Prolog

Materiały do wykładu Komputerowe Przetwarzanie Wiedzy

Tomasz Kubik

Programowanie w logice i Prolog

- **Programowania w logice** (ang. *logic programming*) bazuje na regułach wnioskowania logicznego, które opracowano już w starożytności (Aristoteles)
- Odkryte na nowo w latach 1960 i 1970 przy okazji debaty nad użyciem deklaratywnej lub proceduralnej reprezentacji sztucznej inteligencji

- **Prolog** (akronim od ang. *PROgramming in LOGic*)
  - Symboliczny język programowania (do obliczeń symbolicznych, nie do numerycznych)
  - Stworzony w 1972 (autorami Alain Colmerauer oraz Philippe Roussel)

- Istota programowania w Prologu
  - Definiowanie faktów i relacji, zadawanie pytań o relacje
  - Interpreter
    - | ?- zaczęta główna
    - | zachęta podrzędna

Dostępne implementacje Prologu

- SWI-Prolog
  - http://www.swi-prolog.org/
- SWI-Prolog-Editor
  - http://lernen.bildung.hessen.de/informatik/swiprolog/indexe.htm
- YAProlog
  - http://www.ncc.up.pt/~vsc/Yap/
- Strawberry Prolog
  - http://www.dobrev.com/
- Visual Prolog
  - http://www.visual-prolog.com/
  - Successor of Turbo Prolog (by Borland)
- SICStus Prolog
  - http://www.sics.se/isl/sicstuswww/site/index.html
- GNU GProlog
  - http://www.gprolog.org/

Składnia

- Znaki
  - duże litery (`A-Z`), małe litery (`a-z`), cyfry (`0-9`),
  - symbole (`+ - * / \ ^ , . ~ : . ? @ # $ &`)
- Termy
  - liczby
    - zazwyczaj całkowite, choć mogą być też zmiennoprzecinkowe
  - atomy
    - ciągi znaków zaczynające się małą literą (`czlowiek`, `duzy_czlowiek`)
    - ciągi znaków zamieszczone a apostrofach (`'dobry zart'`, `'&^%&#@$ &*'`), są to nazwy atomów
    - ciągi znaków specjalnych (`@=` oraz `====>` oraz `;` oraz `:-`), niektóre mają predefiniowane znaczenie
  - zmienne
    - ciągi znaków zaczynające się od dużej litery lub podkreślenia (`X`, `_jakis`)
    - podkreśleniem zaczynają się **zmienne anonimowe**
  - termy złożone
    - składają się z funktora (ang. *functor*), który musi być atomem, oraz
    - sekwencji argumentów rozdzielonych przecinkami, które mogą być czymkolwiek, umieszczonej w nawiasach (`ojciec(X, ojciec(Y, ela))`)
    - ilość argumentów to arność predykatu (do odwołań w tekście oznaczana liczbą po ukośniku, np. `ojciec/1`)
    - mogą mieć strukturę rekursywną

Struktura programu

- Kod zazwyczaj w pliku z rozszerzeniem .pl
- Kod to sekwencja klauzul
  - fakty lub predykaty (`pred(arg1, arg2, … argN).`)
  - reguły (`head :- body.`)
    - termy w klauzuli mogą być rozdzielone przecinkami (`,` – koniunkcja) lub średnikami (`;` – alternatywa)
    - pomiędzy termami może pojawić się znak wykrzyknika (! – odcięcie)
  - zapytanie lub cel (`goal`)
  - komentarze
    - blokowe (`/* This is a comment */`)
    - linijkowe (% `This is also a comment`)

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
happy(yolanda).
listensToMusic(yolanda):- happy(yolanda).
?- woman(mia).
yes
?- playsAirGuitar(mia).
no
```

Wczytanie kodu

- Kod z pliku można wczytać do interpretera
  - polecenie `consult(nazwaPliku).`
- Fakty mogą być dołączane lub usuwane dynamicznie
  - polecenie dołączania zawsze kończy się sukcesem (`assert(fakt).`)
  - polecenie usuwania może zakończyć się niepowodzeniem (`retract(fakt).`)

```
assert(mammal(rat)).
assert(mammal(bear)).
assert(fish(salmon)).

mammal(bear).
yes
fish(rat).
no

fish(X).
X = salmon
mammal(X).
X = rat;
X = bear.
```

Zadanie pytania

- Zapytania wprowadza się po znaku zachęty `?-`
- Zapytania mają postać termów
- Jeśli istnieje fakt pasujący do celu zapytania, wtedy zapytanie kończy się sukcesem i generowana jest odpowiedź `yes`
- Jeśli nie istnieje fakt pasujący do celu zapytania, zapytanie kończy się porażką i odpowiedzią `no`.
- Jeśli zapytanie zawiera zmienną, znajdowane i wypisywane są wszystkie znalezione dla niej dopasowania (po znaku `;`)

```
room(kitchen).
room('dinning room').
room(cellar).
?- room(X).
X = kitchen ;
X = 'dinning room' ;
X = cellar.
```

Reguły

- Służą do tworzenia nowych predykatów
  - zazwyczaj z użyciem zmiennych i rekurencją, która jest realizacją jakiegoś algorytmu
- Przykład
  - Dla wszystkich X i Y
    - X jest w łańcuchu pokarmowym Y-ka jeśli:
    - Y zjada X lub
    - Y zjada jakieś Z i X jest w łańcuchu pokarmowym Z-ta

```
food_chain(X,Y) :- eats(Y,X).
food_chain(X,Y) :- eats(Y,Z), food_chain(X,Z).
```

Dopasowanie

- Dwa termy są dopasowane jeśli:
  - są identyczne lub dla zmiennych w obu termach można znaleźć takie obiekty, że po ich podstawieniu w miejsce zmiennych termy stają się identyczne, a inaczej mówiąc:
- Inaczej mówiąc:
  - Jeśli S i T są stałymi, to S pasuje do T tylko jeśli oba są tym samym obiektem
  - Jeśli S jest zmienną, a T czymkolwiek, to pasują do siebie i S staje się instancją T.
  - Jeśli S i T są złożone, to zostaną dopasowane tylko jeśli
    - S i T mają ten sam główny funktor i taką samą liczbę komponentów i
    - wszystkie odpowiadające sobie komponenty pasują do siebie
    - Końcowy wynik dopasowania określony jest przez dopasowanie komponentów

    `course(N,S,95)` pasuje do `course(X,fall,G)`
    `course(N,S,95)` nie pasuje do `course(Y,M,996)`
    `course(X)` nie pasuje do `semester(X)`

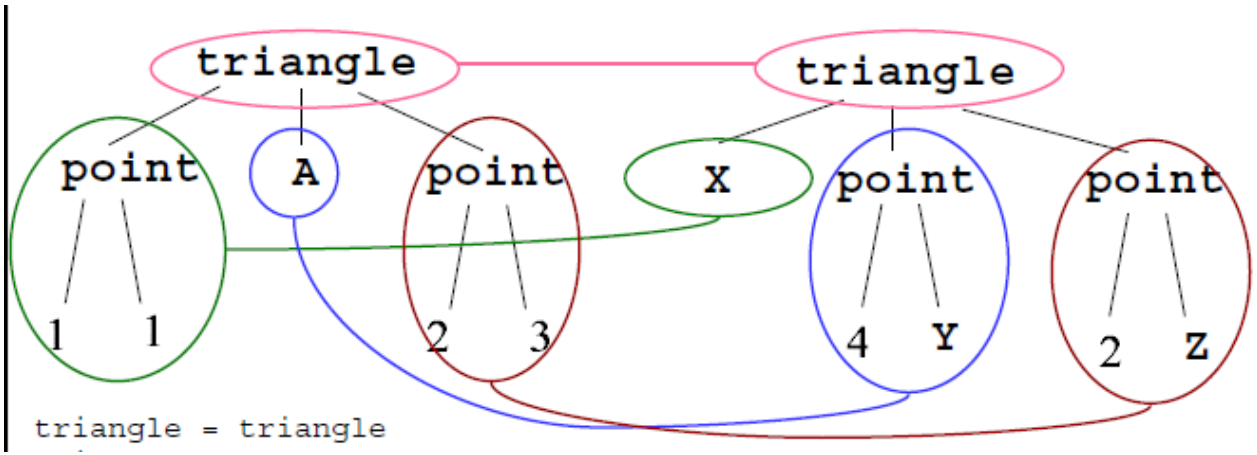
  - jeśli dopasowanie zakończy się sukcesem, zwracany wynik jest zawsze jak najbardziej ogólny

```
course(N,M,85) = course(N1,fall,G).
N = N1
M=fall
G=85
```

Przykład dopasowania

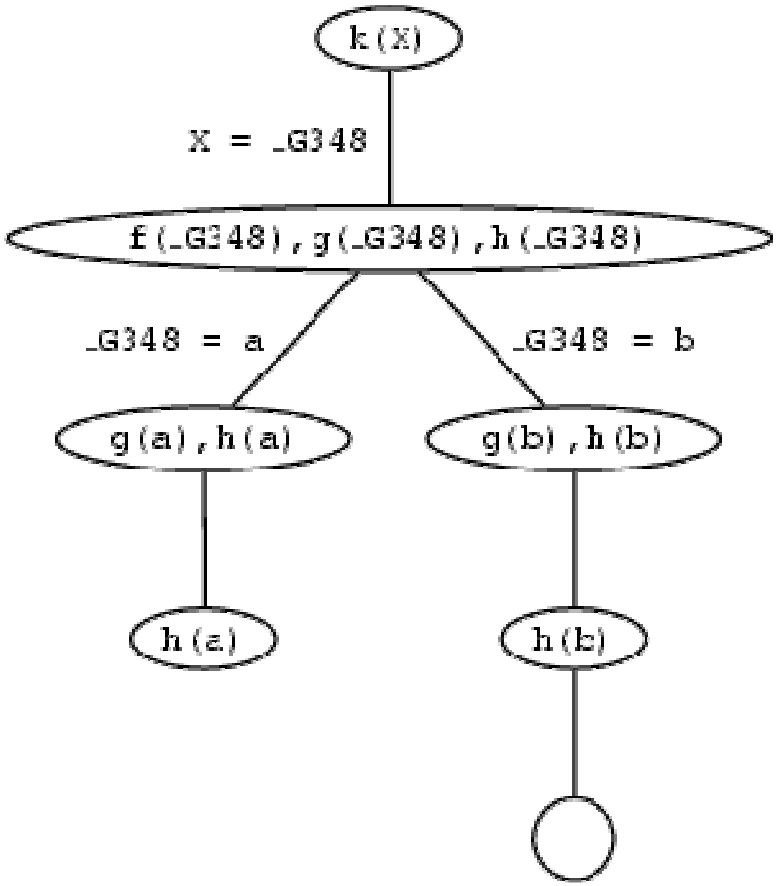
```
point(1,1).
seg( point(1,1), point(2,3) ).
triangle( point(4,2), point(6,4), point(7,1) ).

?- triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y),
  point(2,Z)).
A = point(4,Y)
X = point(1,1)
Z = 3
```

```
triangle = triangle
point(1,1) = X
A = point(4,Y)
point(2,3) = point(2,Z)
```

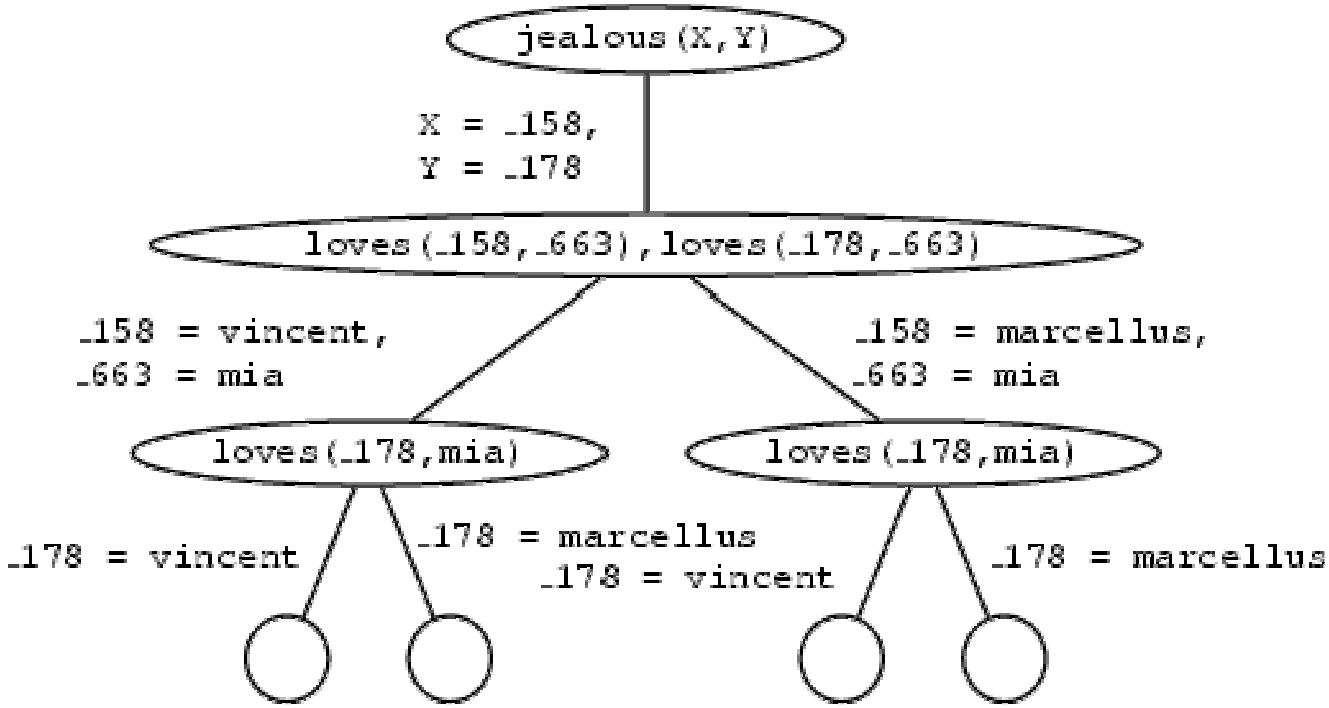


Przykład dopasowania

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

?- k(X).

X = b.
```



```
loves(vincent,mia).
loves(marcellus,mia).
jealous(X,Y) :- loves(X,Z),loves(Y,Z).

?- jealous(X,Y).

X = vincent,
Y = vincent ;
X = vincent,
Y = marcellus ;
X = marcellus,
Y = vincent ;
X = marcellus,
Y = marcellus.
```



Arytmetyka

- Predefiniowane operatory podstawowych operacji arytmetycznych

  `+, -, *, /, mod`
- Jeśli nie zażądano wprost, operatowy traktowane są jak każda inna relacja

```
?- X = 1 + 2.
X=1+2
```
- Wymuszenie wykonania operacji zapewnia użycie słówka `is`

```
?- X is 1 + 2.
X=3
```

- `A is B` (gdzie A i B mogą być czymkolwiek) wymusza dokonanie oszacowania wartości `B`, po którym znajdowane jest dopasowanie otrzymanego rezultatu (liczby) do `A`

- Użycie operatorów porównania również wymusza dokonanie oszacowanie wartości

```
?- 145 * 34 > 100.
true.
```

Operatory logiczne

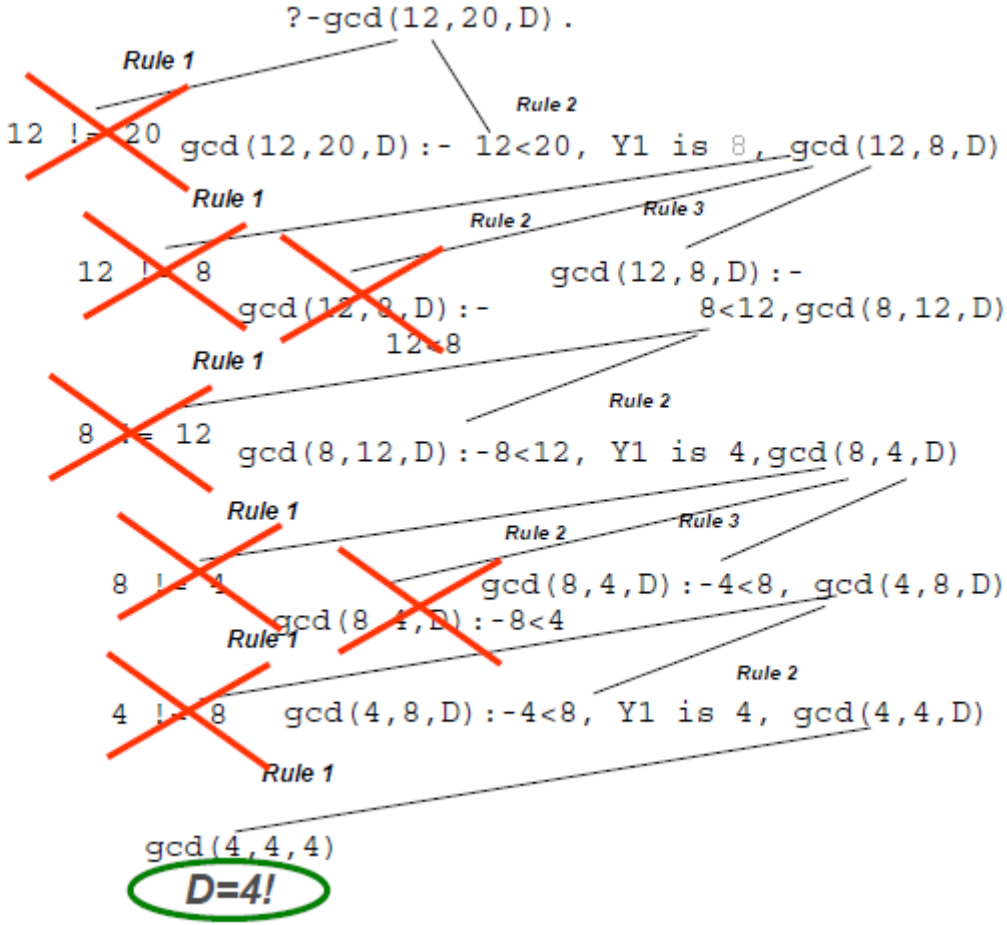
- `X > Y, X < Y, X >= Y, X =< Y`
- `X =:= Y` (równe wartości `X` i `Y`), `X =\= Y` (różne wartości `X` i `Y`)
  - operator `=:=` wymusza dokonanie oszacowanie wartości, jednak w odróżnieniu od `=` nie powoduje dopasowania

```
?- 1 + 2 =:= 2 + 1.
 true.
?- 1 + 2 = 2 + 1.
 false.
?- 1 + A = B + 2.
 A = 2
 B = 1
?- 1 + A =:= B + 2.
 ERROR: =:=/2: Arguments are not sufficiently instantiated
```

Przykład największego wspólnego dzielnika

- Dla danych `X` i `Y` `gcd` `D` obliczyć można następująco:
  - jeśli `X` i `Y` są równe to `D` równa się `X`.
  - jeśli `X < Y` to `D` jest `gcd X` i `(Y-X)`.
  - jeśli `Y < X` to `D` jest `gcd` Y i `(X-Y)`.

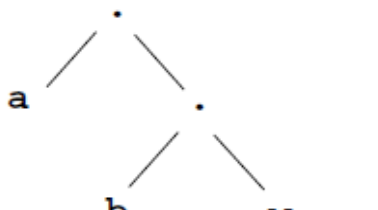
```
gcd(X,X,X).

gcd(X,Y,D) :-
   X < Y,
   Y1 is Y - X,
   gcd(X,Y1,D).

gcd(X,Y,D) :-
   Y < X,
   gcd(Y,X,D).
```



Listy

- Są to sekwencje elementów w wyróżnioną głową i ogonem
- Synonimy:

```
[tom, jerry]          .(tom, .(jerry, [])
[a | tail]            .(a, tail)
[a,b,c] = [a | [b,c]] = [a,b | [c]] = [a,b,c | []]
```



- Elementami listy mogą być listy i termy (w tym termy złożone)

```
[a, [1, 2, 3], tom, 1995, date(1,may,1995) ]
```

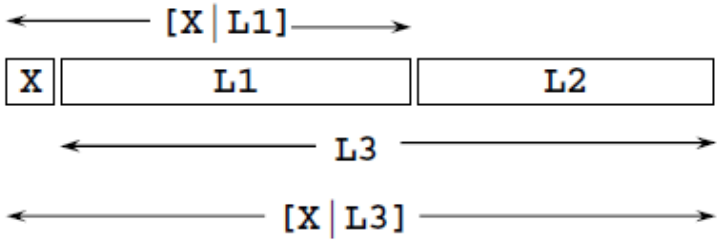
Operacje na listach

- Przynależność
  - `member( X, L)` jeśli `X` należy do `L`.

```
member(X, [X | Tail]).
member(X, [Head | Tail]) :-
member(X, Tail).
```

- Złączenie
  - conc(L1, L2, L3) jeśli L3 jest złączeniem L1 i L2.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
conc(L1, L2, L3).
```



Przykłady operacji na listach

```
?- conc( [a,b,c], [1,2,3], L).
L = [a,b,c,1,2,3]
?- conc( L1, L2, [a,b,c] ).
L1 = [],
L2 = [a, b, c] ;
L1 = [a],
L2 = [b, c] ;
L1 = [a, b],
L2 = [c] ;
L1 = [a, b, c],
L2 = [] ;
false.
?- conc( Before, [4|After], [1,2,3,4,5,6,7]).
Before = [1, 2, 3],
After = [5, 6, 7] ;
false.
?- conc(_, [Pred, 4, Succ |_], [1,2,3,4,5,6,7]).
Pred = 3;
Succ = 5.
```

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
conc(L1, L2, L3).
```

Przykłady operacji na listach

- Definicja członkostwa za pomocą złączenia

```
member1(X, L) :-
conc(_, [X|_], L).
```

- Dodanie elementu na początek

```
add(X, L, [X|L]).
```

- Usuwanie jednego elementu

```
del(X, [X|Tail], Tail).
del(X, [Y|Tail], [Y|Tail1]) :-
del(X, Tail, Tail1).
```

- Usuwanie można użyć do wstawiania

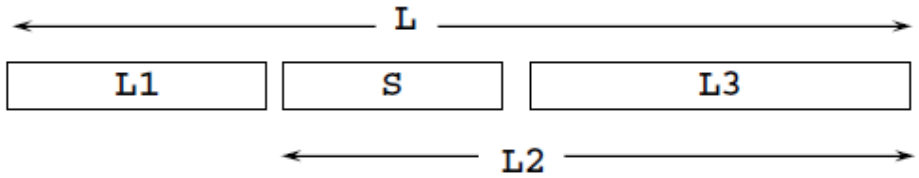
```
?- del(a, L, [1,2,3]).
L = [a,1,2,3];
L = [1,a,2,3];
L = [1,2,a,3];
L = [1,2,3,a];
false.
```

- Wstawianie za pomocą usuwania

```
insert(X,List,BiggerList) :- del(X, BiggerList,
 List).
```

- Podlista

```
sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).
```




```
?- sublist(S, [a,b,c]).
S = [];
S = [a];
...
S = [b,c];
...
```

```
permutation([], []).
permutation([X|L], P) :-
permutation(L, L1),
insert(X, L1, P).

?- permutation( [a,b,c], P).
P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
...

permutation2([], []).
permutation2(L, [X|P]) :-
del(X, L, L1),
permutation2(L1, P).
```

- Długość listy (nadpisanie wbudowanego predykatu)

```
length([], 0).
length([_|Tail],N) :-
length(Tail, N1),
N is 1 + N1.

?- length([a,b,[c,d],e], N).
N = 4
?- length(L,4).
[_5, _10, _15, _20] ;
..... ?
```