

Prolog: Podstawy programowania logicznego

Programy do uruchomienia Prologa

- <https://wiki.ostrowski.net.pl/prolog/> na bazie <https://tau-prolog.org/>
- <https://swish.swi-prolog.org/>

Wstęp

Prolog (Programming in Logic) to jeden z najstarszych i najbardziej znanych języków programowania deklaratywnego. Został stworzony w latach 70-tych XX wieku przez Alaina Colmeraura i Phillipa Rousselota. Jest to język, w którym programista opisuje problem w postaci faktów, reguł i zapytań, a system komputerowy samodzielnie wyciąga wnioski i szuka rozwiązań.

Zastosowania Prologa

Prolog jest szeroko stosowany w dziedzinach, które wymagają rozwiązywania problemów logicznych, takich jak:

- Sztuczna inteligencja (AI): Prolog jest używany do tworzenia systemów eksperckich, systemów wnioskowania i robotyki, gdzie konieczne jest podejmowanie decyzji na podstawie dostępnych danych.
- Analiza i przetwarzanie języka naturalnego: Prolog znajduje zastosowanie w przetwarzaniu języka naturalnego (NLP), ponieważ potrafi analizować i przetwarzać struktury językowe.
- Bazy danych: Prolog może być używany do tworzenia baz danych i systemów wyszukiwania, w których relacje między danymi są wyrażone za pomocą faktów i reguł.
- Rozwiązywanie problemów matematycznych: Dzięki swojej logice, Prolog jest wykorzystywany do rozwiązywania problemów związanych z teorią grafów, szukaniem ścieżek, algorytmami planowania i innymi problemami kombinatorycznymi.

Drzewo Genealogiczne



To jest fakt w Prologu, który opisuje relację „rodzic”. W tym przypadku:

```
rodzic(jozef,jacek) oznacza, że Józef  
jest rodzicem Jacka.
```

Fakty w Prologu są podstawowymi stwierdzeniami, które są uznawane za prawdziwe. Każdy fakt



składa się z predykatu (np. rodzic) i argumentów (np. józef i jacek), które stanowią dane związane z tym predykatem.

Predykaty i reguły:

% fakty

```
małżeństwo(jacek,iza).
małżeństwo(andrzej,anna).
małżeństwo(jan,krystyna).
małżeństwo(józef,halina).
małżeństwo(cezary,cecylia).
małżeństwo(henryk,hanna).
małżeństwo(darek,dorota).
```

% dzieci(jacka i iza)

```
rodzic(jacek,krzys).
rodzic(iza,krzys).
rodzic(jacek,ola).
rodzic(iza,ola).
rodzic(iza,julek).
```

%dzieci anrzej i anna

```
rodzic(andrzej,jas).
rodzic(anna,jas).
```

% dzieci jana i krystyny

```
rodzic(krystyna,iza).
rodzic(krystyna,jagoda).
rodzic(krystyna,andrzej).
rodzic(krystyna,jurek).
rodzic(jan,iza).
rodzic(jan,jagoda).
```

```
rodzic(jan,andrzej).
rodzic(jan,jurek).
```

% dzieci cezary i cecylia

```
rodzic(cecylia,halina).
rodzic(cezary,halina).
```

% dzieci dorota i darek

```
rodzic(dorota,danuta).
rodzic(dorota,nadzieja).
rodzic(darek,danuta).
rodzic(jacek,nadzieja).
```

% dzieci jozefa i haliny

```
rodzic(halina,jacek).
```

W Prologu `\+` oznacza negację. Jest to operator, który sprawdza, czy wyrażenie jest fałszywe. Możesz to rozumieć jako zapytanie „Czy to nie jest prawda?”. Operator `\+` działa jak negacja logiczna w innych językach programowania.

Przykład użycia negacji:

```
\+ rodzic(józef, jacek).
```

To zapytanie sprawdza, czy Józef nie jest rodzicem Jacka. Jeśli fakt `rodzic(józef, jacek)` nie jest zapisany w bazie danych, wynik będzie prawdą (ponieważ negacja fałszywego stwierdzenia daje prawdę). Jeśli taki fakt istnieje, wynik będzie fałsz.



Negacja w Prologu działa w następujący sposób:

- `\+ A` będzie prawdą, jeśli `A` jest fałszywe.
- `\+ A` będzie fałszem, jeśli `A` jest prawdą.

Przykłady:

- Jeśli mamy fakt `rodzic(józef, jacek)`, zapytanie `\+ rodzic(józef, jacek)` zwróci fałsz.
- Jeśli mamy zapytanie `\+ rodzic(krzysztof, jacek)`. (które nie jest zapisane jako fakt w bazie), to zwróci prawdę.

```
rodzic(halina,hanna).
rodzic(halina,piotrek).

rodzic(jozef,jacek).
rodzic(jozef,hanna).
rodzic(jozef,piotrek).

rodzic(adam,julek).

kobieta(iza).
kobieta(jagoda).
kobieta(ola).
kobieta(krystyna).
kobieta(halina).
kobieta(hanna).
kobieta(cecylia).
kobieta(dorota).
kobieta(anna).
kobieta(nadzieja).

%reguły

męczyzna(X) :- \+
kobieta(X).
ojciec(X,Y) :- rodzic(X,Y),
męczyzna(X).
matka(X,Y):- rodzic(X,Y),
kobieta(X).
dziecko(X,Y) :- rodzic(Y,X).

wnuk(X,Y) :- dziecko(D,Y),
dziecko(X,D).

rodzeństwo_n(X,Y) :-
    matka(M,Y),
    matka(M,X),
    ojciec(O,Y),
    ojciec(O,X), X \= Y.

rodzeństwo_p(X,Y) :-
    matka(M,Y),
    matka(M,X),
    ojciec(O1,Y),
    ojciec(O2,X),
    X \= Y,
    O1 \= O2.

rodzeństwo_p(X,Y) :-
    matka(M1,Y),
    matka(M2,X),
    ojciec(O,Y),
```

```
ojciec(0,X),
X \= Y,
M1 \= M2.

rodzeństwo(X,Y) :-
    rodzeństwo_n(X,Y);
    rodzeństwo_p(X,Y).

siostra(X,Y) :-
    rodzeństwo(X,Y), kobieta(X).
brat(X,Y) :-
    rodzeństwo(X,Y),
    mężczyzna(X).

mąż(X,Y) :-
    mężczyzna(Y),
    małżeństwo(X,Y),
    kobieta(Y).

żona(X,Y) :-
    kobieta(X),
    małżeństwo(X,Y),
    mężczyzna(Y).
```

Przykładowe zapytania:

```
% Zapytania do modelu Prolog
% Komentarze wyjaśniające, co każde zapytanie robi

% Pytanie 1: Sprawdzamy, czy Jacek i Iza są małżeństwem.
% Zapytanie sprawdza fakt w bazie danych
małżeństwo(jacek, iza). % Oczekiwana odpowiedź: tak (True)

% Pytanie 2: Sprawdzamy, kto jest ojcem Krzysia.
% Zapytanie testuje regułę "ojciec"
ojciec(X, krzys). % Oczekiwana odpowiedź: X = jacek

% Pytanie 3: Sprawdzamy, kto jest matką Oli.
% Zapytanie testuje regułę "matka"
matka(X, ola). % Oczekiwana odpowiedź: X = iza

% Pytanie 4: Kto jest dzieckiem Jacka?
% Zapytanie testuje regułę "dziecko"
dziecko(X, jacek). % Oczekiwana odpowiedź: X = krzys ; X = ola ; X =
nadzieja

% Pytanie 5: Sprawdzamy, czy Krzysiu i Ola to rodzeństwo.
% Zapytanie testuje regułę "rodzeństwo_n" (rodzeństwo na podstawie tych
samyh rodziców)
rodzeństwo_n(krzys, ola). % Oczekiwana odpowiedź: tak (True)
```

```
% Pytanie 6: Kto jest wnukiem Jana?
% Zapytanie testuje regułę "wnuk"
wnuk(X, jan). % Oczekiwana odpowiedź: X = iza ; X = jagoda ; X = andrzej ;
X = jurek

% Pytanie 7: Sprawdzamy, czy Iza i Jagoda są siostrami.
% Zapytanie testuje regułę "siostra"
siostra(iza, jagoda). % Oczekiwana odpowiedź: tak (True)

% Pytanie 8: Kto jest mężem Anny?
% Zapytanie testuje regułę "mąż"
mąż(X, anna). % Oczekiwana odpowiedź: X = andrzej

% Pytanie 9: Kto jest żoną Jana?
% Zapytanie testuje regułę "żona"
żona(X, jan). % Oczekiwana odpowiedź: X = krystyna

% Pytanie 10: Kto jest bratem Izy?
% Zapytanie testuje regułę "brat"
brat(X, iza). % Oczekiwana odpowiedź: X = andrzej ; X = jurek

% Pytanie 11: Kto jest ojcem Jasem?
% Zapytanie testuje regułę "ojciec"
ojciec(X, jas). % Oczekiwana odpowiedź: X = andrzej

% Pytanie 12: Sprawdzamy, czy Jacek i Halina są małżeństwem.
% Zapytanie testuje fakt w bazie danych
małżeństwo(jacek, halina). % Oczekiwana odpowiedź: nie (False)

% Pytanie 13: Kto jest ojcem Jagody?
% Zapytanie testuje regułę "ojciec"
ojciec(X, jagoda). % Oczekiwana odpowiedź: X = jan

% Pytanie 14: Kto jest matką Krystyny?
% Zapytanie testuje regułę "matka"
matka(X, krystyna). % Oczekiwana odpowiedź: brak odpowiedzi, ponieważ nie
mamy takiego faktu

% Pytanie 15: Kto jest rodzeństwem Haliny?
% Zapytanie testuje regułę "rodzeństwo"
rodzeństwo(X, halina). % Oczekiwana odpowiedź: X = cezary ; X = cecylia

% Pytanie 16: Kto jest siostrą Izy?
% Zapytanie testuje regułę "siostra"
siostra(X, iza). % Oczekiwana odpowiedź: X = jagoda
```

Zagadka kryminalna



W Prologu `\=` oznacza nierówność.

To jest operator porównania, który sprawdza, czy dwie wartości (lub zmienne) są różne.

W tym przypadku:

`X \= 0` oznacza, że `X` jest różne od 0.

Predykaty i reguły:

% Fakty

```
osoba(tomasz, 55, stolarz).
osoba(krzysztof, 25,
piłkarz).
osoba(krzysztof, 25,
rzeźnik).
osoba(piotr, 25, złodziej).
osoba(anna, 39, chirurg).
```

```
romans(anna, piotr).
romans(anna, krzysztof).
romans(agnieszka, piotr).
romans(agnieszka, tomasz).
```

```
zamordowana(agnieszka).
prawdopodobnie_zamordowana(a
gnieszka, kij_golfowy).
prawdopodobnie_zamordowana(a
gnieszka, łom).
```

```
pobrudzony(tomasz, krew).
pobrudzony(agnieszka, krew).
pobrudzony(krzysztof, krew).
pobrudzony(krzysztof,
błoto).
pobrudzony(piotr, błoto).
pobrudzony(anna, krew).
```

```
posiada(tomasz,
sztuczna_noga).
posiada(piotr, rewolwer).
```

```
podobne_obrażenia(sztuczna_n
oga, kij_golfowy).
podobne_obrażenia(noga_od_st
ołu, kij_golfowy).
podobne_obrażenia(łom,
```

W Prologu `_` jest tzw. anonimową zmienną.

Oznacza to, że nie interesuje nas wartość tej zmiennej i nie będziemy jej używać w dalszej części programu. Prolog przyjmuje ją, ale nie przypisuje jej żadnej konkretnej wartości.

W Prologu możesz używać `_`, gdy nie zależy ci na wynikach tej zmiennej, np. w przypadku:

```
motyw(X, zazdrość) :-
    kobieta(X),
    zamordowana(0),
    romans(0, M),
    romans(X, M),
    X \= 0.
```



W przypadku powyższym, zmienna `M` w regule `romans(0, M)` jest używana, ponieważ sprawdzamy `romans` między `0` a `M`, ale jeśli w innym przypadku nie chcemy używać jakiejś zmiennej, zapisujemy ją jako `_`:

```
romans(_, _). % przykładowy zapis,
który oznacza, że nie zależy nam na
wartościach
```

To mówi Prologowi: „Przyjmij wszystkie możliwe wartości, ale nie będziemy ich używać ani sprawdzać”.

Zatem `_` pełni rolę zmiennej, której wartości nie będziemy wykorzystywać w dalszej logice.

```
kij_golfowy).
podobne_obrazenia(nożyczki,
nóż).
podobne_obrazenia(but_piłkar
ski, kij_golfowy).

% Fakty o płci
męczyzna(piotr).
męczyzna(krzysztof).
męczyzna(tomasz).

kobieta(anna).
kobieta(agnieszka).

% Reguły
posiada(X, but_piłkarski) :-
osoba(X, _, piłkarz).
posiada(X, piłka) :-
osoba(X, _, piłkarz).
posiada(X, nóż) :- osoba(X,
_, rzeźnik).
posiada(X, nóż) :- osoba(X,
_, chirurg).
posiada(X, nożyczki) :-
osoba(X, _, chirurg).
posiada(X, łom) :- osoba(X,
_, złodziej).
posiada(X, noga_od_stołu) :-
osoba(X, _, stolarz).

posiada(X,
narzędzie_zbrodni) :-
    posiada(X, rewolwer);
    posiada(X, nóż);
    posiada(X, kij_golfowy);
    posiada(X, nożyczki);
    posiada(X,
but_piłkarski);
    posiada(X,
noga_od_stołu);
    posiada(X,
sztuczna_noga);
    posiada(X, łom).

podejrzany(X) :-
    zamordowana(0),
prawdopodobnie_zamordowana(0
, Y),
    podobne_obrazenia(N, Y),
    posiada(X, N).
```

```
motyw(X, zazdrość) :-  
    mężczyzna(X),  
    zamordowana(0),  
    romans(0, X).  
  
motyw(X, zazdrość) :-  
    kobieta(X),  
    zamordowana(0),  
    romans(0, M),  
    romans(X, M),  
    X \= 0.  
  
motyw(X, pieniądze) :-  
    mężczyzna(X),  
    osoba(X, _, złodziej).  
  
morderca(X) :-  
    podejrzany(X),  
    zamordowana(0),  
    motyw(X, _),  
    pobrudzony(0, S),  
    pobrudzony(X, S).  
  
motyw_mordercy(M) :-  
    morderca(X),  
    motyw(X, M).
```

Odpowiedz na pytania:

Kto posiada narzędzia zbrodni ?

```
posiada(X, narzędzie_zbrodni).
```

Kto jest podejrzany o morderstwo?

```
podejrzany(X).
```

Jakie motywy zbrodni miały poszczególne osoby?

```
motyw(X, M).
```

Kto jest mordercą?

```
morderca(X).
```

Jaki motyw miał morderca?

```
motyw_mordercy(M).
```

Struktury listowe w języku Prolog

Struktury listowe w języku **Prolog** stanowią podstawowy mechanizm reprezentacji zbiorów danych. Listy są strukturami rekurencyjnymi, co umożliwia ich elastyczne przetwarzanie.

Definicja listy

Lista w Prologu to uporządkowany zbiór elementów, zapisany w nawiasach kwadratowych i oddzielony przecinkami:

```
[el1, el2, el3]
```

Lista może być również pusta:

```
[]
```

Operacje na listach

Dostęp do elementów

Lista może być rozbita na głowę (pierwszy element) i ogon (reszta listy):

```
[H|T]
```

1. **H** – głowa listy (head)
2. **T** – ogon listy (tail)

Przykład:

```
?- [H|T] = [1,2,3].  
H = 1,  
T = [2,3].
```

Sprawdzanie przynależności

Operator `member/2` sprawdza, czy element należy do listy:

```
member(X, [a,b,c]).
```

Łączenie list

Operator `append/3` służy do łączenia dwóch list:

```
append([1,2], [3,4], Result).  
Result = [1,2,3,4].
```

Długość listy

Predykat `length/2` służy do określania długości listy:

```
length([a,b,c], N).  
N = 3.
```

Rekurencja na listach

Ze względu na rekurencyjny charakter list, większość algorytmów operujących na listach korzysta z rekurencji.

Przykład sumowania elementów listy:

```
sum([], 0).  
sum([H|T], S) :-  
    sum(T, S1),  
    S is H + S1.
```

Lista jako struktura danych

Listy mogą zawierać zmienne, inne listy, a także złożone struktury:

```
[a,b], [c,d]
```

Mogą także być niedomknięte (ang. open-ended lists):

```
[1,2|X]
```

Jest to przydatne przy konstruowaniu dynamicznych list.

Wypisywanie elementów listy

Predykat wypisujący każdy element listy w osobnej linii:

```
print_list([]).  
print_list([H|T]) :-  
    write(H), nl,  
    print_list(T).
```

Przykład użycia:

```
?- print_list([apple, banana, cherry]).
```

```
apple  
banana  
cherry
```

Wyodrębnianie pierwszego elementu listy

Aby uzyskać pierwszy element listy, możemy użyć dopasowania wzorców (pattern matching) z użyciem operatora `|`.

Przykładowy predykat:

```
first_element([H|_], H).
```

1. **[H|_]** – dopasowuje listę, gdzie `H` to pierwszy element, a `_` ignoruje resztę.
2. **H** – zmienna zwracająca pierwszy element.

Przykład użycia:

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

Wyodrębnianie drugiego elementu listy

Aby uzyskać drugi element listy, możemy użyć dopasowania wzorców, omijając pierwszy element.

Predykat:

```
second_element([_, Second|_], Second).
```

1. **[_, Second|_]** – ignoruje pierwszy element (`_`), przypisuje drugi do zmiennej `Second`, a resztę ignoruje.

Przykład użycia:

```
?- second_element([x, y, z], X).  
X = y.
```

Wyodrębnianie ostatniego elementu listy

Aby uzyskać ostatni element listy, możemy wykorzystać rekurencyjne dopasowanie wzorców.

Predykat:

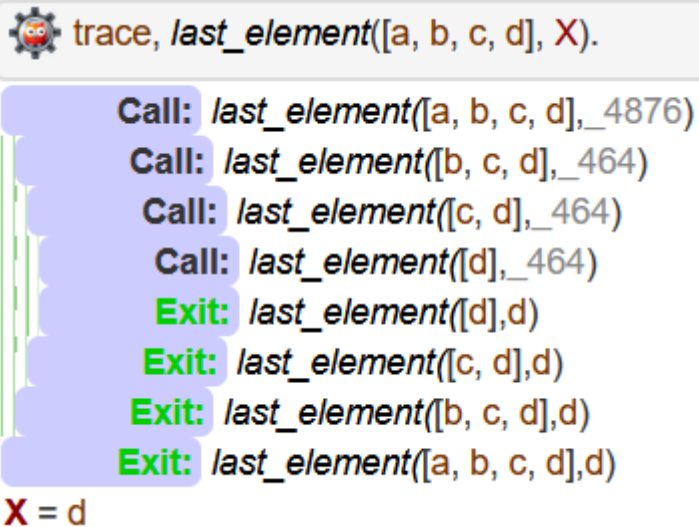
```
last_element([X], X).  
last_element([_|T], X) :-  
    last_element(T, X).
```

1. **[X]** – dopasowanie jednoelementowej listy (ostatni element).
2. **[_T]** – rekurencyjne przeszukiwanie ogona listy, aż zostanie tylko jeden element.

Przykład użycia:

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

Wywołania rekurencyjne:



```
trace, last_element([a, b, c, d], X).
Call: last_element([a, b, c, d],_4876)
Call: last_element([b, c, d],_464)
Call: last_element([c, d],_464)
Call: last_element([d],_464)
Exit: last_element([d],d)
Exit: last_element([c, d],d)
Exit: last_element([b, c, d],d)
Exit: last_element([a, b, c, d],d)
X = d
```

Znajdowanie elementu w liście

Aby sprawdzić, czy dany element znajduje się w liście, można skorzystać z wbudowanego predykatu ``member/2``, albo zdefiniować własną wersję.

Wersja oparta na rekurencji:

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

1. **[X|_]** – dopasowanie, jeśli pierwszy element listy to szukany element.
2. **[_|T]** – rekurencyjne przeszukiwanie ogona listy.

Przykład użycia:

```
?- in_list([1, 2, 3, 4], 3).
true.

?- in_list([a, b, c], d).
false.
```

Alternatywa: użycie wbudowanego predykatu ``member/2``:

```
?- member(3, [1,2,3,4]).
```

```
true.
```

Sprawdzanie, czy lista jest uporządkowana rosnąco

Predykat sprawdzający, czy elementy listy numerycznej rosną lub są równe (nie maleją):

```
sorted_asc([]).  
sorted_asc([_]).  
sorted_asc([X, Y | T]) :-  
    X <= Y,  
    sorted_asc([Y | T]).
```

1. `[]` i `[_]` — lista pusta lub jednoelementowa jest uporządkowana.
2. `[X, Y | T]` — sprawdzamy, czy pierwszy element jest mniejszy lub równy drugiemu, a następnie rekurencyjnie resztę listy.

Przykład użycia:

```
?- sorted_asc([1, 2, 2, 4, 5]).  
true.  
  
?- sorted_asc([1, 3, 2, 4]).  
false.
```

Wstawianie elementu do listy uporządkowanej rosnąco

Predykat, który wstawia element `X` do posortowanej listy rosnącej `List`, zwracając nową listę `Result`, również uporządkowaną rosnąco:

```
insert_sorted(X, [], [X]).  
insert_sorted(X, [H|T], [X,H|T]) :-  
    X <= H.  
insert_sorted(X, [H|T], [H|R]) :-  
    X > H,  
    insert_sorted(X, T, R).
```

1. Jeśli lista jest pusta, nowa lista to `[X]`.
2. Jeśli `X` jest mniejsze lub równe pierwszemu elementowi `H`, wstawiamy `X` przed `H`.
3. W przeciwnym razie rekurencyjnie wstawiamy `X` w ogon listy.

Przykład użycia:

```
?- insert_sorted(3, [1, 2, 4, 5], Result).  
Result = [1, 2, 3, 4, 5].
```