Wstęp do Sztucznej Inteligencji - rok akademicki 2022/2023

Przed rozpoczęciem pracy z notatnikiem zmień jego nazwę zgodnie z wzorem: `NrAlbumu_Nazwisko_Imie_PoprzedniaNazwa`.

Przed wysłaniem notatnika upewnij się, że rozwiązałeś wszystkie zadania/ćwiczenia.

Temat: Prolog - Programowanie w logice cz. II

Zapoznaj się z treścią niniejszego notatnika czytając i wykonując go komórka po komórce. Wykonaj napotkane zadania/ćwiczenia.

Listy w Prologu

Podstawową strukturą danych w Prologu są listy. Są one strukturami, które są przetwarzane rekurencyjnie. W Prologu lista składa się z `głowy` (pierwszego elementu) oraz `ogona` (pozostałe elementy). Na przykład lista:

```
[3,4,5,7]
```

ma `głowę`, którą jest element 3 oraz `ogon`, który jest listą $[4,5,7]$.

Uwaga: poniższe informacje o tworzeniu list za pomocą "." dotyczą wcześniejszych wersji SWI Prologa. Aktualnie operator ten jest wykorzystywany w innym celu. Przeczytaj więcej:

http://www.swi-prolog.org/pldoc/man?section=ext-lists

http://www.swi-prolog.org/pldoc/man?section=ext-dict-functions

W Prologu do tworzenia listy służy predykat `.` (kropka). Aby więc do zmiennej X zapisać powyższą listę, można napisać:

```
?- X = .(3,[4,5,7]).
X = [3, 4, 5, 7]
```

Należy pamiętać o spacji pomiędzy znakiem równości i kropką, gdyż `=.` ma w Prologu inne znaczenie i jest osobnym predykatem. Listę można również stworzyć poleceniem

```
?- X = [3, 4, 5, 7].
```

Mając taką listę, można uzyskać dostęp do jej `głowy` i `ogona` za pomocą składni:

```
Lista = [Glowa|Ogon].
```

Jeśli `Lista` jest ukonkretniona listą, to `Glowa` zostanie ukonkretniona pierwszym elementem tej listy a zmienna `Ogon` pozostałymi elementami, tj. ogonem.

Wypróbuj polecenie:

```
?- X = [1,2,3,4,5], X = [Glowa|Ogon].
```

Możemy zatem stworzyć listę podając wprost jej głowę oraz ogon, np.:

```
?- G = 3, O=[4,5,6], L=[G|O].
```

`L` zostaje ukonkretniona listą $[3,4,5,6]$ o czterech elementach.

Co jednak otrzymamy, jeśli wykonamy następujące polecenie?

```
?- O = 3, G=[4,5,6], L=[G|O].
```

Tym razem wynik będzie odmienny, otrzymamy listę $L=[[4,5,6]\vee 3]$ zawierającą dwa elementy: pierwszy to lista z trzema elementami $[4,5,6]$, a drugi to 3.

```

```

Ćwiczenie 1:

Co się stanie jak napiszemy?

```
?- G = 3, O=[4,5,6], L=[O|G].
```

Zostanie Wypisane G = 3, L = [[4, 5, 6]|], O = [4, 5, 6]

Ostatnim elementem listy jest zawsze lista pusta `[]`. Zatem listę z jednym tylko elementem można stworzyć tak:

```
?- L = [3].
```

lub tak:

```
?- L = [3|[]].
```

ale NIE tak:

```
?- L = [3,[]].
```

Jak widać elementami listy nie muszą być tylko liczby, mogą nimi by także inne listy lub cale struktury. Na przykład:

```
[ [5,4], [5,6], [3,4] ].
[kasia, lubi, jarka].
[autor(clive, barker), autor(bertrand, russell) ].
```

Mając listę L można ukonkretnić więcej niż jedną zmienną pierwszymi elementami listy, np.:

```
?- L = [1,2,3,4,5], L=[X, Y, Z| Reszta].
```

Ćwiczenie 2:

Co się stanie jeśli napiszemy?

```
?- L = [2], L=[X|Y].
```

Zostanie wypisane L = [2], X = 2, Y = []

A co jeśli:

```
?- L = [2], L=[X, Y|Z].
```

Zostanie wypisane False

Ćwiczenie 3:

Wypróbuj jaki efekt będzie miało uzgadnianie następujących list:

```
[] = X.

X = X

[] = [X|Y].

False

[a,b,c,d] = [H|T].

H = a, T = [b,c,d]

[1,2,3] = [X,Y,Z].

X = 1, Y = 2, Z = 3

[marta] = [G|O].

G = marta, O = []

[A, B, C] = [jan, lubi, piwo].

A = jan, B = lubi, C = piwo
```

```
[1,2,3] = [G|[2,3]].

G = 1

[[marta, A], B] = [[X, lubi], koty].

A = lubi, B = koty, X = marta

[[1,2],[3,4],[5,6]] = [X|Y].

X = [1,2], Y = [[3,4], [5,6]]

[7,6] = [6,X].

False

[kobieta(marta), kobieta(jola)] = [X|Y].

X = kobieta(marta), Y = [kobieta(jola)]
```

Operacje na listach

Do operacji na listach w Prologu często wykorzystuje się rekurencję. Na przykład aby wypisać każdy element listy w nowej linijce, możemy zdefiniować predykat `pisz_liste`, będący predykatem rekurencyjnym tzn. odwołującym się do samego siebie:

```
pisz_liste([]).
pisz_liste(Lista) :- Lista = [Glowa|Ogon], write(Glowa), nl,
pisz_liste(Ogon).
```

Predykat `nl` powoduje przejście do nowej linii i nigdy nie zawodzi. Predykat `write` wypisuje na ekran i również nigdy nie zawodzi.

Powyższy predykat `pisz_liste` można zdefiniować krócej:

```
pisz_liste2([]).
pisz_liste2([Glowa|Ogon]) :- print(Glowa), nl, pisz_liste2(Ogon).
```

W obu wersjach predykat `pisz_liste` odwołuje się do samego siebie w drugiej klauzuli. Warunek końcowy jest umieszczony w pierwszej klauzuli tego predykatu. Jest to konieczne jako że Prolog przeszukuje bazę danych od „góry do dołu”.

Ćwiczenie 4:

Wypróbuj działanie powyższego predykatu `pisz_liste`.

Członkostwo listy

Do sprawdzania czy dany element jest w liście służy predykat wbudowany `member`:

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

`member(E, L)` sprawdza czy element E jest na liscie L. Można zdefiniować samemu taki predykat:

```
member2(E, [X|_]) :- E=X.
member2(E, [_|O]) :- member2(E,O).
```

Pierwsza klauzula sprawdza czy E jest równe pierwszemu elementowi listy. Jeśli tak, E należy do L, jeśli nie, sprawdzany jest warunek drugi, czyli sprawdzane jest czy E należy do pozostałej części listy. Zwróć uwagę na wykorzystanie zmiennej anonimowej.

Pierwszą klauzule można zapisać krócej i otrzymać ostatecznie:

```
member2(E, [E|_]).
member2(E, [_|O]) :- member2(E,O).
```

Uwaga: Jeśli `member` nie jest rozpoznawane przez interpreter, wykonaj:

```
?- ensure_loaded(library(lists)).
```

lub użyj:

```
?- L = [1,2,3,4], lists:member(3, L).
```

Ćwiczenie 5:

Sprawdź jakie odpowiedzi otrzymasz na zapytanie:

```
?- member(X, [1,2,3,4,5]).
```

Łączenie list

Za łączenie list odpowiada predykat `append(L1,L2,L3)`. Na końcu listy `L1` dopisuje listę `L2` i ukonkretnia `L3` powstała listą.

Ćwiczenie 6:

Wypróbuj zapytania:

```
?- append([1,2,3], [4,5,6], L3).
?- append(L1, [4,5,6], [1,2,3,4,5,6]).
?- append(L1, L2, [1,2,3,4,5,6]).
```

Można zdefiniować samemu taki predykat:

```
append2([], L, L).
append2([X|L1], L2, [X|L3]) :- append2(L1, L2, L3).
```

Pierwszy warunek mówi, że każda lista dołączona do listy pustej daje tę samą listę append2([], L , L). Drugi predykat opisuje regułę łączenia list w listę wynikową, tzn. mówi, że jeśli lista pierwsza jest niepusta, to głowa listy pierwszej staje się głową listy wynikowej, natomiast ogonem listy wynikowej jest ogon listy pierwszej złączony z listą drugą. Jak widać również tu wykorzystujemy rekurencję. Odcinając za każdym razem głowę listy pierwszej w końcu dojdziemy do warunku końcowego.

Poprawne warunki końcowe rekurencji

Wywołania rekurencyjne muszą w pewnym momencie natrafić na warunek końcowy, tak by wszystkie wcześniejsze wywołania mogły się zakończyć powrotem. Kluczowe jest by wykonać to prawidłowo.

Rozważmy przykład kopiowania listy. Zdefiniujmy predykat `copy(L1, L2)`, który kopiuje zawartość `L1` do `L2`.

```
copy([G|O],[G|L]) :- copy(O, L).
copy([],_).
```

Wywołanie rekurencyjne znajduje się w pierwszej klauzuli. Warunek stopu (druga klauzula) wydaje się na pierwszy rzut oka prawidłowy: jeśli `L1` jest pusta, a `L2` dowolna, zwróć prawdę (bez wywołania rekurencyjnego). Jednak po uruchomieniu dostaniemy następujący wynik.

```
?- copy([1,2,3], X).
X = [1, 2, 3|_3482].
```

Widać, że na końcu listy wynikowej jest jakaś anonimowa niezainicjalizowana zmienna, rekurencja nie została zakończona prawidłowo (można powiedzieć, że ogon listy pozostał otwarty).

Wypróbuj następującą definicję predykatu `copy`.

```
copy([G|O],[G|L]) :- copy(O, L).
copy([],[]).
```

Po uruchomieniu widać, że działa poprawnie.

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

Podobnie rozpatrzmy przykład predykatu `wybierz_mniejsze(E, L, W)`, który kopiuje do listy `W` elementy z listy `L`, jeśli są one mniejsze niż `E`. Przeanalizuj poniższe klauzule. Czy są one prawidłowe?

```
wybierz_mniejsze(E, [G|O], [G|L]) :- G < E, wybierz_mniejsze(E,O,L).
wybierz_mniejsze(E, [G|O], L) :- wybierz_mniejsze(E, O, L).
wybierz_mniejsze(_, [], _).
```

Po uruchomieniu:

```
?- wybierz_mniejsze(3, [1,2,3,4,5], X).
X = [1, 2|_3534] .
```

Widać problem analogiczny do wcześniejszego. Czy poniższa zmiana w definicji `wybierz_mniejsze` wystarczy? Zwróć uwagę, że za `E` nie podstawiamy w trzeciej klauzuli listy pustej `[]`, gdyż `E` nie jest listą.

```
wybierz_mniejsze(E, [G|O], [G|L]) :- G < E, wybierz_mniejsze(E,O,L).
wybierz_mniejsze(E, [G|O], L) :- wybierz_mniejsze(E, O, L).
wybierz_mniejsze(_, [], []).
```

Po uruchomieniu:

```
?- wybierz_mniejsze(3, [1,2,3,4,5], X).
X = [1, 2] .
```

Definicja `wybierz_mniejsze` zdaje się być prawidłowa, ale czy na pewno? Co jeśli poprosimy o inne możliwe odpoweidzi naciskając `;`?

```
?- wybierz_mniejsze(4, [1,2,3,4,5], X).
X = [1, 2, 3] ;
X = [1, 2] ;
X = [1, 3] ;
X = [1] ;
X = [2, 3] ;
X = [2] ;
X = [3] ;
X = [] ;
false.
```

Okazuje się, że pojawia się wiele odpowiedzi, z których tylko pierwsza jest dla nas prawidłowa. Oznacza to, że jednak źle sterujemy mechanizmem nawracania w prologu. Sprawdźmy następującą definicję.

```
wybierz_mniejsze(E, [G|O], [G|L]) :- G < E, wybierz_mniejsze(E,O,L).
wybierz_mniejsze(E, [G|O], L) :- G >= E, wybierz_mniejsze(E, O, L).
wybierz_mniejsze(_, [], []).
```

Po uruchomieniu:

```
?- wybierz_mniejsze(4, [1,2,3,4,5], X).
X = [1, 2, 3] ;
false.
```

Teraz działa prawidłowo.

Dzielenie listy

Predykat `dziel` przyjmuje cztery elementy:

```
dziel(E, L, L1, L2)`.
```

Dzieli on listę L na L1 i L2 w zależności od tego czy kolejne elementy L są mniejszy czy większe od E.

Zadanie 1:

Przeanalizuj i przetestuj podane klauzule tego predykatu. Czy są poprawne? Weź pod uwagę infromacje z wcześniejszego punktu. Napraw błędy i wpisz poniżej poprawną definicję i przykład uruchomienia.

```
dziel(E, [G|O], [G|L1], L2) :- G =< E, dziel(E,O,L1,L2).
dziel(E, [G|O], L1, [G|L2]) :- dziel(E,O,L1,L2).
dziel(_, [], _, _).
```

YOUR ANSWER HERE

dziel(_, [], [], []).

Usuwanie z listy

Predykat `usun(E,L,W)` usuwa wszystkie wystąpienia E w liście L i wynikową listę zapisuje w liście W.

```
usun(_, [], []).
usun(E, [E|O], W) :- !, usun(E,O,W).
usun(E, [X|O1], [X|O2]) :- usun(E,O1,O2).
```

Zastępowanie elementu X w L1 elementem Y i zapis do listy L2

```
zastap(_, [], _, []).
zastap(X, [X|L1], Y, [Y|L2]) :- !, zastap(X, L1, Y, L2).
zastap(X, [A|L1], Y, [A|L2]) :- !, zastap(X, L1, Y, L2).
```

Znajdowanie elementu listy o podanym numerze.

Predykat `znajdz(N, L, X)` znajduje N-ty element listy L i ukonkretnia nim X.

```
znajdz(_,[],_) :- !, fail.
znajdz(1,[X|_], X) :- !.
znajdz(N, [_|O], X) :- M is N-1, znajdz(M, O, X).
```

Ostatni element listy.

Predykat `ostatni(X, L)` ukonkretnia X ostatnim elementem listy L.

```
ostatni(X, [X]).
ostatni(X, [_|Y]) :- ostatni(X, Y).
```

Co ciekawe, powyższy predykat można również zdefiniować za pomocą predykatu `append`:

```
ostatni(X, L) :- append(_, [X], L).
```

Sprawdzanie czy dana lista jest początkiem innej.

Predykat `prefiks(L1, L2)` sprawdza czy L1 jest początkiem L2.

```
prefiks([], _).
prefiks([X|L1], [X|L2]) :- prefiks(L1, L2).
```

Długość listy.

Predykat `dlugosc(L, N)` ukonkretnia N na liczbę elementów listy L.

```
dlugosc([], 0).
dlugosc([G|O], N) :- dlugosc(O, N1), N is N1 + 1.
```

Do wykonania powyższej czynności zliczania elementów listy można użyć innej techniki, mianowicie wykorzystać tzw. licznik (akumulator). Jest to wskazane, jako że posługujemy się wtedy tzw. rekurencją prawostronną, która jest wydajniejsza i powoduje mniejsze zużycie pamięci komputera. Zdefiniujmy pomocniczy predykat `dl`:

```
dl([], Licznik, Licznik).
dl([G|O], N, Licz) :- Licznik is Licz + 1, dl(O,N,Licznik).
```

Z wykorzystaniem predykatu `dl` można zdefiniować:

```
dlugosc2(L, N) :- dl(L,N,0).
```

Usuwanie powtórzeń

Akumulatorem może być także inna struktura, np. lista. Na przykład jeśli chcemy usunąć wszystkie powtarzające się elementy z listy, możemy zdefiniować predykat:

```
usun_powtorzenia([], X, X).
usun_powtorzenia([G|O], X, L) :- member(G, X),
usun_powtorzenia(O,X,L).
usun_powtorzenia([G|O], X, L) :- usun_powtorzenia(O, [G|X], L).
```

Wbudowane operacje na listach

SWI-Prolog ma wbudowane podstawowe operacje na listach. Dokładny ich opis znaleźć można na stronie manuala SWI-Prologa

http://www.swi-prolog.org/pldoc/man?section=builtin.

http://www.swi-prolog.org/pldoc/man?section=lists

```
append(?List1, ?List2, ?List3).
member(?Elem, ?List).
nextto(?X, ?Y, ?List).
delete(+List1, ?Elem, ?List2).
select(?Elem, ?List, ?Rest).
nth0(?Index, ?List, ?Elem).
nth1(?Index, ?List, ?Elem).
last(?List, ?Elem).
reverse(+List1, -List2).
permutation(?List1, ?List2).
flatten(+List1, -List2).
sumlist(+List, -Sum).
numlist(+Low, +High, -List).
```

Set Manipulation

```
is_set(+Set).
list_to_set(+List, -Set).
intersection(+Set1, +Set2, -Set3).
subtract(+Set, +Delete, -Result).
union(+Set1, +Set2, -Set3).
subset(+Subset, +Set).
is_list(+Term).
memberchk(?Elem, +List).
length(?List, ?Int).
sort(+List, -Sorted).
msort(+List, -Sorted).
keysort(+List, -Sorted).
predsort(+Pred, +List, -Sorted).
```

```
merge(+List1, +List2, -List3).
merge_set(+Set1, +Set2, -Set3).
```

Zadanie 2:

Napisz predykat liczący silnię danej liczby.

```
silnia(N, X).
```

YOUR ANSWER HERE

silnia(0, 1).

silnia(N, X) :- N > 0, N1 is N - 1, silnia(N1, X1), X is N * X1.

Zadanie 3:

Napisz predykat znajdujący maksymalna (minimalną) wartość w liście.

```
maximum(L, X).
minimum(L, X).
```

YOUR ANSWER HERE

maximum([], 0).

maximum([G|O], X) :- maximum(O, O1), G > O1, X is G.

maximum([G|O], X) :- maximum(O, O1), G =< O1, X is O1.

minimum([], 0).

minimum([G|O], X) :- maximum(O, O1), G < O1, X is G.

minimum([G|O], X) :- maximum(O, O1), G =< O1, X is O1.

Zadanie 4:

Zdefiniuj predykat liczący sumę wszystkich elementów listy.

```
suma(L, Suma).
```

YOUR ANSWER HERE

suma([], 0).

suma([G|O], Suma) :- suma(O, S), Suma is S + G.

Zadanie 5:

Jak wypisać jednym poleceniem wszystkie permutacje liczb od 1 do 5? Użyj predykatu wbudowanego `permutation`.

YOUR ANSWER HERE

permutation([1,2,3,4,5], X).

Podstawowe operacje wejścia/wyjścia w Prologu

Najprostszą operacją jest czytanie pojedynczego znaku:

```
?- get0(X).
```

Pisanie pojedynczego znaku jest realizowane przez predykat `put`

```
?- put(104), put(101), put(108), put(108), put(111).
```

Można pisać cale termy:

```
?- write('Podaj dane: ').
```

Zwróć uwagę czy różni się to od polecenia (w GNU Prolog (w terminalu uruchamiamy poleceniem `prolog`)):

```
?- write("Podaj dane: " ).
```

Jaka jest sytuacja podczas czytania za pomocą predykatu `read(X)`.

```
?- read(X).
'Jakies dane'

?- read(X).
"Jakies dane"
```

Pisanie do pliku można zrealizować za pomocą predykatu `tell`, który ustawia pisanie do danego pliku. Predykat `told` kończy pisanie do danego pliku. Podobnie, czytanie z pliku zaczyna się predykatem `see`, a kończy predykatem `seen`.

Przykład 1.

Uwaga: Ustaw bieżący katalog za pomocą `working_directory(-Old, +New)`

Sprawdź: http://www.swi-prolog.org/pldoc/man?predicate=working_directory/2

Chcemy zapytać użytkownika o listę liczb i zapisać ją do pliku.

```
pisz_plik :- write('Podaj liste: '), read(L1), tell('przyklad.txt'),
write(L1), write(.), nl, told.
```

`write(.)` jest konieczne do późniejszego odczytu, jako że wszystkie predykaty w Prologu musza kończyć się kropką.

Przykład 2.

Chcemy odczytać wcześniej zapisaną listę z pliku, obliczyć sumę jej elementów, wyświetlić tę sumę oraz dodać do bazy wiedzy Prologu predykat `moja_suma(Suma)`, gdzie zmienna `Suma` jest ukonkretniona na właśnie wyliczoną sumę elementów.

```
czytaj_plik :- write('Czytam z pliku...'), nl, see('przyklad.txt'),
read(L), seen,
write('suma elementow listy z pliku wynosi: '), sumlist(L,Suma),
write(Suma), assertz(moja_suma(Suma)).
```

Zadaj pytanie:

```
?- czytaj_plik.
```

A następnie:

```
?- moja_suma(X).
```

Przykład 3.

Chcemy zapytać użytkownika, z jakiego pliku odczytać dane, nastepnie odczytać z podanego pliku wszystkie listy i wyliczyć sumę elementów każdej listy.

```
czytaj_all :- write('Podaj plik: '), read(Plik), see(Plik), repeat,
not(czyt), seen, nl, write('Skonczone!').
czyt :- read(L), sumlist(L, S), nl, write('suma='), write(S).
```

Uwaga: Nazwę pliku należy podawać jako `'nazwa_pliku'` nie jako `"nazwa_pliku"`.

Zadanie 6:

Zmodyfikuj przykład 3 w taki sposób aby wszystkie imiona zapisane w pliku zostały wczytane i dodane do bazy wiedzy jako predykat `imie(OdczytaneImie)`.

YOUR ANSWER HERE

Dynamiczna zmiana pamięci

Predykaty `asserta` oraz `assertz` dodają do pamięci podane predykaty. Pierwszy z nich dodaje na początek bazy wiedzy, drugi na koniec. Np. chcemy dodać informację, że podane przez użytkownika imię to imie kobiece

```
dodaj_kobieta :- write('Podaj nowe imie kobiety: '), nl, read(Imie),
assertz(imie_kobiece(Imie)).
```

Możemy teraz wykonując `dodaj_kobieta` dodać imię `kasia` a następnie zadać pytanie:

```
?- imie_kobiece(X).
```

I otrzymamy odpowiedź np. X = kasia.

Aby usunąć dany predykat z pamięci używamy predykatów `retract` oraz `retractall`. Np.

```
?- retract(imie_kobiece(kasia)).
?- retractall(imie_kobiece(_)).
```

Predykat

```
abolish(nazwa_predykatu_do_usuniecia, ilosc_argimentow_predykatu)
```

pozwala usunąć wszystkie klauzule danego predykatu, które maja daną ilość argumentów. Np.

```
?- abolish(imie_kobiece, 1).
```