Prolog: Logic programming basics

Programs to run Prolog

- https://wiki.ostrowski.net.pl/prolog/ based https://tau-prolog.org/
- https://swish.swi-prolog.org/

Introduction

Prolog (Programming in Logic) is one of the oldest and best-known declarative programming languages. It was created in the 1970s by Alain Colmeraur and Phillip Rousselot. It is a language in which the programmer describes a problem in terms of facts, rules and queries, and the computer system independently draws conclusions and searches for solutions.

Applications of Prolog

Prolog is widely used in areas that require solving logical problems, such as:

- Artificial Intelligence (AI): Prolog is used to create expert systems, inference systems and robotics, where it is necessary to make decisions based on available data.
- Natural language analysis and processing: Prolog is used in natural language processing (NLP) because it can analyse and process language structures.
- Databases: Prolog can be used to create databases and search systems in which relationships between data are expressed using facts and rules.
- Mathematical problem solving: Thanks to its logic, Prolog is used to solve problems related to graph theory, path-finding, planning algorithms and other combinatorial problems.

Genealogical tree

This is a fact in Prolog that describes the 'parent' relationship. In this case:

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

Facts in Prolog are basic statements that are assumed to be true. Each fact consists of a predicate (e.g. parent) and arguments (e.g. jozef and jacek), which are the data associated with that predicate.

In the Prologue \+ stands for negation. It is an operator that checks whether an expression is false. You can understand it as asking „Is this not true?”. The operator \+ works like a logical negation in other programming languages.

An example of the use of negation:

```
\+ rodzic(jozef, jacek).
```

This query checks if jozef is not the parent of jacek. If the fact parent(jozef, jacek) is not stored in the database, the result will be true (because negation of a false statement yields true). If such a fact exists, the result will be false.

Negation in Prolog works as follows:

- \+ A will be true if A is false.
- \+ A A will be false if A is true.

Examples:

- If we have the fact `parent(jozef, jacek)`, the query `\+ ■ + parent(jozef, jacek).` will return false.
- If we have the query `\+ parent(krzysztof, jacek).` (which is not stored as a fact in the database), it will return true.

Predicates and rules:

```
% fakty
małżeństwo(jacek,iza).
małżeństwo(andrzej,anna).
małżeństwo(jan,krystyna).
małżeństwo(jozef,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).
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

mężczyzna(X) :- \+ kobieta(X).
ojciec(X,Y) :- rodzic(X,Y), mężczyzna(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(O,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).
```

Example queries:

```
% 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
samych 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
```

Crime mystery

In the Prologue \= denotes an inequality.
This is a comparison operator that checks whether two values (or variables) are different.
In this case:
X \= 0 means that X is different from 0.

Predicates and rules:

```
% 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).
```

In the Prologue _ is a so-called anonymous variable. This means that we are not interested in the value of this variable and will not use it in the rest of the program. Prolog accepts it, but does not assign any specific value to it.

In Prolog you can use _, when you do not care about the results of this variable, e.g. in the case of:

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

In the case above, the variable M in the romance(O, M) rule is used because we are checking the romance between O and M, but if we otherwise do not want to use a variable, we write it as _:

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

This tells Prolog: „Accept all possible values, but

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

% Fakty o płci
mężczyzna(piotr).
mężczyzna(krzysztof).
mężczyzna(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);
```

we will not use or check them”.

So _ acts as a variable whose value we will not use in further logic.

```
    posiada(X, łom).

podejrzany(X) :-
    zamordowana(O),
prawdopodobnie_zamordowana(O
, Y),
    podobne_obrażenia(N, Y),
    posiada(X, N).

motyw(X, zazdrość) :-
    mężczyzna(X),
    zamordowana(O),
    romans(O, X).

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

motyw(X, pieniądze) :-
    mężczyzna(X),
    osoba(X, _, złodziej).

morderca(X) :-
    podejrzany(X),
    zamordowana(O),
    motyw(X, _),
    pobrudzony(O, S),
    pobrudzony(X, S).

motyw_mordercy(M) :-
    morderca(X),
    motyw(X, M).
```

Answer the questions:

Who owns the tools of the crime ?
`owns(X, tool_of_crime).`

Who is suspected of the murder?
`suspect(X).`

What motives did the individuals have for the crime?
`Motive(X, M).`

Who is the murderer?
`murderer(X).`

What motive did the murderer have?
`motive_murderer(M).`

List structures in Prolog

List structures in the **Prolog** are the primary mechanism for representing data sets. Lists are recursive structures, which allows flexible processing.

Definition of a list

A list in Prolog is an ordered collection of elements, written in square brackets and separated by commas:

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

A list can also be empty:

```
[]
```

List operations

Access to items

A list can be broken down into a head (the first element) and a tail (the rest of the list):

```
[H|T]
```

1. **H** - head of the list (head)
2. **T** - list tail

Example:

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

Affiliation check

The `member/2` operator checks whether an element belongs to a list:

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

Combining lists

The `append/3` operator is used to merge two lists:

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

List length

The predicate `length/2` is used to specify the length of a list:

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

Recursion on lists

Due to the recursive nature of lists, most algorithms that operate on lists use recursion.

An example of summing the elements of a list:

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

List as a data structure

Lists can contain variables, other lists, as well as complex structures:

```
[[notatki:a_b_c_d]]
```

They can also be open-ended lists:

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

This is useful when constructing dynamic lists.

Writing out the elements of a list

A predicate that prints out each list element on a separate line:

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

Example of use:

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

Extracting the first element of a list

To obtain the first element of a list, we can use pattern matching using the operator |.

Example predicate:

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

1. **[H|_]** - matches a list where `H` is the first element and `_` ignores the rest.
2. **H** - variable returning the first element.

Example of use:

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

Extracting the second element of a list

To obtain the second element of the list, we can use pattern matching, bypassing the first element.

Predicate:

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

1. **[_, Second|_]** - ignores the first element (`_`), assigns the second to the variable `Second`, and ignores the rest.

Example of use:

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

Extracting the last element of a list

To extract the last element of a list, we can use recursive pattern matching.

Predicate:

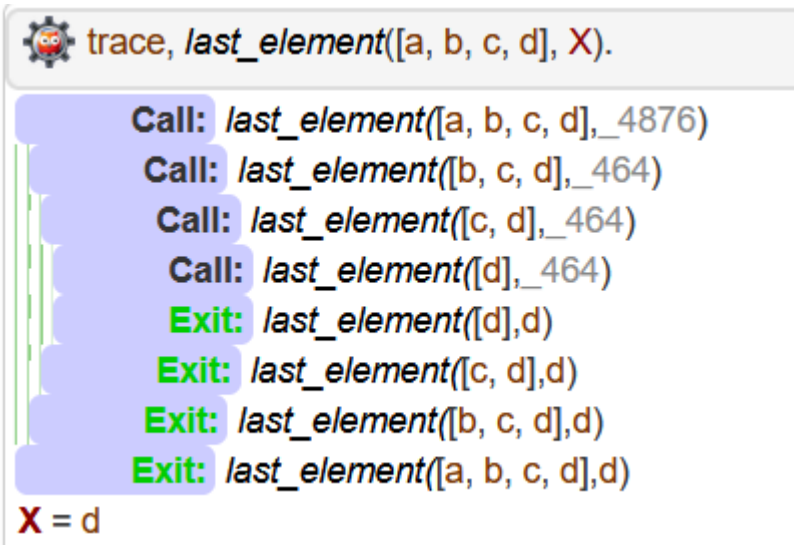
```
last_element([X], X).
last_element([_|T], X) :-
    last_element(T, X).
```

1. **[X]** - matching a one-element list (last element).
2. **[_|T]** - recursive search of the tail of the list until only one element remains.

Example of use:

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

Recursive calls:



Finding an element in a list

To check whether an element is in a list, you can use the built-in `member/2` predicate, or define your own version.

Recursion-based version:

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

1. **[X|_]** - match if the first element in the list is the element being searched for.
2. **[_|T]** - recursive search of the tail of a list.

Usage example:

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

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

Alternative: using the embedded predicate `member/2`:

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

Checking whether the list is in ascending order

A predicate that checks whether the elements of a numeric list are increasing or equal (not decreasing):

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

1. **[]** i **[_]** - empty or one-element list is ordered.
2. **[X, Y | T]** - we check if the first element is less than or equal to the second, and then recursively the rest of the list.

Example of use:

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

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

Inserting an element into an ascending ordered list

Predicate that inserts an element `X` into a sorted ascending list `List`, returning a new list `Result`, also ordered ascending:

```
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. If the list is empty, the new list is `[X]`.

2. If `X` is less than or equal to the first element of `H`, we insert `X` before `H`.
3. Otherwise, we recursively insert `X` in the tail of the list.

Usage example:

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

Sorting a list using the insertion method

Predicate that sorts a numerical list ascendingly using the insertion sort algorithm (*insertion sort*).

Definition:

```
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).

insertion_sort([], []).
insertion_sort([H|T], Sorted) :-
    insertion_sort(T, SortedT),
    insert_sorted(H, SortedT, Sorted).
```

Action description:

1. `insert_sorted/3` - inserts an element into the correct place in the sorted list.
2. `insert_sort/2` - recursively sorts the tail of the list and inserts the current element.

Usage example:

```
?- insertion_sort([4, 1, 3, 2], Sorted).
Sorted = [1, 2, 3, 4].
```

Calculating the length of a list

The predicate `length_list/2` calculates the length of a list - that is, the number of its elements - using recursion.

Definition:

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

1. **[]** - the empty list has a length of 0.
2. **[_|T]** - we ignore the first element and recursively count the length of the tail of the list, increasing the result by 1.

Usage example:

```
?- length_list([a, b, c, d], N).
N = 4.
```

Alternatively, the built-in predicate `length/2` can be used:

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

Summing list elements

The predicate `sum_list/2` calculates the sum of all numeric elements contained in a list.

Definition:

```
sum_list([], 0).
sum_list([H|T], Sum) :-
    sum_list(T, Rest),
    Sum is H + Rest.
```

1. **[]** - the sum of the empty list is 0.
2. **[H|T]** - adds the current element `H` to the sum of the remaining elements of `T`.

Example of use:

```
?- sum_list([1, 2, 3, 4], S).
S = 10.
```

You can also use the built-in predicate `sum_list/2`, which works identically:

```
?- sum_list([1,2,3,4], S).
S = 10.
```

Arithmetic mean of the list elements

The predicate `average_list/2` calculates the arithmetic mean of all numeric elements in a list.

Requires two auxiliary predicates:

- `sum_list/2` - calculates the sum of the elements,
- `length_list/2` - calculates the length of the list.

Definition:

```
sum_list([], 0).
sum_list([H|T], Sum) :-
    sum_list(T, Rest),
    Sum is H + Rest.

length_list([], 0).
length_list([_|T], N) :-
    length_list(T, N1),
    N is N1 + 1.

average_list(List, Avg) :-
    sum_list(List, Sum),
    length_list(List, Length),
    Length > 0,
    Avg is Sum / Length.
```

Action description:

1. First we calculate the sum of the elements of the list.
2. Then we calculate its length.
3. We calculate the average as `Sum / Length`.

Example of use:

```
?- average_list([2, 4, 6, 8], A).
A = 5.0.
```

Note: the predicate checks that the length of the list is greater than zero to avoid dividing by zero.