Laboratorium nr 3 - Sztuczna Inteligencja

Listy i operacje na listach

Podstawy teoretyczne

a. Listy

Lista jest podstawową strukturą w prologu i jest ona przetwarzana rekurencyjnie. Lista składa się z głowy (pierwszego elementu) oraz ogona. Np. lista: [3, 4, 2, 1] ma głowę, którą jest element 3 oraz ogon, który jest listą [4, 2, 1].

b. Tworzenie list.

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

?- X = .(3,[4,2,1]). *?- Pets = .(dogs, .(cats,[])).*

X = [3, 4, 2, 1] *Pets = [dogs, cats]*

Należy pamiętać o spacji pomiędzy znakiem równości i kropką.

Listę można również stworzyć **poleceniem**:

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

Dostęp do jej elementów odbywa się poprzez zapisanie listy w postaci:

Lista = [Glowa|Ogon].

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

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

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

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

L zostaje ukonkretnioną listą, której pierwszym elementem jest 3 natomiast drugim jest lista [4,5,6]. L nie ma zatem czterech elementów, a tylko dwa: pierwszy liczbę, drugi – inną listę.

Jeszcze jednym sposobem **utworzenia listy** jest predykat **findall**:

Predykat ten ma trzy argumenty findall(X,Y,Z). Argument X określa term, który spełnia warunek, Y określa warunek i tu wstawiamy nazwę faktu, związanego z warunkiem, a Z to lista tworzona z termów, które spełniają warunek. Czyli: stwórz listę Z elementów X na podstawie tego, że X są elementami z faktu Y i go spełniają.

Przykład 1

Mamy bazę wiedzy wbudowaną w Prologu - (plik likes.pl w katalogu demo):

indian(curry,x).
indian(dahl,x).
indian(tandoori,x).
indian(kurma,x).
mild(dahl).
mild(tandoori).
mild(kurma).
chinese(chow_mein).
chinese(chop_suey).
chinese(sweet_and_sour).

Następnie formułujemy dla Prologa pytania:

?- findall(X,indian(X,x),C).
C = [curry, dahl, tandoori, kurma].

?- findall(X,mild(X),C).

C = [dahl, tandoori, kurma].

Ostatnim elementem listy jest zawsze lista pusta []. Zatem listę z jednym tylko elementem można stworzyć tak: L = [3] lub tak: L = .(3,[]).

Elementami listy nie muszą być tylko liczby, mogą nimi być także inne listy lub całe struktury, np.:

[[5,4], [5,6], [3,4]].

[kasia, lubi, jarka].

[autor(clive barker), autor(bertrand, russell)].

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

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

X=1, Y=2, Z=3, Reszta=[4,5]

Jaką odpowiedź da prolog na następujące pytania? :

a. G = 3, O=[4,5,6], L=[O|G].
b. L = [2], L=[X|Y].
c. L = [2], L=[X, Y|Z].

Jaki efekt będzie miało ujednolicenie następujących list:

a. [] = X.
b. [] = [X|Y].
c. [a,b,c,d] = [H|T].
d. [1,2,3] = [X,Y,Z].
e. [marta] = [G|O].
f. [A, B, C] = [jan, lubi, piwo].
g. [1,2,3] = [G|[2,3]].
h. [[marta, A], B] = [[X, lubi], koty].
i. [[1,2],[3,4],[5,6]] = [X|Y].
j. [7,6] = [6,X].
k. [kobieta(marta), kobieta(jola)] = [X|Y].

c. Predykat ! i fail

Automatyczne nawracanie jest jedną z cech charakterystycznych Prologa. Czasami są sytuacje, kiedy takie nawracanie powoduje stratę czasu, bo przeszukiwanie rozwiązań prowadzi donikąd. Wtedy można zastosować operator !, który pozwala na zablokowanie procesu nawrotu w wybranym miejscu. Jest to predykat bezargumentowy zawsze prawdziwy, który w czasie swojej weryfikacji powoduje zaniechanie badania innych, pozostałych jeszcze do weryfikacji definicji predykatów na bieżącym poziomie drzewa wnioskowania (uniemożliwia dokonanie nawrotu powyżej miejsca, w którym został wstawiony znak odcięcia (!)).

Przykład 2

Zastosowanie operatora odcięcia zmienia logicznie (deklaratywnie) zdanie. Rozważmy program:

p :- a, b.
p :- c.
który jest równoważny zdaniu:
p ⇔ (a &b) ∨ c.
a więc p będzie prawdziwe przy założeniu, że a i b są prawdziwe lub c jest prawdziwe.
Natomiast po użyciu operatora odcięcia program ma postać:
p :- a, !, b.
p :- c.

co jest równoważne zapisowi:
p ⇔ (a &b) ⋁ (~a & c).
Zmiana polega na tym, iż Prolog podczas sprawdzania założeń jeżeli rozpocznie sprawdzanie od pierwszej klauzuli i okaże się, że a jest prawdziwe, wtedy natrafi na ! i już nie będzie sprawdzał po nawróceniu klauzuli drugiej. Natomiast sprawdzenie klauzuli drugiej będzie możliwe tylko w przypadku gdy w pierwszej klauzuli okaże się że a nie jest prawdziwe i prolog nie dojdzie do operatora !.

Przykład 3

Rozważmy program:

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

c(g).
c(h).
c(i).
```

oraz wynik zapytania:

```
?- a(X,Y).
X = d Y = g ;
X = d Y = h ;
X = d Y = i ;
X = e Y = g ;
X = e Y = h ;
X = e Y = i ;
X = f Y = g ;
X = f Y = h ;
X = f Y = i ;
false.
```

Powyżej otrzymaliśmy wszystkie możliwe rozwiązania. Prolog rozważał trzy razy klauzulę pierwszą, ze względu na mechanizm nawracania. Następnie po użyciu predykatu !:

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

otrzymamy już mniej rozwiązań:

```
?- a(X,Y).
X = d Y = g ;
X = d Y = h ;
X = d Y = i ;
false.
```

A więc Prolog podczas przeszukiwania bazy wiedzy, tylko raz rozważał klauzulę pierwszą, gdyż po wejściu do niej natrafił na !, który dezaktywuje mechanizm nawracania[1].

Wyróżniamy dwa rodzaje odcięć:
• **odcięcia „czerwone**” - to takie, które zmieniają interpretację **deklaratywną programu***, utrudniają jego

[1] Przykład zaczerpnięty ze skryptu "Programowanie w logice - Prolog" P. Fulmański

zrozumienie i powodują utratę pewnych rozwiązań.
• **odcięcia „zielone**" - takie, które nie wpływają na interpretację deklaratywną programu, nie zmniejszają jego czytelności i zachowują wszystkie rozwiązania (choć obcinają drzewo poszukiwań)

Przykład 4
Jest dany predykat **funkcja/2**, który nie zawiedzie, gdy pierwszy argument X i drugi argument Y przyjmą wartości opisane warunkami:

- *Jeżeli X <3, to Y = 0.*
- *Jeżeli X ≥ 3 i X < 6, to Y = 2.*
- *Jeżeli X ≥ 6, to Y = 4.*

Rozwiązanie:
funkcja(X , 0) :- X < 3,!.
funkcja(X , 2) :- X >= 3, X < 6, !.
funkcja(X , 4) :- X >= 6.

Zadać pytanie:
funkcja(1,Y),Y > 2.
Aby zwiększyć optymalność kodu można zapisać źródło w postaci następujących reguł:

- Jeżeli X<3, to Y = 0.
- W przeciwnym przypadku jeżeli X < 6, to Y = 2.
- W przeciwnym przypadku Y = 4.

Rozwiązanie:

funkcja(X , 0) :-X < 3,!.
funkcja(X , 2) :- X < 6, !.
funkcja(X , 4).

Przykład 5

indian(curry).
indian(dahl).
indian(tandoori).
indian(kurma).
mild(dahl).
mild(tandoori).
mild(kurma).
chinese(chow_mein).
chinese(chop_suey).
chinese(sweet_and_sour).
italian(pizza).
italian(spaghetti).

szukaj(X):-indian(X),write(X),!.

Następnie po zapytaniu szukaj(X) otrzymamy w wyniku:
curry
X = curry.

Tak więc Prolog przeszukuje bazę wiedzy i zatrzymuje się na ostatniej klauzuli szukaj... , następnie szuka dopasowania dla X w indian(X), przeszukuje bazę wiedzy i natrafia na fakt indian(curry), więc Prolog znalazł pierwsze dopasowanie X=curry, następnie X jest wyświetlany w write(X), po czym wywołuje się predykat ! co oznacza, że Prolog po wyjściu z treści klauzuli szukaj... zakończy przeszukiwanie tuż po zwróceniu znalezionego wyniku X=curry.

Predykat **fail** służy do wymuszania nawrotów. Zamiast wpisywać średnik, aby wyszukać kolejne rozwiązanie,

możemy użyć predykatu fail.

Przykład 6

Dany jest program:
nazwa1(1) :- write('Jeden').
nazwa1(2) :- write('Dwa').
nazwa1(3) :- write('Trzy').
nazwa1(_) :- write(' Nie wiem!').

działanie Prologa:
?- nazwa1(2).
Dwa
Yes
?- nazwa1(2), fail.
Dwa Nie wiem!
No
?- nazwa1(X), fail.
JedenDwaTrzy Nie wiem!
No
W przypadku, gdy połączy się fail z predykatem odcięcia pojawią się takie wyniki:
nazwa2(1) :- !, write('Jeden').
nazwa2(2) :- !, write('Dwa').
nazwa2(3) :- !, write('Trzy').
nazwa2(_) :- write(' Nie wiem!').
Wyniki:
?- nazwa2(2).
Dwa
Yes
?- nazwa2(2), fail.
Dwa
No
?- nazwa2(X), fail.
Jeden
No

Więcej szczegółów można uzyskać podczas wykonywania zapytania w trybie trace.

d. Operacje na listach.

Na listach można wykonać wiele użytecznych operacji, większość z nich bazuje na rekurencji, czyli sprawdzamy pierwszy element listy, po czym w regule odwołujemy się do tej samej reguły wpisując ogon jako następną listę do modyfikacji. Należy jeszcze wpisać przed regułą warunek początkowy lub końcowy. Prolog posiada wbudowane predykaty, jednakże w ramach ćwiczeń skupimy się na własnej implementacji.
Poniżej znajdują się implementacje niektórych predykatów manipulujących listami:

- **Członkostwo w liście**: predykat *member(X,L)* sprawdza czy element X znajduje się w liście L. Przy implementacji korzystamy z następujących faktów: *Element jest członkiem listy, jeżeli jest głową listy. Jeżeli X jest członkiem ogona pewnej listy, to jest członkiem tej listy*:

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

Przykład:

L=[3,4,6,7], member(3,L).

- **Łączenie list**: Aby połączyć dwie listy w jedną należy zaimplementować predykat concat/3. Predykat concat(L1,L2,L3) sprawdza czy lista L3 to połączenie listy L1 i listy L2. Przy implementacji concat/3 korzystamy z następujących faktów: Połączenie listy *pustej z jakakolwiek listą L w wyniku daje listę L. Jeżeli L3 to lista powstała z połączenia listy Tail i L2, to lista L z dodatkowym elementem Head na początku jest listą powstałą z listy Tali z dodatkowym elementem Head na początku oraz listy L2.* Tak więc, od strony nagłówka patrząc, Prolog sprawdza sukcesywnie czy w wyniku zdejmowania elementów (głowy) z list L1 I L3 otrzymamy w rezultacie listę L2:

```
concat([],L,L).
concat([Head|Tail],L2,[Head|L3]):-concat(Tail,L2,L3).
```

Przykłady:

concat([1,2,3], [4,5,6], L3).

wynik: L3=[1,2,3,4,5,6]

concat(L1, [4,5,6], [1,2,3,4,5,6]).

wynik: L1=[1,2,3]

concat(L1, L2, [1,2,3,4,5,6]).

Wynikiem są wszystkie możliwe listy od pustej L1 do pustej L2.

- **Usuwanie elementów z listy**:

Predykat *delete(X,List1,List2) sprawdza czy po usunięciu elementu X z listy List1 otrzymamy listę List2.* Rozważamy następujące fakty: *W przypadku gdy usuwany element jest na początku listy, wynikiem usunięcia jest podlista równa ogonowi listy wejściowej. Natomiast gdy usuwany element jest w środku listy: Jeżeli L to lista powstała z usunięcia elementu X z listy Tail, to tym bardziej lista L z dodatkowym elementem H na przedzie będzie wynikiem usunięcia elementu X z listy Tail z dodatkowym elementem H na przedzie.*

```
delete(X,[X|Tail],Tail).
delete(X,[H|Tail],[Y|L]):-delete(X,Tail,L).
```

Przykład. Dodaj do listy [a,b,c,d] element m. Ile list może powstać w wyniku dodania elementu m?
?- delete(m,X,[a,b,c,d]).

```
X = [m, a, b, c, d] ;
X = [a, m, b, c, d] ;
X = [a, b, m, c, d] ;
X = [a, b, c, m, d] ;
X = [a, b, c, d, m] ;
false.
```

Jak widać powyżej, możemy otrzymać 5 różnych list. Ponadto łatwo zauważyć, że predykat delete/3 może posłużyć także do tworzenia list.

- **Odwracanie listy:** predykat *reverse(L1, L2) sprawdza czy kolejność elementów listy* L1 jest odwrócona w stosunku do kolejności elementów listy L2. Przy implementacji korzystamy z faktów: *Lista pusta jest odwrócona w stosunku do samej siebie. Jeżeli lista L1 ma odwrócone elementy w stosunku do listy L2 oraz lista L3 powstała przez połączenie list L2 i [H], to wtedy lista powstała przez połączenie [H] i L1 ma odwrócone elementy w stosunku do listy L3.*

```
reverse1([],[]).
reverse1([H|L1],L3):-reverse1(L1,L2), concat(L2,[H],L3).
```

Niestety predykat reverse1/2 jest mało efektywny ponieważ za każdym razem, gdy Prolog sięga do drugiej klauzuli musi wykorzystać predykat concat/3. Aby temu zapobiec możemy skorzystać ze struktury danych o nazwie akumulator w celu przechowania listy roboczej. Nowy predykat reverse2/3, który jest trójargumentowy możemy symbolicznie oznaczyć przez reverse2(L1,L2,L3) przy czym jest on prawdziwy, jeżeli lista L3 jest wynikiem połączenia odwróconej listy L1 z listą L2 (tj. połączenie listy L2 z listą L2). Nowa implementacja opiera się na następujących faktach: Połączenie odwróconej listy pustej z listą A jest tą samą listą A. Jeżeli lista

X powstała przez połączenie odwróconej listy T i listy A z dodanym elementem H na początku, to można powiedzieć także, że lista X powstała przez połączenie odwróconej listy L=[H|T] z listą A. Tak więc Prolog przenosi pierwszy element z listy "drugiej" do listy "pierwszej", co w rezultacie nie wpływa na kształt listy "trzeciej".

```
reverse2([],A,A).
reverse2([H|T],A,X):-reverse2(T,[H|A],X).
```

Przykład:

```
132 ?- reverse2([1,2,3],[3,2,1],[3,2,1,3,2,1]).
wynik:
true
132 ?- reverse2([1,2,3],X,[3,2,1,3,2,1]).
X = [3, 2, 1].
```

W celu poprawienia predykatu reverse2/3 na taki, który będzie odwracał kolejność elementów w podlistach, należy dodać odpowiednią klauzulę wychwytującą element złożony:

```
reverse3([],A,A).
reverse3([[H|T]|S],A,X):-!,reverse3([H|T],[],Y), reverse3(S,[Y|A],X).
reverse3([H|T],A,X):-reverse3(T,[H|A],X).
```

Zadania

Zadanie 1.
a) Napisz predykat *len(N,L)*, który jest prawdziwy, gdy N jest długością listy. Podpowiedź: *Nie używaj predykatu length, gdyż jest on predykatem wbudowanym. Wykorzystaj predykat is do porównywania liczb całkowitych.*
b)* Co Prolog zwróci dla zapytania len([a,[a,b],c],X). ? Napisz predykat rlen(X,N), który jest prawdziwy gdy N oznacza liczbę wszystkich termów (licząc także w podlistach elementy) listy X.
Zadanie 2.
Zdefiniuj predykat sum(X,N), który jest prawdziwy, gdy N jest sumą liczb całkowitych z listy X.
Zadanie 3. Zdefiniuj predykat *avg(X, N)* liczący średnią arytmetyczną wszystkich elementów listy X.
Zadanie 4. Zdefiniuj predykat count(X,Y,N), który jest prawdziwy jeśli lista Y zawiera N wystąpień elementu X.
Zadanie 5. Zdefiniuj predykat double(X,Y), który jest prawdziwy jeśli lista Y zawiera każdy z elementów X powtórzony dwa razy. *Przykład: double([a,b],[a,a,b,b]) jest prawdziwy*
Zadanie 6. Zdefiniuj predykat repeat(X,Y,N), który jest prawdziwy jeśli lista Y zawiera każdy z elementów X powtórzony N razy
Przykład
repeat([a,b],[a,a,a,b,b,b], 3) jest prawdziwy
Zadanie 7*. Zdefiniuj predykat sortujący listę liczb całkowitych. Podpowiedź: Zdefiniuj najpierw predykat sort1/1 który sprawdza czy lista jest posortowana. Następnie zdefiniuj predykat *naivesort/2*, który posłuży do posortowania (tj. znalezienia posortowanej) listy.