

$$G = \ \leftarrow C_1, \ldots, C_i, \ldots C_n$$

$$K = C \leftarrow \qquad (C \equiv C_i)$$

wtedy

$$G_1 = \leftarrow C_1, \ldots, C_{i-1}, C_{i+1} \ldots C_n$$

w szczególności gdy

$$G = \leftarrow C$$

$$K = C \leftarrow$$

rezolwenta jest pusta, co oznacza koniec dowodu.

# Mechanizm rezolucji — klauzule ze zmiennymi

Mechanizm rezolucji można uogólnić na przypadek klauzul ze zmiennymi

Niech G będzie celem (klauzulą negatywną):

$$G = \leftarrow C_1, \ldots, C_i, \ldots C_n$$

jeśli program zawiera klauzulę pasującą do $C_i$ ($mgu(C, C_i) = \theta$)

$$K = C \leftarrow B_1, \ldots, B_k$$

Wówczas cel $G$ możemy zastąpić jego *rezolwentą* $G_1$:

$$G_1 = \leftarrow (C_1, \ldots, C_{i-1}, B_1, \ldots, B_k, C_{i+1}, \ldots C_n)\theta$$

# Uwaga o zmiennych

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$$

oznacza

$$\forall \vec{x}. (B_1 \wedge \ldots \wedge B_n) \rightarrow (A_1 \vee \ldots \vee A_k)$$

czyli wystapienia zmiennej $X$ w dwóch różnych klauzulach to tak naprawdę **różne** zmienne.
Dlatego przy rezolucji używamy wariantu K' klauzuli K ze świeżymi nazwami zmiennych, np.

$$G = \leftarrow p(X, b)$$

$$K = p(Y, X) \leftarrow q(Y, X)$$

$$K' = p(Y', X') \leftarrow q(Y', X')$$

$$G' = \leftarrow q(Y', X')\theta$$

gdzie $\theta = mgu(p(X, b), p(Y', X'))$.

# Algorytm rezolucji

Krok algorytmu: dany program $P$, cel $G_k = \leftarrow C_1, \ldots, C_n$
Jeśli $n = 0$ to algorytm kończy się sukcesem.

1. Wybieramy $C_i$ (jeden z atomów $G$).
2. Wybieramy klauzulę
$$K = C \leftarrow B_1, \ldots, B_k$$
taką że $C$ uzgadnialne z $C_i$. Jeśli nie ma takiej klauzuli, to algorytm kończy się porażką.
3. Tworzymy $K'$ — wariant $K$ ze świeżymi zmiennymi
4. Za $G_{k+1}$ przyjmujemy rezolwentę $G_k$ i $K'$, zaś $\theta_{k+1} = mgu(C', C_i)$

# Algorytm rezolucji

- Jest to (z pewnymi uproszczeniami) tzw SLD-rezolucja.
- Jest to algorytm niedeterministyczny: daje odpowiedź pozytywną jeśli dla danego celu początkowego $G$ istnieje skończony ciąg kroków prowadzący do sprzeczności (dający podstawienia $\theta_1, \ldots, \theta_n$
- Wtedy $\theta = \theta_1 \ldots \theta_n|_{var(G)}$ jest kontrprzykładem dla $P \cup G$.
- Takie $\theta$ nazywamy *odpowiedzią obliczoną*.

Zastanowimy się teraz nad możliwościami determinizacji tego algorytmu.

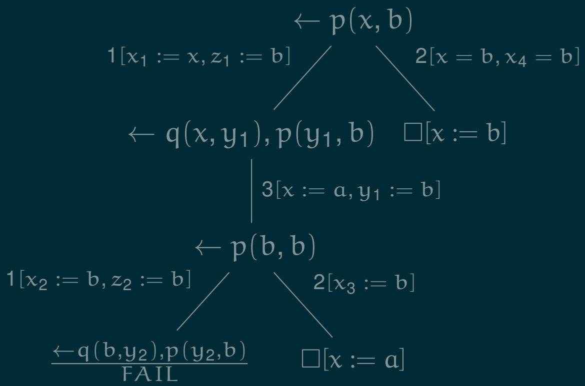
# Drzewa poszukiwań

- Możliwe przebiegi rezolucji możemy przedstawic przy pomocy drzewa
- W korzeniu początkowe zapytanie
- Potomkowie klauzuli odpowiadają możliwym wyborom klauzul dla wybranego atomu
- Dla uproszczenia zawsze wybieramy skrajny lewy atom.
- Każda z gałęzi może
  - kończyć się sukcesem (z podstawieniem — rozwiązaniem)
  - kończyć się porażką (nie ma uzgadnialnych klauzul dla atomu)
  - być nieskończona
- Prolog realizuje przeszukanie w głąb, w kolejności definicji klauzul.

# Drzewa poszukiwań

1. $p(x,z) \leftarrow q(x,y), p(y,z)$
2. $p(x,x) \leftarrow$
3. $q(a,b) \leftarrow$

G. $\leftarrow p(x,b)$

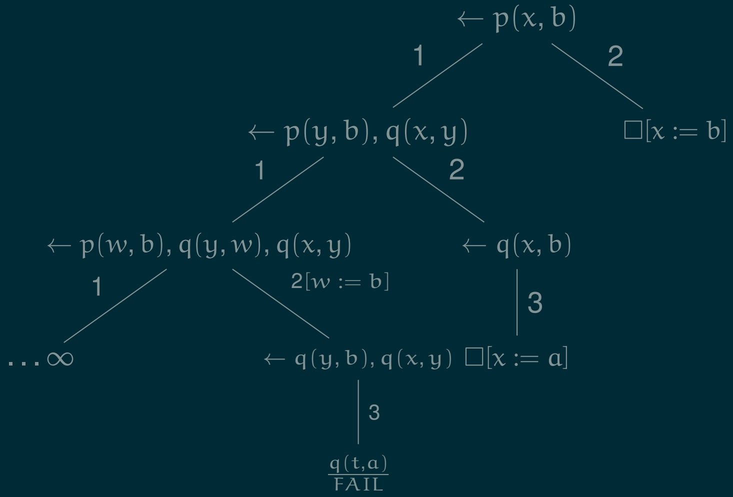



- Drzewo zawodzi, jeśli wszystkie jego poddrzewa zawodzą
- W tej sytuacji cofamy się do wierzchołka wyżej (i ew. próbujemy kolejne poddrzewo) — nawracanie (backtracking).

# Drzewa poszukiwań

1. $p(x, z) \leftarrow p(y, z), q(x, y)$
2. $p(x, x) \leftarrow$
3. $q(a, b) \leftarrow$

G. $\leftarrow p(x, b)$

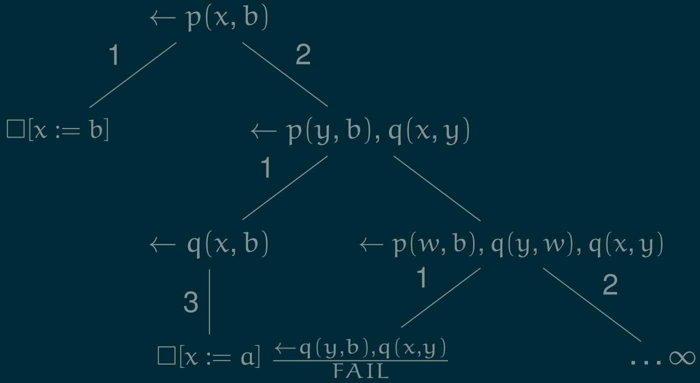



**Morał:** kolejnośc atomów w klauzuli ma znaczenie. . .

# Drzewa poszukiwań

1. $p(x, x) \leftarrow$
2. $p(x, z) \leftarrow p(y, z), q(x, y)$
3. $q(a, b) \leftarrow$

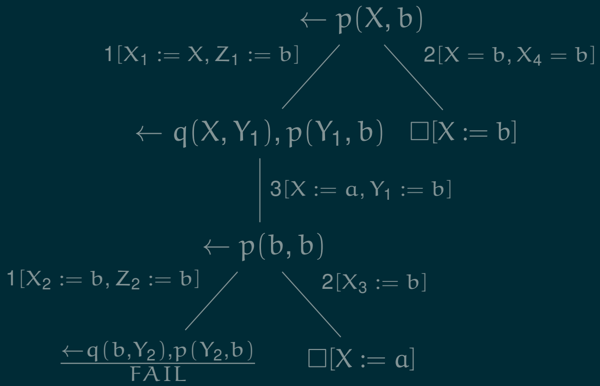
G. $\leftarrow p(x, b)$




**Morał:** kolejnośc klauzul ma istotne znaczenie. . .

# Śledzenie przebiegu wyszukiwania

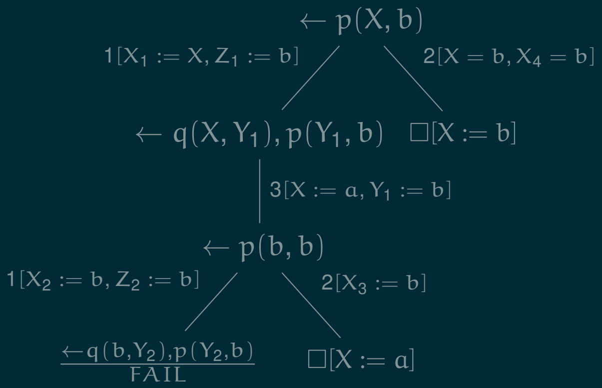
```
p(X,Z) :- q(X,Y),p(Y,Z).
p(X,X).
q(a,b).

| ?- trace.
| ?- p(X,b).
  1   1 Call: p(_496,b)
  2   2 Call: q(_496,_736)
  2   2 Exit: q(a,b)
  3   2 Call: p(b,b)
...
```




Pierwsza liczba wskazuje poziom w drzewie rezolucji
Druga — na którym poziomie literał został wprowadzony.

# Śledzenie przebiegu wyszukiwania

```
...
  4  3 Call: q(b,_941) ?
  4  3 Fail: q(b,_941) ?
  3  2 Exit: p(b,b) ?
? 1  1 Exit: p(a,b) ?
X = a ? ;
  1  1 Redo: p(a,b) ?
  1  1 Exit: p(b,b) ?
X = b ?
```




`Call` oznacza przejście wierzchołka w dół, `Exit` w górę.
`Redo` oznacza ponowne odwiedzenie wierzchołka w poszukiwaniu alternatywnego rozwiązania

`?` w pierwszej kolumnie oznacza punkt nawrotu

# Nawracanie (backtracking)

- wybór klauzuli programu (uzgodnienie)
- jeśli możliwy inny wybór — utworzenie *punktu nawrotu*
- porażka powoduje *nawrót* — wycofanie uzgodnień dokonanych pomiędzy punktem nawrotu a porażką i wybór kolejnej klauzuli.

```
g(X) :- f(X), h(X).
f(a).
f(b).
h(b).

?- g(X).
```

Pierwszym rozwiązaniem dla $f(X)$ będzie $X = a$,
ale nie da się udowodnić $h(a)$ — nawrót i kolejna próba dla $f(X)$.

# Nawracanie – przykład

```
g(X) :- f(X), h(X).
f(a).
f(b).
h(b).
?- g(X).
        1       1 Call: g(_496) ?
        2       2 Call: f(_496) ?
?       2       2 Exit: f(a) ? % mozliwy nawrót
        3       2 Call: h(a) ?
        3       2 Fail: h(a) ?
        2       2 Redo: f(a) ? % nawrót
        2       2 Exit: f(b) ?
        4       2 Call: h(b) ?
        4       2 Exit: h(b) ?
        1       1 Exit: g(b) ?
X = b
```

# Wyszukiwanie pasujących klauzul — indeksacja

Prolog zwykle nie przeszukuje wszystkich klauzul, ale indeksuje je po nazwie (tudzież arności) predykatu

Wiekszość implementacji (w tym SICStus, SWI) indeksuje także funktor główny pierwszego argumentu.

Nie zmienia odpowiedzi, ale może zmienic przebieg (i wydajność) poszukiwań.

Lepiej używać (i planować użycie) predykatów z pierwszym argumentem (choćby częściowo) ustalonym.

# Indeksacja — przykład

```
k(a,b).
k(b,a).

% trace
| ?- k(a,X).
        1      1 Call: k(a,_511) ?
        1      1 Exit: k(a,b) ?
X = b ? ;
no
% trace
| ?- k(X,b).
        1      1 Call: k(_499,b) ?
?       1      1 Exit: k(a,b) ?
X = a ? ;
        1      1 Redo: k(a,b) ?
        1      1 Fail: k(_499,b) ?
no
```

# Odcięcie (cut)

Odcięcie jest mechanizmem ograniczającym przestrzeń poszukiwań.

```
g(X) :- f(X), !, h(X).
f(a).
f(b).
h(b).
```

Odcięcie (!) po $f(X)$ oznacza, że po dokonaniu wyboru reguły dla $f$ nie można go zmienić (odcięcie pozostałych możliwości):

```
?- g(X).
        1       1 Call: g(_496) ?
        2       2 Call: f(_496) ?
?       2       2 Exit: f(a) ?
% ! odcięło możliwośc wyboru f(b).
        3       2 Call: h(a) ?
        3       2 Fail: h(a) ?
        1       1 Fail: g(_496) ?
no
```

# Odcięcie (cut)

```
h(X,Y) :- g(X), !, f(Y).
h(p,q).
f(a).
f(b).
g(c).
g(d).
```

Odcięcie po $\mathrm{g}(\mathrm{X})$ oznacza również, że jeśli do niego dotrzemy, nie możemy zmienić wyboru reguły dla $\mathrm{h}$:

```
  ?- h(X,Y).
X = c,
Y = a ;
X = c,
Y = b.

  ?- h(p,Y).
Y = q.
```

# Odcięcie (cut)

```
g(X) :- f(X), h(X).
f(a) :- !.
f(b).
h(b).
?- g(X).
        1      1 Call: g(_496) ?
        2      2 Call: f(_496) ?
        2      2 Exit: f(a) ?
        3      2 Call: h(a) ?
        3      2 Fail: h(a) ?
        1      1 Fail: g(_496) ?
no
| ?- f(b).
yes
| ?- f(X).
X = a ? ;
no
```

NB brak punktu nawrotu przy `Exit f(a)`.

# Maksimum

Rozważmy predykat `max/3`: trzeci argument jest maksimum dwóch pierwszych.

```
?- max(2,3,Max).
Max = 3

?- max(2,1,Max).
Max = 2.
```

Mozemy go prosto zdefiniować:

```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- X>Y.
```

# Maksimum (1)

```
max(X,Y,Y) :- X =< Y.
max(X,Y,X) :- gt(X,Y).
gt(X,Y) :- X>Y.
```

```
| ?- max(1,2,Max),fail.
        1      1 Call: max(1,2,_527) ?
?       1      1 Exit: max(1,2,2) ?
        1      1 Redo: max(1,2,2) ?
        2      2 Call: gt(1,2) ?
        2      2 Fail: gt(1,2) ?
        1      1 Fail: max(1,2,_527) ?
```

Zupełnie niepotrzebnie wchodzimy do drugiej klauzuli `max`. która nie ma szans powodzenia. Marnotrawstwo dla bardziej kosztownego `gt`.

# Maksimum (2)

Możemy usprawnić przy pomocy odcięcia:

```
max(X,Y,Y) :- X =< Y, ! .
max(X,Y,X) :- gt(X,Y).
gt(X,Y) :- X>Y.

% trace
| ?- max(1,2,Max),fail.
        1      1 Call: max(1,2,_527) ?
        1      1 Exit: max(1,2,2) ?
no
```

To jest tzw. *zielone* odcięcie: program daje dokładnie takie same odpowiedzi jak bez niego, ale potencjalnie sprawniej.

# Maksimum (3)

Skoro w sytuacji gdy $X \leqslant Y$ nie wchodzimy do drugiej klauzuli, moze być kuszące jej uproszczenie:

```
max(X,Y,Y) :- X =< Y, ! .
max(X,Y,X).
```

```
| ?- max(2,3,Max).
Max = 3 ? ;
no
| ?- max(2,1,Max).
Max = 2 ? ;
no
| ?- max(2,3,2).
yes
```

Oops.
Takie odcięcie zmienia semantykę programu; tzw *czerwone* odcięcie.
Unikać (zwłaszcza w programie zaliczeniowym).

# otherwise ?

W Haskellu zapisalibyśmy `max`

```
max x y | x <= y     = y
        | otherwise = x
```

W Prologu

```
max(X,Y,Z) :-
    (  X =< Y
    -> Z = Y
    ;  Z = X
     ).
```

Semantyka `->` jest dość złożona, ale w przybliżeniu

```
If -> Then  ; _Else :- If, !, Then.
If -> _Then ; Else  :-     !, Else.
If -> Then          :- If, !, Then.
```

# Używanie odcięcia

- Jeśli nie jest sie pewnym, czy użyć w danym miejscu odcięcia, to nie używać.
- Główną zaletą odcięcia jest usprawnienie programu, więc wstawiamy **po** napisaniu i uruchomieniu.
- Wstawiamy w miejscu, w którym mamy pewność, że jesteśmy na właściwej ścieżce.

```
insert(X,[Y|Z],[Y|T]) :- X>=Y, !, insert(X,Z,T).
insert(X,[Y|Z],[X|[Y|Z]]) :- X<Y.
insert(X,[],[X]).
```

Zielone odcięcie.

# Potwierdzenie wyboru reguły

```
fac(M,1) :- M=<0.
fac(M,N) :- M1 is M-1, fac(M1,N1),N is M*N1.

?- fac(0,X).
X = 1 ;
X = 0 ;
X = 0 ;
...

fac(M,1) :- M=<0,!.

?- fac(0,X).
X = 1.
```

Czerwone odcięcie, ale w tym wypadku potrzebne.

# Potwierdzenie wyboru reguły

```
elem1(X,[X|_]).
elem1(X,[_|Ys]) :- elem1(X,Ys).

?- elem1(a,[a,b,r,a,c,a,d,a,b,r,a]).
true ;
true ;
true ;
true ;
true ;
false.

elem2(X,[X|_]) :- !.
elem2(X,[_|Ys]) :- elem2(X,Ys).

?- elem2(a,[a,b,r,a,c,a,d,a,b,r,a]).
true.
```

# Potwierdzenie wyboru reguły - uwagi

```
elem1(X,[X|_]).
elem1(X,[_|Ys]) :- elem1(X,Ys).

elem2(X,[X|_]) :- !.
elem2(X,[_|Ys]) :- elem2(X,Ys).
```

SICStus i tak optymalizuje `elem1/2`:

```
 ?- elem1(a,[a,b,r,a,c,a,d,a,b,r,a]).
yes
```

To nie zawsze jest to czego chcemy:

```
?- elem1(X,[a,b]).
X = a ;
X = b ;
false.

?- elem2(X,[a,b]).
X = a.
```

# Negacja

Czasem potrafimy szybko stwierdzić, że dany cel musi zawieść:

```
brother(X,Y) :- female(X), !, fail.
brother(X,) :- ...
```

Prolog ma wbudowany predykat `\+(X)` który odnosi sukces, gdy X zawodzi, np.

```
?- \+(1=1).
false.

?- \+(1=2).
true.
```

# Generuj i testuj

- Częsta technika symulacji obliczeń niedeterministycznych:
  - generuj potencjalne rozwiązania (kandydaty)
  - sprawdź czy kandydat jest rozwiązaniem.

```
solution(X):- generate(X), test(X).
```

- Przykład:

```
cousin(P1,P2) :-
    child(P1,P3),
    child(P2,P4),
    sibling(P3,P4).
```

- Faza testowania przebiega przeważnie na termach ustalonych.

# Deklaracje operatorów

```
:- op(priorytet, rodzaj, nazwa)
```

- priorytet — liczba 1..1200
- rodzaj określa pozycję i wiązanie,np
  - fx — prefiksowy, nie wiążący
  - xfy — infiksowy, wiążący w prawo
  - yf — postfiksowy wiażący w lewo
- nazwa — atom

```
op(1200,xfx,':-').
op(1200,fx,[:-,?-]).
op(1100,xfy,';').
op(1000,xfy,',').
op(700,xfx,[=,is,<,>,=<,>=,==,=:=]).
op(500,yfx,[+.-]).
op(500,fx,[+,-,not]).
op(400,yfx,[*,/,div]).
op(300,xfx,mod).
```

# Kilka użytecznych predykatów

`atom(X)` — `X` jest atomem
`integer(X)`
`atomic(X)` — liczba lub atom
`var(X)` —- `X` jest (nieukonkretnioną) zmienną
`atom_chars(A,L)` — A jest atomem złożonym listy znaków L, np

```
| ?- atom_codes(abc,L).
L = [97,98,99] ? ;
no
| ?- atom_codes(A, "ala").
A = ala ?
yes
| ?- char_code('a',X).
X = 97 ? ;
no
| ?- char_code(C,97).
C = a ? ;
```

# “Naiwny” Interpreter Prologu

```
-- github.com/mbenke/toy-interpreters/tree/master/Prolog1/
-- Credit: Mark Jones
data Prooftree = Done Subst | Choice [Prooftree]

prooftree :: Database -> Int -> Subst -> [Term] -> Prooftree
prooftree db = pt where
      -- Depth, part res, goals
   pt :: Int -> Subst -> [Term] -> Prooftree
   pt n s [] = Done s
   pt n s (g:gs) = Choice [ pt (n+1) (u@@s) $mapApply u (tp++gs)
                          | tm :- tp <- renClauses db n g
                          , u <- unify g tm
                          ]

search :: Prooftree -> [Subst]
search (Done s) = [s]
search (Choice pts) = [s | pt <- pts, s <- search pt]
```