Metody Inżynierii Wiedzy
Systemy wnioskujące, automatyzacja wnioskowania - wykład 12

Adam Szmigielski
aszmigie@pjwstk.edu.pl
materiały: $ftp(public) : //aszmigie/MIW$

Paradygmat programowania w logice

Algorytm = logika + kontrola

Historia

- Robert Kowalski - w późnych latach 60-tych wykazał, że dowody logiczne mogą być wspomagane obliczeniowo,
- Colmerauer - w wczesnych latach 70-tych rozwinął pierwszą wersję prologa,
- Warren - środek lat 70 powstała efektywna wersja prologa,
- W 1987 powstaje pierwsza wersja SWI-Prolog.

Algorytm = logika + kontrola

Alternatywne spojrzenie na algorytm. Logika i kontrola algorytmu mogą być analizowane oddzielnie.

Logika:

- Określa znaczenie, cel algorytmu,
- Logikę problemu realizuje programista,
- Modeluje się problemy opisane logiką pierwszego rzędu,

Kontrola:

- Określa złożoność algorytmu,
- Kontrola, złożoność algorytmu może być zmieniona bez zmiany logiki,
- Kontrole może osobno od logiki realizować komputer
- W znacznej części kontrola jest realizowana przez automatyczne wnioskowanie.

Czym jest prolog ?

- **Prolog** jest deklaratywnym językiem obliczeń symbolicznych. Oznacza to, że wyraża się w nim jedynie “Co”, a nie “Jak” chce się policzyć oraz obliczenia przeprowadza się na danych symbolicznych.
- Obliczenia takie różnią się od obliczeń numerycznych tym, że nie wyliczają one wartości liczbowych ale obiekty reprezentowane w postaci napisów o szczególnej strukturze (tzw. termów).

Proramowanie w prologu

Programowanie w prologu składa się z następujących faz:

- *deklarownaie* pewnych faktów o obiektach wraz z relacjami,
- *zdefiniowanie* reguł dotyczących obiektów i relacji między nimi,
- *zadawanie pytań* o obiektach i relacji między nimi.

Cechy języka prolog

- Prolog może modelować logiki pierwszego rzędu,
- Sortowania dokonuję się poprzez backtracking i unifikacją,
- Podstawowymi danymi danych są termy (terminy),
- Zmienne są nieznane bez możliwości lokalizacji,
- Prolog nie wyróżnia wejścia i wyjścia,
- Prolog operuje na relacjach/predykatach.

Składnie Prologa

- Prolog ma bardzo prostą składnie: *stałe*, *zmienne*, *termy* odnoszące się do *obiektów*
- Zmienne pisze się z dużej litery
- Funktory nie mają wartości, lecz używa się ich do nazewnictwa.
- Predykaty określają relację pomiędzy obiektami
- Klauzule określają prawdziwą składnie
- Odpowiedzi na zapytania są tworzone poprzez dopasowania nagłówka klazuly. Może być więcej niż jedna odpowiedź

Zmienne

- Zmienne są tworzone przez prologa dla obiektów które nie możemy nazwać,
- Zmienne zaczynają się wielką literą lub podkreśleniem

Przykłady:
X, *Object01*, *ShoppingList*, *_xy*.

Stałe

- Stałe nie mogą zmieniać swojej wartości.
- Zapisuje się je jako ciągi liter, cyfr i znaku podkreślenia rozpoczynające się od małej litery.
- Szczególnymi stałymi w Prologu są liczby całkowite i rzeczywiste.
- Dowolny ciąg znaków może być w prologu również stałą ale w tym celu należy go ująć między cudzysłowy.

Przykłady:
ala 123 'Ala ma Asa'

Termy w prologu

Termami w prologu mogą być:

- **integer** - liczba całkowitoliczbowa,
- **atom** - tekst zaczynający się z małej litery (np. ala, f_16) lub tekst w cudzysłowiu ('ala ma kota'),
- **zmienna** - zaczynająca się z wielkiej litery lub podkreślenia (_) (np. X, Zet, $\_k2$),
- **predykat** - określający relacje pomiędzy obiektami,
- **termy złożony** .

Fakty w prologu

Są podobne jak w relacyjnych bazach danych. Składnia faktu wygląda następująco:

$$predname(arg_1, arg_2, ...arg_N).$$

gdzie:

- *predname* - nazwa predykatu (zaczynająca się z małej litery),
- $arg_1, \ldots$ - argumenty, N - liczba argumentów,
- . - oznacza koniec klauzuli.
- Argumentami mogą być dowolne terminy w prologu,
- Zbiór *faktów* oraz *klauzul* stanowi w Prologu *bazę wiedzy*

Fakty - termy proste

- Argumenty są nazwami obiektów,
- Predykaty są to nazwy relacji,
- *Fakty* w wyrażają relację pomiędzy obiektami np.:
  - mebel(stol, 24, 35).
  - mieszkaniec('Jan').
  - rodzenstwo('Jan', ala).
  - drzwi(kuchnia, pokoj).

Fakty- termy złożone

- **Term złożony** ma następującą postać:

$$f(T_1, T_2, ..., T_n),$$

gdzie f jest nazwą n-argumentowego predykatu, natomiast T_i, dla $i = 1, 2, ..., n$, są termami.

- Nazwa predykatu łączy termy w jeden term złożony.
- np. term złożony *odcinek*:

$$odcinek(poczatek(0, 0), koniec(1, 1)).$$

jako argumenty termy proste $poczatek(0, 0)$ i $koniec(1, 1)$

Klauzula Horna

- Klauzulę Horna można przedstawić w postaci:

$$(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \longrightarrow B$$

lub w notacji prologa:

$$B : -A_1, A_2, A_3$$

- Zmieniając implikację otrzymujemy alternatywę literałów

$$\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee B$$

w której tylko jeden literał jest niezanegowanym atomem.

Klauzula w prologu

- Klauzula w prologu ma następującą składnie:

$$head :- body.$$

  gdzie:
  - *head* - definicja predykatu (podobnie jak fakt),
  - $:-$ - "neck" symbol, (odpowiednik Jeśli)
  - *body* - jedno lub więcej zapytań,
- *Klauzula* może być rozumiane w dwojaki sposób:
  - *deklaratywnie* - jako fragment wiedzy, umożliwiający deklarowanie nowych terminów,
  - *proceduralnie* - jako sekwencje kroków umożliwiających osiągnięcie rozwiązania.

Klauzula w prologu - przykład

Klauzula:

$$grandmother(X, Y) :- mother(X, Z), parent(Z, Y).$$

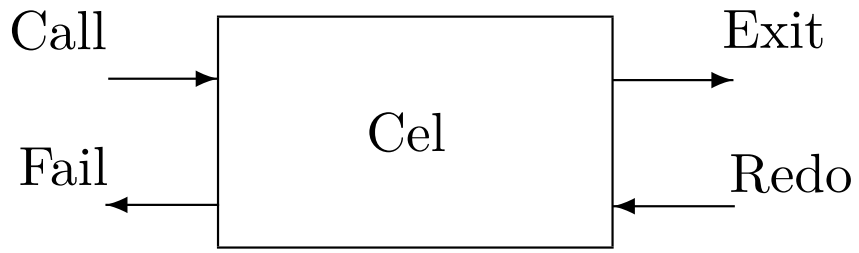
może być rozumiana jako:

- *deklaratywnie* mówi, że “W każdym świecie, w którym X jest matką Z i Z jest rodzicem Y, X jest babcią Y”.
- *proceduralnie* oznacza to, że ”aby znaleźć X który jest babką Y, najpierw należy znaleźć takie Z które jest matką X i sprawdzić czy Z jest rodzicem Y”

Uruchomienie programu

- Programy są zapisywane w pliku o rozszerzeniu $*.pl$ np. $prog.pl$,
- Prolog jest interpreterem,
- Program można wgrać poleceniem $['prog.pl']$.

Blok przeszukiwań



- *call* - program zaczyna szukać rozwiązania,
- *exit* - program sygnalizuje osiągnięcie celu,
- *fail* - program stwierdza, że przeszukiwanie klauzul zakończyło się niepowodzeniem.
- *redo* - program ponawia szukanie celu,

Zapytania

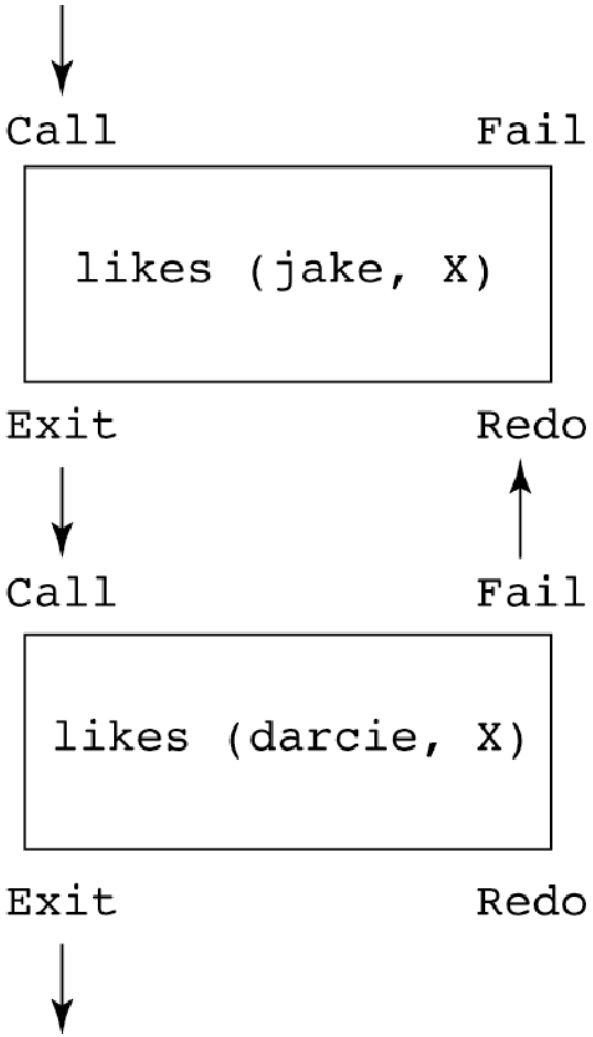
Baza:

```
likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).
```

Zapytanie:

?- likes(jake,X), likes(darcie,X).

```
[trace]  ?- likes(jake,X), likes(darcie,X).
   Call: (8) likes(jake, _G1123) ? creep
   Exit: (8) likes(jake, chocolate) ? creep
   Call: (8) likes(darcie, chocolate) ? creep
   Fail: (8) likes(darcie, chocolate) ? creep
   Redo: (8) likes(jake, _G1123) ? creep
   Exit: (8) likes(jake, apricots) ? creep
   Call: (8) likes(darcie, apricots) ? creep
   Exit: (8) likes(darcie, apricots) ? creep
X = apricots.
```


Call
Fail
likes (jake, X)
Exit
Redo
Call
Fail
likes (darcie, X)
Exit
Redo

Przeszukiwanie wstecz - backtracking

```
female(jane).
female(mary).
rich(mary).
loves(john, X) :- female(X), rich(X).
```

Zapytanie: loves(john, X).

```
[trace]  ?- loves(john,X).
   Call: (7) loves(john, _G2399) ? creep
   Call: (8) female(_G2399) ? creep
   Exit: (8) female(jane) ? creep
   Call: (8) rich(jane) ? creep
   Fail: (8) rich(jane) ? creep
   Redo: (8) female(_G2399) ? creep
   Exit: (8) female(mary) ? creep
   Call: (8) rich(mary) ? creep
   Exit: (8) rich(mary) ? creep
   Exit: (7) loves(john, mary) ? creep
X = mary.
```

Wykonanie programu

- Odpowiedź, czy z zawartych w programie klauzul i faktów wynika pozytywna odpowiedź na postawione przez użytkownika pytanie,
- W trakcie wykonania programu za zmienne w zapytaniu podstawiane są konkretne obiekty, dla których odpowiedź jest prawdziwa,
- W trakcie poszukiwań prolog może tworzyć własne, dodatkowe zmienne.

Arytmetyka w prologu

- Prolog zawiera podstawowe operacje arytmetyczne,
- zapytania proste:

```
?- 2>1.
true.

?- 3<2.
false.

?- 3=\=3.
false.

?- 3==3.
true.
```

- Pytania złożone:

```
?- 2>1,4<3.
false.

?- 2>1;4<3.
true
```

Przecinek odpowiada koniunkcji, a średnik alternatywie.

Wyrażenia arytmetyczne

Dostępne są następujące predykaty dwuargumentowe porównujące wartości wyrażeń arytmetycznych:

=:=	równe
=\=	różne
<	mniejsze
=<	mniejsze równe
>	większe
>=	większe równe

Predykat “is”- Podstawienie w prologu

- Predykat “is” jest odpowiednikiem instrukcji podstawienia, X is <wyrażenie arytmetyczne>
- Predykat ten wymaga, by po prawej stronie *is* znajdowało się wyrażenie arytmetyczne bez wolnych zmiennych natomiast jeśli po lewej stronie:
  - jest wolna zmienna, to obliczona wartość wyrażenia zostanie podstawiona pod tą zmienną,
  - jest liczba, to zostanie ona porównana z wartością wyrażenia.

```
?- X is 1, Y is X+1.
X = 1,
Y = 2.

?- 2 is 4-3.
false.

?- 2 is 4-2.
true.
```

Unifikacja terminów

- Ważną operacją na termach jest **unifikacja**.
- Dla zadanych dwóch termów T_1 i T_2 szuka ona wyrażeń jakie trzeba podstawić pod zmienne występujące w T_1 i T_2 by po podstawieniu termy stały się identyczne.
- Jeśli takiego podstawienia nie ma, to unifikacja kończy się niepowodzeniem,
- Do unifikacji służy dwuargumentowy predykat $=$

Unifikacja - przykłady

```
?- a=a.
true.

?- a=b.
false.

?- a=X.
X = a.

?- g(A,B,C,D) = g(a,f(A,A), f(B,B),f(C,C)).
A = a,
B = f(a, a),
C = f(f(a, a), f(a, a)),
D = f(f(f(a, a), f(a, a)), f(f(a, a), f(a, a))).

?- 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.

?- [1,2,3,2,1]=[A,B,C,B,A].
A = 1,
B = 2,
C = 3.
```

Rekurencja w prologu

- Prolog dopuszcza definicje rekurencyjne,
- Rekurencja jest metodą przeszukiwania bazy wiedzy,
- Przykład: Relacja *bycia przodkiem* może być zdefiniowana jako:
  - przodek(X, Z):- rodzic(X, Z).
  - przodek(X, Z) :- rodzic(X, Y), rodzic(Y, Z).
  - przodek(X, Z) :- rodzic(X, Y1), rodzic(Y1, Y2), rodzic(Y2, Z).
  - ...
- Ten sposób definiowania relacji *bycia przodkiem* jest zbyt długi i może być nieskuteczny.

- Prostszy, elegancki sposób zdefiniowania relacji *bycia przodkiem* odwołujący się do rekurencji.

  X jest przodkiem Z jeśli X jest rodzicem Y i Y jest przodkiem Z

- Co można zapisać w prologu jako:
  przodek(X,Z) :- rodzic(X,Y), przodek(Y,Z).

- Kompletny opis relacji *bycia przodkiem* zawiera dwie reguły:
  przodek(X,Z) :- rodzic(X,Z).
  przodek(X,Z) :- rodzic(X,Y),przodek(Y,Z).

Instrukcje sterujące

- **Instrukcja cut !**
  - Predykat *cut* został wprowadzony dla zwiększenia efektywności obliczeń.
  - Wspomaga on kontrolę nad procesem *backtracking*
  - Predykat *cut* oznacza się wykrzyknikiem !
  - *Cut* kończy procedurę przeszukiwania.
- **Instrukcja fail**
  - Użycie *fail* jako instrukcji wymusza dalsze przeszukiwanie,
- **Instrukcje warunkowe typu if**

Instrukcja odcięcia cut !

Mechanizm odcięcia pozwala zaniechać nawracania przez Prolog do wcześniej dokonanych wyborów.

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

Po wydaniu zapytania $a(X, Y)$. Prolog udzieli następujących odpowiedzi:

```
?- a(X,Y).
X = d Y = g ;
X = d Y = h ;
X = d Y = i ;
No
```

Instrukcja odcięcia cut !

Odcięcie będzie miało także wpływ na stosowanie innych reguł. Załóżmy, że pewien program ma dwie reguły dla celu a.

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
b(e).
b(f).
c(g).
d(f).
```

Prolog nie będzie brał pod uwagę drugiej reguły. W tym przypadku zapytanie $a(X)$. zwróci

```
?- a(X).
No
```

Instrukcja odcięcia cut ! - przykład

Obliczamy sumę dwóch liczb. Jeśli pierwsza z nich jest mniejsza od 10, w przeciwnym przypadku wynik powinien mieć taką samą wartość jak druga liczba.

```
suma(X,Y,Z):-X<10, Z is X+Y.
suma(X,Y,Z):-Z is Y.
```

Wynik działania programu dla wywołania - podaje dwa wyniki

```
?- suma(2,9,X).
X = 11 ;
X = 9.
```

Modyfikacja programu:

```
suma(X,Y,Z):-X<10, Z is X+Y,!.
suma(X,Y,Z):-Z is Y.
```

Daje prawidłowy wynik:

```
X = 11.
```

Instrukcja warunkowe if "->"

warunek -> klauzula gdy prawda ; klauzula gdy fałsz

Przykład:

```
?- 3>2 -> write('wyrazenie prawdziwe') ; write('wyrazenie falszywe').
wyrazenie prawdziwe
true.

?- 3<2 -> write('wyrazenie prawdziwe') ; write('wyrazenie falszywe').
wyrazenie falszywe
true.
```

Dynamiczne usuwanie i dodawanie termów

- Prolog umożliwia dynamiczne usuwanie i dodawanie termów w czasie korzystania z programu,
- Termy dynamiczne muszą być wcześniej określone jako

  :- dynamic term/N.

  gdzie N jest liczbą argumentów termu.
- Można tego dokonać posługując się następującymi predykatami:
  - *assert(A)* dodaje term A do bazy wiezy,
  - *retract(A)* usuwa term A,
  - *retractall(A)* usuwa wszystkie termy A.

Przykład usuwania i dodawania termów

```
:- dynamic gdzie/1. %deklaracja zmiennej dynamicznej
gdzie(kuchnia).
drzwi(kuchnia, pokoj).

jestprzejscie(Miejsce):- gdzie(X), drzwi(X, Miejsce).

przejdz(Miejsce) :- jestprzejscie(Miejsce), retract(gdzie(X)),

asserta(gdzie(Miejsce)),write('Przeszlem do '), write(Miejsce), nl.
```

Listy

- Lista jest uporządkowaną sekwencją elementów.
- Elementem listy może być dowolny termin - stała, zmienna, struktura lub inna lista,
- Lista może być pusta (oznaczamy ją jako [])
- Lista niepusta zawiera dwa składniki - nagłówek i ogon,

Opis listy

- Aby rozgraniczyć nagłówek i ogon w prologu istnieje specjalna notacja:
  $[X|Y]$ oznacza, że X jest nagłówkiem a Y ogonem.
- Listy mogą być opisane w różne sposoby:
  $.(a, .(b, .(c, [])))$,
  $[a|[b|[c|[]]]]$,
  $[a|[b|[c]]]$,
  $[a|[b, c]]$,
  $[a, b, c]$,
  $[a, b|[c]]$

Predykaty list

- **Predykat member** można używać zarówno do sprawdzania czy dany term jest elementem listy, jak również do wybierania kolejnych elementów listy:

```
?- member(2,[1,3,2,7]).
true .

?- member(X,[1,3,2,7]).
X = 1 ;
X = 3 ;
X = 2 ;
X = 7.
```

- **Predykat select** służy przede wszystkim do wybierania elementów z listy ale można również go używać do wstawiania elementów na listę:

```
?- select(X,[1,2,3],POZOSTALE).
X = 1,
POZOSTALE = [2, 3] ;
X = 2,
POZOSTALE = [1, 3] ;
X = 3,
POZOSTALE = [1, 2] ;
false.

?- select(1,LISTA,[a,b,c]).
LISTA = [1, a, b, c] ;
LISTA = [a, 1, b, c] ;
LISTA = [a, b, 1, c] ;
LISTA = [a, b, c, 1] ;
false.
```

- **Predykat append** zasadniczo służy do łączenia dwóch list w jedną. Można go również używać do rozrywania listy na dwa kawałki a nawet do sprawdzania czy term jest elementem listy:

```
?- append([1,2,3],[a,b,c],X).
X = [1, 2, 3, a, b, c].

?- append(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 2, 3],
Y = [] ;
false.
```

- **Predykat reverse** odwraca elementy na liście. Można również sprawdzać nim czy jedna lista ma w odwrotnej kolejności elementy niż druga:

```
?- reverse([1,2,3,4],X).
X = [4, 3, 2, 1].

?- reverse([k,a,j,a,k],[k,a,j,a,k]).
true.

?- reverse([a,1,a],[o,1,a]).
false.
```

- **Predykat length** oblicza długość listy ale można wykorzystać go do generowania list o zadanej długości:

```
?- length([a,b,c,d],X).
X = 4.

?- length(X,3).
X = [_G8504, _G8507, _G8510].
```

Sortowanie list - przykład

Sortowanie bąbelkowe. Implementacja algorytmu w Prologu wygląda następująco.

```
bSort(List,Sorted) :- swap(List,List1), !, bSort(List1,Sorted).
bSort(Sorted,Sorted).
swap([X,Y|Rest],[Y,X|Rest]) :- X > Y.
swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
```

Wynik działania programu:

```
?- bSort([4, 5, 1],V).
V = [1, 4, 5].
```

Integracja prologa z innymi językami

Prolog posiada rozbudowane interfejsy języka JAVA, C, C++

MIW PROLOG projekt

1. Sporządź mapę mieszkania lub jego części (np. pokoju, pokoju wraz z korytarzem). Na mapę nanieś istniejące meble, sprzęty itp.
2. Zaproponuj język opisujący zaistniałą na planie sytuację (wybór faktów).
3. Zaproponuj definicję pojęć przestrzennych opisujących związki przestrzenne, np. pomiędzy, obok, na itp.
4. Wprowadź elementy dynamiczne umożliwiające dodawanie nowych lub zmianę istniejących faktów.
5. Zaproponuj zasady nawigacji, umożliwiające poruszanie się w opisywanej przestrzeni. Zrealizuj konkretne zadanie nawigacji - np. przemieszczenie się blisko telewizora.

6. Elementy, które powinny znaleźć się w projekcie:
   - Termy, termy złożone (fakty),
   - Klauzule (definicje nowych pojęć),
   - Elementy dynamiczne (dodawanie i usuwanie termów w trakcie działania programu),
   - Rekurencja,
   - Unifikacja, cięcie,
   - Listy (operacje na listach).

Elementy dodatkowe

1. Integracja z innymi językami programowania (C, C++, Java).
2. **Istnieje możliwość wyboru indywidualnych projektów (np. implementacja do gier) po wcześniejszym uzgodnieniu z osobą prowadzącą ćwiczenia.**