Programowanie w logice

PROLOG

Struktury danych - listy

- **Lista** – ciąg uporządkowanych elementów o dowolnej długości.
- Elementy mogą być dowolnymi termami: stałymi, zmiennymi, strukturami (w tym listami)
- Lista jest albo **listą pustą**, nie zawierającą żadnych elementów, albo jest strukturą z dwiema składowymi: **głową** i **ogonem**.

Listy

- Lista jest strukturą rekurencyjną (do jej konstrukcji użyto funktora . (kropka)
- Listę pustą zapisuje się: []
- Głowa i ogon listy są argumentami funktora . (kropka)
- Przykłady:

  .(a,[]) lista jednoelementowa

  .(a,.(b,.(c,[]))) lista o elementach a, b, c

Przykłady list

- Wygodniejszy zapis listy: elementy oddziela się przecinkami i umieszcza między [i]

Zamiast: .(5,.(8,.(3,[])))

pisze się: [5,8,3]

[wydzial, matematyki,i,informatyki]

[X,posiada,Y]

[autor(adam,mickiewicz),"Pan Tadeusz"]

[[2,3],[5,6,7],[2,8]]

Lista z głową X i ogonem Y: [X|Y]

lista	głowa	ogon
[]	niezdefiniowane	niezdefiniowane
[jan]	jan	[]
[jan,marta]	jan	[marta]
[a,b,c,d]	a	[b,c,d]
[[1,2],[3,4],5]	[1,2]	[[3,4],5]

- .(a,.(b,.(c,[]))) = [a,b,c].
- .(a,.(B,.(C,[]))) = [a,b,c].
- [a,V,1,[c,s],p(X)] = [A,B,C,D,E].
- [1,2,3,4] = [1|[2,3,4]].
- [1,2,3,4] = [1,2|[3,4]].
- [1,2,3,4] = [1,2,3|[4]].
- [1,2,3,4] = [1,2,3,4,[]].
- [1,2,3,4] = [1|2,3,4].
- [Head|Tail] = [1,2,3,4].
- [Head|Tail] = [[1,1,ala],2,3,4].
- [Head|Tail] = [X+Y,x+y].
- p([_,_,_,[_|X]]) = p([ala, ma, bardzo, [malego, burego, psa]]).

Przetwarzanie list

- Listy są **strukturami rekurencyjnymi**, do ich przetwarzania służą procedury rekurencyjne.
- Procedura – zbiór klauzul zbudowany w oparciu o ten sam predykat.

- **Procedura rekurencyjna** składa się z klauzul:
  - Faktu opisującego sytuację, która powoduje zakończenie rekurencji, np. napotkanie listy pustej,
  - Reguły, która przedstawia sposób przetwarzania listy. W jej ciele znajduje się ten sam predykat, co w nagłówku, tylko z innymi argumentami.

Przykład procedury rekurencyjnej

Drukowanie elementów listy:

```
pisz([]).
pisz([X|Y]):-write(X),nl,pisz(Y).
```

Fakt mówi, że w przypadku napotkania listy pustej (końca listy) nie należy nic robić.

Reguła mówi: podziel listę i ogon, wydrukuj głowę listy, następnie ja pomiń i zastosuj tę samą metodę do powstałego ogona.

- **is_list** (L) - sprawdza, czy L jest listą

- **append** (L1,L2,L3) – łączy listy L1 i L2 w listę L3

```
append([b,c,[s,a],a],[a],X).
append([a],[b],[a,b]).
append(L1,L2,[b,c,[s,a]]).
```

- **member**(E,L) – sprawdza, czy element E należy do listy L

  np. member(5,[3,6,5,7,6])

  lub wypisuje elementy listy L

  np. member(X,[2,3,4,9]).

- **memberchk**(E,L) - równoważny predykatowi member, ale podaje tylko jedno rozwiązanie

- **nextto**(X,Y,L) – predykat spełniony, gdy Y występuje bezpośrednio po X

  nextto(X,Y,[2,3,4,5]).

  nextto(3,Y,[2,3,4,5]).

  nextto(X,4,[2,3,4,5]).

- **delete**(L1,E,L2) – z listy L1 usuwa wszystkie wystąpienia elementu E, wynik uzgadnia z listą L2
- **select**((E,L,R) – z listy L wybiera element, który daje się uzgodnić z E. Lista R jest uzgadniana z listą, która powstaje z L po usunięciu wybranego elementu.
- **nth0**(I,L,E) – predykat spełniony, jeśli element listy L o numerze I daje się uzgodnić z elementem E
- **nth1**(I,L,E) – predykat podobny do nth0.

- **last**(L,E) – ostatni element listy L
- **reverse**(L1,L2) – odwraca porządek elementów listy L1 i unifikuje rezultat z listą L2
- **permutation**(L1,L2) – lista L1 jest permutacją listy L2
- **flatten**(L1,L2) – przekształca listę L1 w listę l2, w której każda lista składowa zostaje zastąpiona przez swoje elementy, np.flatten([a,[b,[c,d],e,f]],X).
- **sumlist**(L,S) – suma listy liczbowej L
- **numlist**(M,N,L) – jeśli M,N są liczbami całkowitymi takimi, że M<N, to L zostanie zunifikowana z listą [M,M+1,…,N]
- **length**(L,I) – liczba elementów listy L

Operacje na listach

Sprawdzenie **czy element jest na liście** (2 argumenty):

Procedura: X jest elementem listy L, jeżeli X jest głową listy L lub X jest elementem ogona listy L.

```
element(X,[X|_]).
element(X,[_|Ogon]) :- element(X,Ogon).
```

_ to zmienna anonimowa zastępująca głowę listy [_,Ogon], jej nazwa nie ma znaczenia

Predykat wbudowany: **member**

Operacje na listach

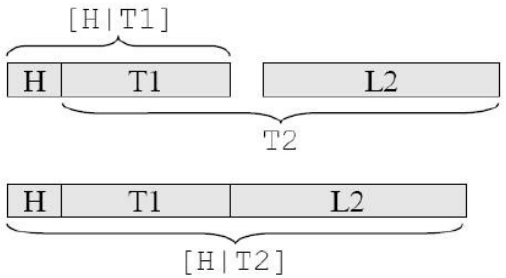
Łączenie list (3 argumenty):

Procedura: Jeżeli pierwszy element listy jest pusty [], to drugi i trzeci element muszą być takie same (L = L).
Jeżeli pierwszy element nie jest pusty, to

- głową listy L3 staje się głowa listy L1,
- ogonem listy L3 jest ogon listy L1 złączony z listą L2.

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

Predykat wbudowany: **append**



Operacje na listach

Usuwanie z listy (3 argumenty):

usun(E,L,W)
E – element, który należy usunąć z listy L,
L – lista wyjściowa,
W – lista wynikowa
Procedura: Jeżeli X jest w głowie listy, to wynikiem będzie ogon listy. Jeżeli X jest w ogonie to rekurencyjnie przeszukaj ogon.

```
usun(X,[X|L1],L1).
usun(X,[Y|L2],[Y|L3]) :- usun(X,L2,L3).
```

Predykaty wbudowane: **delete** i **select**

Operacje na listach

Liczba elementów listy (2 argumenty):

dlugosc(L,N)
L – lista elementów,
N – liczba elementów.
Procedura:

- długość listy pustej jest równa 0 (fakt)
- Długość listy, to długość jej ogona plus jeden (reguła)

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

Predykat wbudowany: **length**

Operacje na listach

Odwracanie kolejności elementów listy

```
odwracanie([],[]).
odwracanie([A|B],C):-
  odwracanie(B,D),append(D,[A],C).
```

Predykat wbudowany: **reverse**

Zadanie
Zdefiniować predykaty: parzysta(Lista) i nieparzysta(Lista), które zwrócą wartość true, jeżeli argument jest odpowiednio listą z parzystą lub nieparzystą liczbą elementów.

Wykorzystano zależność, że jeżeli lista jest w jednym kroku jest parzysta, to w następnym kroku po odjęciu jednego elementu będzie sprawdzana, czy jest nieparzysta, aż do osiągnięcia listy pustej.

```
parzysta([]).
parzysta( [_| T]) :- nieparzysta( T).
nieparzysta( [_| T ]) :- parzysta( T).
```

```
parzysta(L):-
        length(L,N),mod(N,2)=:=0.
nieparzysta(L):-
        length(L,N),mod(N,2)=:=1.
```

```
parzysta([]):-true.
parzysta(L):-length(L, X), 0 is X mod 2.

nieparzysta([]):-false.
nieparzysta(L):-length(L, X), 1 is X mod 2.
```

Literatura

- W.F.Cloksin, C.S.Mellish, Prolog.Programowanie, Helion.
- E.Gatnar, K.Stąpor, Prolog, Wyd. PLJ.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP.
- F.Kluźniak, S.Szpakowicz, Prolog, WNT.
- R.Kowalski, Logika w rozwiązywaniu zadań, WNT.