```
  subor ..\prg2\tex\cvicenia.tex , ...cvic00.txt
30.10.2001, 5.cv.

1. cv. ...
2001: rodokmen

add: vypisovanie a nacitanie obsahu dynamickej pamati
  - do termu, do XML; obecne do dobre zatvorkovanej struktury

2.cv. 9.10.2001
datumy - navrh repr.
ine repr.

3cv 2001
zoznamove operacie ...
multiappend alias concat (spojenie zoznamu zoznamov)
prefix (alebo suffix), vsetky prefixy (alebo suffixy)
kombinacie (variacie),
  (s unarnou reprezentaciou cisel pre pocitanie velkosti)
  alebo s pripravenou dlzkou zoznamu
(n-arne stromy) NIE

DC: podmnoziny backtrakingom, vsetky podmnoziny do zoznamu

4.cv 2001
- DC podmnoziny
  idea: generuj a testuj; kedy to nie je vhodne pouzit
- vyroba vyvazeneho stromu z usporiadaneho zoznamu
  - delenie zoznamu na polovice, rozne pristupy
  - pre neusporiadany: pouzitie v Heapsorte (asi len idea)
    - budovanie stromu
    - prebublavanie prvkov kvoli usporiadaniu
      - tvorba haldy zo stromu
    - vypustanie prvkov pri zmensovani hlady (len idea)
- isset: interpret mnozinovych vyrazov (bude na prednaske)
*  s operatormi + zjednotenie, * prienik, - rozdiel
*  bez definicie isset ako operatora - to este nemali

DC: transpozicia matice

5.cv 2001
lookup/3 pre zoznamy (member), stromy (s @< a ->),
 - struktura a pristupove (a budovacie) procedury si musia odpovedat
   - prolog to nepodporuje (vs. zdruzovanie v Objektoch a Moduloch)
 - s 2D klucmi : a) ako kart. sucin elementarnych klucov
  - mozno pouzit minule programy
b) 2 urovnova struktura, kazda uroven samostatne vyhladavanie
(s odpovedajucou procedurou)
   - obecne n-dimenzionalne: analogicky
     - ak to napisem rekurzivne (okrem poslednej vrstvy- kluca), potom musia
vsetky urovne byt rovnake
```

- pre obecne vyhladavanie s roznym poctom dimenzii (odpoveda Aho-Corasick)
  - hodnota musi byt dvojzlozkova: 1 cast je vlastna hodnota (pre dany kluc)
2. cast je zostavajuca vyhladavacia struktura, pre dlhsie kluce
s rovnakym prefixom

boolovske vyrazy:
definicia operatorov
rek. struktura vyrazu
vyhodnocovanie, bez premennych, s premennymi (predavane v prostredi)
kazda spojka ma popis semantiky
(pr. and - nespravne, spravne s opakovanim, bez op.)
- nemame moznost parametrizacie procedur (hi-or. kod)

DC:
1. vybudovanie vyhl. stromu AC
1a. rozdelenie zoznamu na triedy ekvivalencie (podla prveho pismena)
2. uprava vyhodnocovanie na preklad do zasobnikoveho stroja

6.cv.
"xref" : vypis funktorov termu s cetnostami (bez pocitania)
...

7cv. (priprava)
- grep(Str, Reg, Poloha) :- retazec Str odpoveda regularnemu vyrazu Reg
od zaciatku do dlzky Poloha.
- fold pre ine struktury
- kompilacia ofarbenia grafu do prologu
(mozno nabuduce: setof ...: dynamicke progr., prehl. gr. do sirky)

1-2. cv. rodokmen:
surodenci, rodicia, nejstarsi bratr ...
okrem predikatu rodic/2 dalsie predikaty: manzele(M,Z), rodina(M,Z,Deti), ...

dve reprezentacie pre pohlavie
1) muz(adam). zena(bozena).
2) pohlavi(muz,adam). pohlavi(zena, bozena).
Repr. je abstraktnejsia a obecnejsia. Jej prvy argument je novy "vyctovy typ".
V 2) je pohlavie ako data, napr. mozem porovnavat, v 1) je schovane v mene
predikatu.
- Analogicky dve reprezentacie pre bohatsie triedy: botanicky kluc, zivocisna
hierarchia
- Prevod medzi reprezentaciami (pouzije sa len jeden - pre spravny smer)
pohlavi(muz,X) :- muz(X). ...
muz(X) :- pohlavi(muz,X). ...
- retazec muz je v prvom a stvrtom pripade atom (konstanta, funkcny symbol
s nulovou aritou), v 2. a 3. pripade predikatovy symbol. Sucasny vyskyt nevadi.

Q: negativne informacie: ako definovat sirotu?
zatial len: pridat do DB. (neskor negacia a CWA - predpoklad uzavreteho sveta)

```
DC: opravdu nevlastni sourozenec
onsour(X,Y) :- rodic(RX,X), rodic(RXY,X) , RX\=RXY,
rodic(RXY,Y), rodic(RY,Y), RXY\=RY, RX\=RY, X\=Y.
DC: bratranec ...
I. najstarsi bratr.
II. reprezentacia fronty

--------------------------------------------------
2.cv.
datumy: moznosti reprezentacie
1) [2001,10,9]
2) datum(2001,10,9)
2') datum(2001,rijen,9)
3) datum(9,10,2001)
4) datum(den(9),mes(10),rok(2001))
5) [rok(2001),mes(10),den(9)]
5') datum([rok=2001,mesiac=10,den=9])
6) datum(rok=2001,mesiac=10,den=9)
7) "2001-10-9"  retazec
8) '2001-10-09' atom
a dalsie varianty

vhodne riesenie: 2) alebo 3), pre 2) bude naviac zdarma usporiadanie datumov
nevyhody:
7) a 8) - nie je to struktura, netrivialny pristup na zlozky
1) zoznam sa bude pliest s inymi zoznamami, naviac mierne pamatovo neefektivny
- a osetrovanie vacsej pamate stoji vacsi cas !!
4) pomenovanie v d.s. sa nevyuzije, to nie je zaznam s pomenovanymi zlozkami
vyberanie casti/zloziek je dane polohou argumentu
5), 5') pamatovo neefektivne, nutne postupny pristup na zlozky pomocou
member - ak nie je fixovane poradie; ak je fixovane, tak mozem
pouzit kompaktnejsiu repr. 2.)
    - nevhodne, ak mame pevny vycet poloziek;
ak je vycet poloziek premenny (ale to je iny pripad), potom 5) je riesenie
(viz "zaznamy" v Haskelli)
6) pamatovo neefektivne, na polozky sa pristupuje polohou, nie menom

2') operaciu "za mesiac" musim zadat vyctom; v 2) staci pricitat 1

- v jednom programe je mozno pouzivat viac reprezentacii s roznymi
funkcnymi symbolmi
  - prevody a udrziavanie konzistencie je na programatorovi (jazyk nepodporuje)
prevod 2' na 2:
mk_datum2(datum_r2a(R,M1,D),datum(R,M,D)):- prevodm(M1,M).
prevodm(1,leden).
...
prevodm(12,prosinec).

----------------------------------------
--navrhy d.s.
nerekurzivnych: hradla (potrebuju ZOZNAM vstupov), ...
rekurzivnych:
```

```
- pr: elektr. schema (par. a sekv.), layout 2d (novinovej) stranky, graf,
binarny strom, 2-3 strom, polynomy, synt. strom programu a vyrazu, Html & XML
  -  layout2d: ne zcela obecny, len delenim stranky
stranka -> pod(stranka,stranka); vedle(stranka, stranka);
sloupek(text); fotka(binary)
- navic: parametry: pod(params,s,s) ...

- unarne cisla: plus, krat, ...

zaver:
- pomocou termov je mozne reprezentovat d.s.
- (nizka) pamatova neefektivnost sposobuje casovu neefektivnost

------------------------------------------
3. CV. 20.10.2000.
opakovanie append/3

-- mappend/2 (Xs,Ys)
multiappend: spoji zoznam zoznamov Xs do jednoho zoznamu Ys.
  Len jedna uroven vnorenia.
Zobecnenie append na viac nez dva zoznamy.
mappend([],[]).
mappend(([X|Xs],Ys):-
  append(X,Ys1,Ys), %(1)
  mappend(Xs,Ys1).  %(2)
-Q(uestion): preco append moze byt pred rek. volanim mappend
A: Premenna Ys1 sa vyuziva ako logicka premenna, pri
volani append ostane volna a neskor (v rek. volani) sa substituuje.
-- prehodenie (1) a (2) by bolo mozne. Z hladiska toku dat by bolo cistejsie.
?- mappend([[1,2],[],[3,4],[a]],X). % volanie
  X=[1,2,3,4,a]

-- na mappend vysvetlena strukturalne rekurzia (podla prveho argumentu)

-- prefix/2
- dve varianty:
a) rekurzivne napisana
b) pomocou append
prefix(X,P):- % X je prefix P
  append(X,_,P).
- prefix ako specialne pouzitie append
- rekurzivne napisany prefix je "projekcia" append na prislusne argumenty (1.
    a 3.)
- podla sub b) sa lahsie pisu podobne predikaty, napr. suffix
  - pretoze pouziva predikaty z "kniznice"
-Q: mody: (nemali na prednaske)
A: mozne mody 1. (+,+), 2. (+,-) 3. (-,+) 4. (-,-)
1. OK, pouzitie ako test: splni sa alebo nesplni sa
2. OK, Vysledok je jeden.
- analogicky mod u suffixu sa zacykli, pretoze vysledkov je vela
3. OK, pouzitie ako generator: generuje (pomocou backtrackingu) prefixy,
od najkratsieho
```

```
4. CYKLI, nekonecne mnozstvo rieseni
?- prefix(X,P)
X=[],    P= _1
X=[_2],    P= [_2|_1]
X=[_2,_3],    P= [_2,_3|_1]
...
pozn.: volne premenne vystupuju ako podtrzitko a cislo:
syntakticky je to (volna) premenna, cislo je nejaka interna
reprezentacia, tj. zhodne premenne maju zhodne cislo.

-- DC (supl. 18.10): preskumat chovanie member(a,X)

-- suffix/2
- viz minule pozn.
- mody
- specialne mod (?,-)
  riesenie v tomto pripade nejde vyjadrit v kompatnej podobe!
  pretoze rekurzia je riadena druhym argumentom, ak nie je vstupny,
  program cykli v backtrackingu (pricom vydava nekonecne mnoho roznych! rieseni)
-Q: poradie vysledkov: od najdlhsieho
- od najkratsieho by to slo pouzitim akumulatora

-- suffixy/2 (Xs,Sss)
Pre vstupny zoznam Xs sa vyrobi zoznam vsetkych jeho sufixov (pripon)
v zozname Sss. Typovo: Sss je zoznam zoznamov.
- predikat suffix mohol tiez generovat vsetky pripony, ale (len) pomocou
backtrackingu. AK taketo riesenie nevyhovuje, vydanie vysledku (v zozname,
naraz a deterministicky) je obecnejsie.
- pouzitie: vygenerujem explicitne sufixy do zoznamu a potom ich
dalej spracovavam, napr. vyberiem z nich tiem, ktore splnaju nejaku
dalsiu podmienku.
(ped.: suffixy su jednoduchsie nez prefixy)
suffixy([],[[]]). % (1)
suffixy([X|Xs],[[X|Xs]|Ss]):- % (2)
  suffixy(Xs,Ss).
- v (1) je vysledkom , (jednoprvkovy) zoznam, ktory obsahuje prazdny zoznam
- v (2) sa vstupny zoznam kopiruje (mierne neefektivne)
- generovane v poradi od najdlhsieho

-- prefixy/2 (Xs,Pss)
- idea rekurzivnej klauzule: strukturalne rekurzia podla vstupnehpo zoznamu Xs,
tvorim zoznam prefixov Pss1 v rekurzii, ku kazdemu prefixu musim pridat
prvy prvok, a naviac pridat pradzny prefix.
prefixy([],[[]]).
prefixy([X|Xs],[[]|Pss]):-
  prefixy(Xs,Pss1),
  map_pridejX(X,Pss1,Pss). %(1)
-Q: poradie generovanych prefixov: od najkratsieho
-Q: jednoprvkovy zoznam (resp. vsetky dlhsie) vo vysledku vznikne
pridanim prvku X k prazdnemu zoznamu vratenemu z rekurzie (resp. k ostatnym)
- map_pridejX(X,L1,L2): konkretne v tomto pripade chceme ku kazdemu
prvku zoznamu L1 pridat (na zaciatok) prvok X. Obecna myslienka je spracovanie
```

```
zoznamu (L1) tak, ze na kazdy jeho prvok sa vykona ta ista operacia a vytvori
sa novy prvok vystupneho zoznamu L2. Zatial pre obecne parametricke zadanie
operacie nemame prostriedky, ale myslienka (progr. idiom) je obecny. Predikat
tohoto druhu sa nazyva map.
V map_pridejX (aj map obecne) ide o transformaciu 1:1, tj. vstup aj
vystup su rovnako dlhe.
map_pridejX(_,[],[]).
map_pridejX(X,[L|Ls],[V|Vs]):-
  V=[X|L],  % (1)
  map_pridejX(X,Ls,Vs).
- v (1) je operacia explicitne oddelena od zvysku programu, tj. da sa zmenit,
 ak chceme inu funkcionalitu
  - alebo zaintegrovat do hlavy klauzule, ak chceme optimalizovat

-- rozobratie reverse
- dva (klasicke) sposoby
  poroznanie asymptotickej zlozitosti
- idea akumulatora
- mame pristup na uz spracovane casti (zoznamu)
  - na rozdiel od skladania substitucii
    (resp. spracovanie vysledkov z rekurzie)
-Q!: nutnost vystupneho argumentu
A: Vysledky zanorene v rekurzii musime dostat na povrch.
- nutnost interface: schovanie pritomnosti akumulatora a jeho inicializacie

--generuj a testuj - pre backtracking
--generuj a filtruj - pre zoznam moznosti
- vhodne, len ak pocet vsetkych moznosti nie je omnoho
vacsi nez pocet uspesnych moznosti.
  - ak je generovanie postupne, aj testovanie ma byt postupne
(odmietnutie (fail) co najskor), pr. damy
  pr: nevhodne: kombinacie z podmnozin - lepsie specialny generator

DC 18.10.:
b) subset/2
k danemu zoznamu reprezentujucemu mnozinu generovat (/testovat) vsetky
podmnoziny, poradie prvkov je v zhodnom poradi so vstupom
c) subsets/2
 dtto, vratit vsetky podmnoziny v zozname

-- do buduca:
pr: sublist, subpart, halve ...
filter - idea
map obecne
---------------------------------------------------------------
-- podmnozina
moznosti:
a) generovanie
b) zoznam podmnozin
  - v oboch pripadoch je riesenie O(exp(n)), a) cas, b) cas aj priestor
  => nie je vhodne pre metodu generuj a testuj, lepsie pouzit specialne
generatory
```

```
% subset(+Set,?Subset)
subset([],[]).
subset([I|Is],Os):-
  subset(Is,Os).
subset([I|Is],[I|Os]):-
  subset(Is,Os).

Q: poradie generovania:
 - pridavame od poslednych prvkov
 - poradie v ramci mnozin: zdedene zo vstupu.

varianta:
- vybranie zaverecnej unifikacie "za zatvorku"
subset([],[]).
subset([I|Is],Os):-
  subset(Is,O1),
  (Os=O1 ; Os=[I|O1]).
- v tomto pripade sa "vybranim za zatvorku" neusetri cas ani pamat.
  - ak by bol predikat deterministicky, tj. z oboch vetiev sa
uplatnila len jedna, potom takato zmena moze program zefektivnit
(napr. zjenotenie/union a prienik/...)

b) vsetky podmnoziny v zozname
- subsets(Is,Os).
subsets([],[[]]).
subsets([I|Is],Os):-
  subsets(Is,O1),
  map_pridejX(I,O1,O2),
    % map(pridej(I),O1,O2), % obecne map
  append(O1,O2,Os).
% pridej(I,L,[I|L]).
- map: obecne pouzitelny vzor rekurzie, dostava
proceduralny(!) parameter

------------------------------------------------------------
-- reprezentacia fronty:
- len idea, bez kodu
chceme pridavat dopredu zoznamu, uberat z konca
-Q: ako navrhnut d.s. rozumne
A: kym pridavame, OK, ak chcem ubrat, tak obratime (naraz) vsetky dosial
pridane prvky a uberieme zase hlavu.
d.s. q(Pridane, Obratene)
prazdna fronta: init(q([],[])).
add_q(X,q(P,O),q([X|P],O).
get_q(X,q(P,[X|O]),q(P,O)). % je z coho uberat
get_q(X,q([],[]),_):- error('get_q: vybirani z prazdne fronty').
get_q(X,q(P,[]),V):-  % (kl3)
P\=[]  %(1)
reverse(P,O),
get_q(X,q([],O),V). % taky moznost
- (1) obrana pred zacyklenim - nekonecnym obracanim prazdneho zoznamu
```

```
- misto kl3:
get_q(X,q(P,[]),q([],O)):-  % (kl3)
P\=[]  %(1)
reverse(P,[X|O]). %(2)
-(2) vyuzivam vlastnost logickych promennych: castecne urcena vystupna
premenna nevadi

----------------------------------------------------------------
26.10. 4.cv.
-- vyroba vyvazeneho stromu a delenie zoznamu na polovicu
- rozdelenie zoznamu na polovicu:
  - vcetne osetrenia lichej dlzky
moznosti:
a) spocitat dlzku, potom vydelit a rozdelit
b) delit postupne pri prechode zoznamu
  b1) suda - licha
  b2) prva cast, druha cast
(c) spocitat dlzku, pomocou append rozdelit a nasledne vybrat spravnu dlzku.

ad a,c) pouziva sa predikat length/2 z kniznice, resp. prednasok
- pouzivanie predikatov z kniznic, resp. obecne dopredu pripravenych
predikatov je rozumne, vhodne a podporovane.
ad c) metoda generuj a testuj:
generuj pomocou append (v mode (-,-,+) ), testuj - vyber spravnej dlzky.
  - v tomto pripade je tato metoda nevhodna (neefektivna), pretoze sa
generuje prilis vela moznosti, a v podstate sa nepouzije informacia, ze
predikat "rozdelenie na polovicu" je deterministicky. (sposobuje
neefektivitu pri backtrackingu a spotrebu pamati - info o backtrackingu
sa musia niekde ulozit)

ad b1) priamociare
ad b2) 1) spocitanie dlzky - viz a)
- pouzije sa predikat rozdel - specialne pre tuto ulohu
- pouzije sa take/3 a drop/3 (Haskell) -
take(+N, +-Is, -Os) vezme z Is prave N prvych prvkov do Os, ak existuju
vsetky, ak ich je menej
drop(+N, +-Is, -Os) zahodi z Is prave N prvych prvkov, zvysok do Os
2) kopia zoznamu sa pouzije ako citac a bude sa prechadzat
dvojnasobnou rychlostou

- pozn.: neskor to budeme potrebovat  ako rozdelenie na prvu a druhu cast,
preto to programujeme takto.
Ina uloha je rozdelenie na sude a liche.
b2-2) %halve1(+I,+Counter,-O1,-O2)
halve1(O2,[],[],O2).
halve1(O2,[_],[],O2).
halve1([I|Is],[_,_|C],[I|O1],O2):-
  halve1(Is,C,O1,O2).
- chyba interface, ktory schova arg. Counter

a)
halve2(I,O1,O2):-
```

```
  length(I,N),
  N1 is N div 2,
  rozdel(I,N1,O1,O2).
rozdel([],N,[],[]).
rozdel(Is,0,[],Is).
rozdel([I|Is],N,[I|O1],O2):-
  N>0, % !nutny test
  N1 is N-1,
  rozdel(Is,N1,O1,O2).
  - rozdel je obecne, nemusi platit N > |I|.
  - invariant rozdel: O1++O2 = I and (O2 = [] or N = |O1|)
  - v Haskelli su "take" a "drop", ktore vracaju (len) prvy, resp.
    druhy zoznam.
Ak sa to robi naraz, je to efektivnejsie.

-- vyroba vyvazeneho stromu: bolo na prednaske
mk_tree([],nil).
mk_tree(I,T):-
  halve(I,I1,[X|I2]), % potrebujem |I2|>=|I1|
  mk_tree(I1,T1),
  mk_tree(I2,T2),
  T=t(T1,X,T2).
- X moze byt strukturovane, napr. Key-Value
- osetrenie jednoprvkoveho vstupu je zbytocne
  (znizuje efektivitu a priehladnost kodu, vracia viac rieseni)

----------------------------------------------------
-- lookup(K,V,S)
- vyhladavanie podla kluca: ku klucu Key chceme v strukture Struct
  najst hodnotu Value
- mozna reprezentacia dat.str.: zoznam, strom, (kombinacie) ...
- lookup je interface pre vyhladavanie k d.s.
- lookup moze byt v principe nedeterministicky,
  - ak nedeterminizmus d.s. nepodporuje (efektivne), mozeme vracat
    zoznam vsetkych hodnot
a) struktura je zoznam, ktory obsahuje dvojice Key-Value
lookup(K,V,S):-member(K-V,S).
b) struktura je binarny vyhladavaci strom
  - musime mat aj usporiadanie na klucoch, nielen rovnost!
struktura stromu:
b1) bud stvorica, kde je kluc a hodnota samostatne
  vyhladava sa podla kluca
type tree(TK,TV):- nil; t(tree(TK,TV),TK,TV,tree(TK,TV))
b2) trojica, pricom kluc a hodnota su spojene, podobne ako
vyssie v zozname, vyhladava sa len podla prvej zlozky

% lookups, stromove, pre t/3
%lookups(K, void, _V) :- fail. % nenaslo
lookups(K, t(L,K1-V1,R), V) :-
K=K1    -> V=V1 % priradenie vystupu
      ; K @< K1 -> lookups(K,L,V) % porovnanie termov(1), rekurzia
      ; /*K@>K1*/  lookups(K,R,V) % druha rekurzia
```

```
(1) pretoze porovnavam termy (nielen cisla), je kod "polymorfny"
vzhladom k typu kluca
(-> funguje aj pre strukturovane kluce, ale s vstavanym usporiadanim)

- lookup je (jedna z procedur pre) univerzalny interface pre vyhladavanie
  (pre pracu so strukturou potrebujeme este
vytvorenie/pridavanie/update struktury)
  (<- ak ma byt program nezavisly na konkretnej strukture, musi byt
nezavisly aj na tvorbe struktury, resp. ta musi byt schovana)
  - Prolog nepodporuje zdruzovanie struktur a kodu (v objektoch alebo moduloch)

-- lookup2d(K,V,S)
(lookup v dvoch dimenziach, K = K_1 x K_2)
- dtto, ale kluc je strukturovany, dvojzlozkovy
  - minuly kod funguje ( je "polymorfny" vzhladom k strukture kluca)
- metody:
a) ako lookup, nevyuziva 2d strukturu
  -> neefektivita, ak su kluce huste
kazdy kluc sa porovnava cely
b) vyhladavanie podla prvej zlozky, ako hodnota sa vrati
vyhl. struktura podla druhej zlozky, v nej sa hlada podla
druhej zlozky
  - kazda uroven moze mat inu vyhl. strukturu
-> a pouzivat ine procedury
napr. vonkajsia uroven strom , vnutorna zoznam:
lookup2d(K1-K2,V,S) :-
lookups(K1,S,S1), % vonk. strom
lookup(K2,S1,V). % vnut. zoznam

DC 2) prevod 2d stromu na zoznam s 2D klucom

-------------------------------------------------------
5.cv. 3.11.
- vyhladavanie s multidimenzionalnym klucom
  - ako minule: kluc je zoznam klucikov
A) jednoduchsia varianta: vsetky kluce su rovnako dlhe
   repr. : zoznam
priklad dvojice: [1,a]-a1 :: [klic]-hodnota
lookupD([],S,S).
lookupD([K|Ks],H,S):-
  lookup(K,HS,S),
  lookupD(Ks,H,HS).
- ak je kluc neprazdny, vyhlada sa na 1 urovni pomocou
klasickeho lookup (v prislusnej d.s.). Tato prva uroven vrati
vyhl. strukt. podla zbytku kluca, na ktoru sa vyhl. vola rekurzivne.
  - tentokrat na vsetkych urovniach musi byt roznaka d.s., pretoze
vsetky kluciky vyhladavame rovnakou procedurou

B) kluce roznej dlzky
- mozna kolizia: kluc je prefix ineho -> d.s. musi vracat Hodnotu
aj zbylu vyhl. strukturu
- povodna hodnota v tvare H:S
```

```
lookupD([],H,H:_S).
lookupD([K|Ks],H,_H:S):-
  lookup(K,HS,S),
  lookupD(Ks,H,HS).
- v oboch pripadoch, aby rekurzi mohla (uniformne) ist cez kluciky,
musia byt vyhl. struktury na jednotlivych urovniach stejne.
  - v pripade klucov ako dvojic (vyz vyssie) moze byt struktura
na kazdej urovni ina, pretoze nepotrebujeme rekurziu cez urovne.
-Q: ako odlisit: s d.s. niest (typovu) info o vyhl.s. na prislusnych
urovniach

-------------
- tvorba vyhl. strukt. (pre kluce zhodnej dlzky)
dve moznosti: 1) inkrementalne a 2) davkovo
ad 1) inkrementalne:
na zaciatku razdna d.s., postupne pridavam dvojice po jednej
- nevyhoda: pomale, mnoho medzivysledkov -> spotreba pamate (docasna)
2) davkovo:
predpokladame, ze mame dvojice v zozname.
idea: podulohy: a) rozdelit dvojice na "triedy ekvivalencie" podla
prveho klucika do zoznamu, b) potom vybudovat rekurzivne vyhl. struktury
pre jednotlive triedy ekv., do zoznamu, c) vybudovat vyhl. strukt.
pre prvy klucik
  - pozn.
a) aby dobre fungovala rekurzia, mali by sa kluce skracovat
(v triedach ekv. vzhladom k celku)
b) pomocou map(stavba_strukt) na ziskany zoznam
c) ak je jednourovnova v.d.s. zoznam, tato cast je prazdna
a) implementacia rozkladu na triedy ekv.:
  - obecna vyrovba tried ekv.
triedy ekv. su charakterizovane proc. parametrom :: a->a->Bool,
ktory urcuje, ci dva kluce patria do stejnej triedy ekv.
  - dimenzia usporiadania
  1) usporiadane kluce: - staci usporiadanie podla prveho klucika,
potom delit na suvisle sekcie
  2) neusporiadane kluce: postupne vyberat triedy rozkladu ekv.
s prvym este nespracovanym prvkom.
- proceduralnym parametrom Feq!
%eqrozklad( +Feq, +Zoznam, -Triedy)
eqrozklad( _F, [], []).
eqrozklad(  F, [I|Is], [[I|T]|Ts]):-
  mk_test(F,I,TestEq), % ! progr. vyssieho radu (lambda)
  rozdel(TestEq, Is, T, Zb),
  eqrozklad(F, Zb, Ts).
- predikat rozdel je zobecneny filter, ktory rozdeli prvky Is
na vyhovujuce a nevyhovujuce podmienke. (filter vracia len
vyhovujuce)
  - parametry rozdel/4: v prvom moze byt kontext predavany
podmienke alebo obecne podmienka (vcetne kontextu).
% rozdel(P, In, Out1,Out2)
rozdel(P, [], [], []).
rozdel(P, [I|Is], Same, Other):-
```

```
  (test(P,I) -> Same = [I|Same1], Other=Other1
             ;  Same = Same1, Other=[I|Other1] ),
  rozdel(P, Is, Same1, Other1).
- a jine varianty ...
  (DC) napr. sucasne rozdelovanie do "tried ekvivalencie"

- tvorba vyhl.d.s. - kluce roznej dlzky
  - ide o dopredne vyhladavanie v algoritme Aho-Corasick
- v com sa zmeni idea:
  - zvolime nejaku hodnotu pre neexistujuce kluce (!monady)
    v nasom pripade 'nic'
a0) vyhladame a odstranime zo zoznamu dvojicu s prazdnym klucom
  jej hodnota (ak existuje, inac nic) bude prva cast vysledku (H v H:S).
- rozdelenie, rekurzia a vybudovanie prvej urovne vyhladavania su
stejne
d) vysledok je dvojica Hodnota:Struktura

----------------------------------------------
bool. vyrazy: (niekde by to uz malo byt spisane?!)
struktura, vyhodnoicovanie, kompilacia
(analogia k mnozinovym vyrazom, ktore su na prednaske)

def. d.s. - bv ako bool. vyraz
obsahuje konstanty true, false,
spojky and/2, or, xor, imp, ekv, non/1
premenne - neskor
argumenty spojek su zase bv.
type bv(T) :- true;false;
non(bv(T)); and(bv(T),bv(T)); ...
prom(T)

- def. operatorov
umysel: non viaze najviac (najnizsia precedencia),
and nad nim dolava asoc. , or, imp doprava, ekv neasoc
:- op(200,  fy, non).
:- op(210, yfx, and).
:- op(215, xfx, [nand,xor]). % napriklad
:- op(220, yfx, or).
:- op(230, xfy, imp). % doprava asoc.
:- op(240, xfx, ekv).

- pr.: prom(a) imp prom(b) imp prom(a) and prom(b)
se pochopi obvyklym sposobom:
imp(prom(a),imp(prom(b),and(prom(a),prom(b))))
- pr.: vyraz  true ekv true ekv true
je syntakticky chybne, ekv je neasociativne

-- vyhodnocovanie bv, bez premennych
- pomocou strukt. rekurzie:
% eval(+vyraz, ?hodnota)
eval(true,true).
eval(false,false).
```

```
eval(non(B),V):-
eval(B,V1), % rekurzia
eval_non(V1,V). % vyh. spojky
eval(and(B1,B2),V):-
eval(B1,V1), eval(B2,V2),
eval_and(V1,V2,V).
... analogicky pre ostatne spojky.

- pomocne predikaty:
% eval_non - vyctom: spracuva len konstanty
eval_non(true, false).
eval_non(false, true).

- eval_and: tri moznosti:
1) implicitna chovanie pre "ostatne" pripady (CHYBNE!)
eval_and(true, true, true).
eval_and(_,    _,    false).
- program nie je deklarativne spravne, pretoze 2. klauzula
sa splni aj pre ?- eval_and(true,true, false). !
  - pri backtrackingu vrati nespravne riesenie
  -> kazda klauzula musi mat spravne podmienky (straz)

2) neuplny rozbor pripadov (deklarativne spravne, ale)
eval_and(true, true, true).
eval_and(false,_,    false).
eval_and(_,    false, false).
- volanie ?- eval_and(false, false, X). sa splni dvakrat
  - nasledny kod sa opakovane vyhodnocuje

3) (skoro) uplny rozbor pripadov
eval_and(true, true, true).
eval_and(true, false, false).
eval_and(false,_,    false).
- deklarativne aj proceduralne spravne
- argument, podla ktoreho robime rozbor pripadov
(a strukturalnu rekurziu) by mal byt vstupny

- eval - pokracovanie
1) kazdu spojku je mozne osetrit samostatne vs.
2) odvodene spojky popiseme pomocou uz znamych pripadov
% implikace
eval(A imp B, V):- eval(non A or B, V).

% ekvivalence
2 moznosti:
1) eval(A ekv B, V):- eval(A and B or non A and non B, V).
  - ale A a B sa vyhodnocuju dvakrat
2) eval(A ekv B, V):-
eval(A,VA),
eval(B,VB),
eval(VA and VB or non VA and non VB, V).
   % obecny! eval s jednoduchsimi spojkami
```

```
-- vyhodnocovanie s premennymi:
- aby sme mohli spocitat hodnotu vyrazu, potrebujeme vediet
hodnoty premennych. Tie predame v novom argumente, ako kontext.
- novy interface: eval(+Tab_prom, +BV, -Hodnota).
  - v tab prem. budeme hladat pomocou lookup a preto na konkretnej
strukture nezalezi (z hladiska textu programu)
  - kontext sa pri spracovani spojek predava nezmeneny do rekurzie
- premenne budeme v d.s. reprezentovat pomocou funktoru prom/1
ktory ma argument meno premennej.
  - nevhodne reprezentacia: za premenne by sme mohli povazovat
vsetky ine identifikatory nez konstanty a spojky, ale ...
  - premennu by sme odlisili len tak, ze by sme testovali, ci
nepatri medzi vyhradene identifikatory
    pr: eval(T, X, V) :- % X ma byt premenna
X \= true, X\= false, X\= and(_,_), ...
X \= non(_), vyhodnot_premennu(T,X,V).
  - pri pridani dalsej spojky, ktora by sa nahodou volala zhodne
s pouzivanou premennou by sa zavliekli chyby.
  - Q,A: vo vyrazoch nepouzivame Prologovske premenne, ale logicke
premenne, ktore su reprezentovane prologovskymi termami:
prom(meno)

eval(_T, false, false).
eval( T, true,  true ).
eval( T, A and B, V) :-
eval(T, A, VA), eval(T, B, VB),
eval_and(VA,VB,V). % bez zmeny
... dalsie klauzule analogicky
% novy pripad:
eval( T, prom(X), V) :- lookup(X,V,T).

- volanie, napr.:
?- eval([a-true,b-false,c-nedef], prom(a) and non prom(b), V).
  V = true

Q: je mozne reprezentovat hodnoty premennych predikatom v databazi?
- mozne to je, ale nevhodne.
  ak potrebujem vyhodnotit rovnaky vyraz pre dve ohodnotenia, musim
pracovat s databazou!

---------------
DC: (kompilator bv.)
- do zasobnikoveho stroja
  - je jednoduchsi nez RAM, pretoze nealokujem pamatove bunky.
  - premenne su alokovane pod svojimi menami -> nepotrebujem ich
ohodnotenie
comp(K, [load(const(K))]) :- K=true;K=false. % konstanty
comp(prom(X), [load(X)]).
comp(A and B, CC) :-
comp(A,CA), comp(B,CB),
mappend([CA,CB,[and]],CC).
```

```
comp(A imp B, CC):-
comp(A,CA), comp(B,CB),
mappend([CA,[non],CB,[or]],CC).
comp(A ekv B, CC):- % netrivialne (!)
comp(A,CA), comp(B,CB),
mappend([CA, % ...,VA zasobnik
 [dup,non], % VA,nVA
 CB, % VA,nVA,VB,
 [dup,non,take(3),and,  % VA,nVA,VB,nVAB
  drop(3),  % VA,VB,nVAB : chcem op. ne na vrcholu
     store(tmp),and,load(tmp), % VAB,nVAB
  or]
                ],CC).
- potrebujeme nejake specialne instrukcie pre ovladanie zasobniku:
  napriklad:
  dup duplikuje vrchol (= take(1))
  take(i) vytiahne i-tu polozku zvrchu a da ju na vrchol
  drop(i) zahodi i-tu polozku zvrchu
  - (kod aj instrukcie postupnym vyvojom)

(pr: uprava vyrazov)
---------------------------------------------------------------
7.cv
- grep(Str, Reg, Poloha) :- retazec Str odpoveda regularnemu vyrazu Reg
od zaciatku do dlzky Poloha.

pouzivane reg. vyrazy:
RV::= *(RV), % iteracia
RV+RV, % zjednotenie
        RV:RV, % sekvencia
[Znaky], %mnozina jednotlivych znakov
        lam, % prazdne slovo - lambda
        ff % nesplnitelny RV !

- nesplnitelny RV (empty) potrebujeme pre vystup vysledkov

grep: moznosti:
0) (kombinatory parserov)
1) priama interpretacia - backtrackingom => nasobne odpovede
2) konstrukcia konecneho automatu
3) kontrukcia vyrazu po prechode znakom (derivacia)

ad 2): vytvaranie prechodoveho stromu a stotoznenie rovnakych vyrazov

ad 3):
grep(Ss,RV,N):- grep1(Ss,RV,0,N). % zavedenie akumulatora
grep1(_Ss, RV, N,N):-acc_empty(RV). %koncovy pripad
grep1([S|Ss],RV,N,O):-
  next(S,RV,RV1), N1 is N+1,
  grep1(Ss,RV1,N1,O).

%next(X,RV,RV1): pocitanie derivacie RV podla X,
```

```
%  RV1 popisuje VSETKY zvysky po prijati X.
next(X,*(RV),RV1: *(RV)):-next(X,RV,RV1).
next(X,RV1:RV2,RV0):-next(X,RV1,R1),next(RV2,R2),
  (acc_empty(RV1) -> RV0 = R1:RV2+R2
                  ;  RV0 = R1:RV2    ).
next(X,RV1+RV2,R1+R2):-next(X,RV1,R1),next(X,RV2,R2).
next(X,Zn,RV0):-
  member(X,Zn) -> RV0 = lam % prijatie, zvysok prazdny
               ;  RV0=ff.   % odmietnutie, z. nesplnitelny
next(_,lam,ff). % nieco vratit musime, dovod zavedenia ff
next(_,ff,ff).

% acc_empty strukturalnou rekurziou
acc_empty(*(_)).
acc_empty(RV1+RV2):-acc_empty(RV1);acc_empty(RV2).
acc_empty(RV1:RV2):-acc_empty(RV1),acc_empty(RV2).
%acc_empty([_|_]):-fail.
acc_empty(lam).
%acc_empty(ff):-fail.

- zjednoduseni RV:
zjednodusia sa podvyrazy, potom sa pouzije pravidlo
%zjRV(RV, Zjednoduseny_RV)
zjRV(*(RV1),RV0):-zjRV(RV1,RV2),prav(RV2,RV0).
zjRV(RV1+RV2, RV0):-zjRV(RV1,R1),zjRV(RV2,R2),prav(R1+R2,RV0).
zjRV(RV1:RV2, RV0):-zjRV(RV1,R1),zjRV(RV2,R2),prav(R1:R2,RV0).
zjRV(RV1, RV0):-list(RV1), RV0= RV1.
zjRV(lam, lam).
zjRV(ff,ff).

- varianta: aby zjednodusovanie bolo deterministicke:
  namiesto volanie prav(I,O) volat one(prav(I,O)):
one(X):-call(X),!. % zakaze backtracking.

% zjednodusovacie pravidla by mali byt deterministicke
% ... ale nie su. Namiesto rezov: one(
prav(*(*(RV)),*(RV)). % rozbor pripadov do druhej urovne
prav(*(lam),lam).
prav(*(ff),ff).
prav(RV1+ff,RV1). % ff je neutralne pre zjednotenie
prav(ff+RV1,RV1).
prav(RV1:lam,RV1). % lam je neutr. pre sekvenciu
prav(lam:RV1,RV1).
prav(RV1:ff,ff). % ff je "nula" pre sekvenciu
prav(ff:RV1,ff).
prav(RV1+RV2,RV0):-list(RV1),list(RV2),RV0 is_set RV1\-/RV2.
% ... atd
prav((RV1+RV2)+RV3, RV1+(RV2+RV3)). % reasociacia
% dtto pre :
prav(X,X). % zachytne pravidlo, pre vsetky ostatne pripady
% toto sposobuje nedeterminizmus !
```

----

map a fold: pre binarne stromy, pre n-arne ...
- zmena myslenia: praca s celymi d.s.
- fold nie je dost obecne
  - minimum potrebuje fold1
  - vkladanie do stromu potrebuje pristup na vstupy
    - (tj. primitivna rekurzia), mozno obist, ale neefektivne
- specialne: fold niekedy nestaci a je nutne pouzivat interfaceovy predikat. Niekedy su vnutorne struktury pouzite pri rekurzii bohatsie nez to, co sa vyzaduje na vystupe. Napr. priemer zoznamu: musime pracovat s dvojicou suma,dlzka , a na konci vypocitat priemer.

... i ine
- prehl. stromu do sirky: vystupy su strukturovane po urovniach, na konci sa musime dodatocnej struktury zbavit.

----

kompilacia ofarbenia grafu DO Prologu

vstup: graf, orientovany, acyklicky, odzadu (hrany vedu do vrcholov, ktore uz su ofarbenem (tj. su pred! akt. vrcholom),
reprezentacia: sousede (tj. predchodcovia)
vystup: zoznam dvojic Meno_vrcholu - Farba ako vystupna datova struktura po spusteni ciela a prologovsky ciel, ktory
kazdemu vrcholu priradi Farbu (zdielanu cez premennu)
  - zoznam dvojich potrebujem, aby som vedel, ktora premenna odpoveda ktoremu vrcholu

- idea spracovania: farbenie/3 pre dany graf G vyrobi ciel C a Obarveni. Po spusteni ciela C sa v Obarveni vracaju mozne obarvenia.
Ciel C potrebuje run-time podporu (viz) a interface: barvy/1, ktory vracia povolene barvy backtrackingom

```
farbenie(graf2(G),C,Obarveni) :- farb1(G,C, [], Obarv).
farb1([], true, Obarv, Obarv).
farb1([V-Sousede|Vs],
(barvy(B),  % novy ciel; farba akt. vrcholu
         and_list(Kontrola),  % kontrola sousedu
 C1), % zvysny ciel
      Acc_obarv, Obarv) :-
  mk_kontrola(B,Sousede,Acc_obarv,Kontrola), %vyroba ciela
  farb1(Vs,C1,[V-B|Acc_obarv],Obarv). % rekurzia s novym Acc

%mk_kontrola :- vyroba ciela pre kontrolu:
mk_kontrola(B,Ss,Obarv,Kontrola):- % kontrola je seznam volani
  map(preklad_hrany(B,Obarv),Ss,Kontrola).
preklad_hrany(B,Obarv,S,Kontr):-
  lookup(S,Obarv,SB), % farba k susednemu vrcholu
  Kontr=(B\=SB). % ciel: spravnost ofarbenia hrany V(B)-S(SB)

% run-time support (alias lokalne ciele):
and_list([]).
and_list([X|Xs]):- call(X),and_list(Xs).
```

```
% varianta: prelozit a zavolat:
and_list(L):- preklad_and(L,C),call(C).
 %preklad_and(L,C):- fold(true, mk_and, L, C).
 % mk_and(A,B,(A,B)).
preklad_and([X],X). % optimalizovane
preklad_and([X|Xs],(X,CXs)):- preklad_and(Xs,CXs).

%interface: barvy musia byt nejak zadane!
barvy(B):- member(B,[1..101]).

Pr:
?- G=graf2([v1-[],v2-[v1],v3-[v1,v2]]),
   farbenie(G,C,Obarv).
 vystup: (mnou pomenovane premenne budu v realnom vystupe
len "_cisla")
Obarv= [v3-B3,v2-B2,v1-B1] % v opacnom poradi
C = barvy(B1),and_list([]), % and_list lze rozgenerovat
    barvy(B2),and_list([B2\=B1]),
    barvy(B3),and_list([B3\=B1,B3\=B2]),
    true

- and_list lze rozgenerovat; takto je to jednodussi
---------------------------------------------------------

zvysky 2000,2001
-OPAK:
  - backtracking vs. zoznam uspechov
  - metoda generuj a testuj
    - nie je vzdy vhodna
  - (navrh a) pouzivanie kniznic
    - zobecnenie predikatu
  - spravnost voci backtrackingu
  - efektivita
  - invariant (volania, uspechu)
  - konecnost vypoctu
  - map: obecny predikat, proceduralny param.

DC: 1) kombinacie
    2) prevod 2d stromu na zoznam s 2D klucom

ESTE
-- (zip(+I1,+I2,Out): mody (+, )
-- konstrukcia 2D stromu, 2D zoznamu
  - rozklad na triedy podla kluca, map (
-- transpozicia matice
-- kombinacie: opt. na nizkej urovni:
 - pocitat kolko prvkov este mozem vyhodit, aby som poznal
  situaciu, ked musim vziat cely zvysok.
  (idea: znizujem citac k nule)
-- rozkladne ulohy (?)

5.cv. 3.11.2000
```

```
Q: algebrogram
fibonnaciho cisla - logaritmicky: minule roky
  Q: rozdiel '='/2 a is/2:  vystup ako sucet jedniciek
     - typove chyby

DC:
1) rozdel,
  pouzitie rozdel pre prienik a zjednotenie
2) rozdel
  v com sa zmeni davkova konstrukcia v.d.s., ak mame kluce
premennej dlzky
---------------------------------------------------------------
6. cv, 10.11. 2000
plan:
interprety vyrazov
 bool, aritm, s premennymi, s funkciami, kompilator
and - or stromy
-------------------------------------------------------------------

Prolog:
1999
  1.cv: Univerzálny výpis pamäti; zložené d.š.
  2.cv (4.,8.10.): unárna repr. čísel: skladanie subst., akumulátor na
sčítaní a násobení, strukturálna rekurzia; otázka: ide sčítanie použiť
pre odčítanie? zavedenie zoznamov, dĺžka zoznamu (unárna) - obojsmerná,
zip dvojice zoznamov do zoznamu dvojíc (nejasná konvencia operátorového
'-') - rekurzia cez dva argumenty, aplikácia: tabulka druhých mocnín.
DC(po): a) vyrobit zoznam druhých mocnín k zoznamu čísel (tj. map_sqr)
b) transpozícia matice
(pá): otázky: d.š. datum/3, dôvody konštruktora (mk_datum), podtypový
polymorf. (procedúra na prevod z inej, bohatšej štruktúry);
rodinné vzťahy: súrodenci (vlastný a nevl.) sestra, teta, DC:
bratranec; unifikácia, occur check; unárna repr., plus a krát skladaním
subst., zložitosť $O(n)$ vs. $O(n^2)$ DC: ako sa chová krát, ak je najprv
sčítanie a potom rekurzívne volanie.
        3cv, 11.10'99: prefix, suffix, mappend, transpozícia, naviac:
flatten, zip?, kombinácie DC: podmnožiny, prienik, zjednotenie
(15.10) ++ výroba vyváženej haldy zo zoznamu, kombinácie, prienik DC:
komb s opakovaním, zjednotenie
        4.cv 18.10'99 prienik, zjednotenie množín, pre usporiadanu aj
neusporiadanu reprezentaciu, idea rozdielu, predikat is_set/2:
vyhodnocovanie mnozinových výrazov (s usporiadanými množinami), s
premennými, is_set ako interpret, zjednotenie usporiadaných mn. bolo
na prednáške ako súčasť
mergesort; aritmetika: fibonaciho čísla zostupne, DC fib.c. vzostuone
(s akumulatorom), idea f.c. logaritmicky: delenim intervalu, umocňovaním
matíc
(22.10.) is_set/2 a operacie, potom s premennymi, zjednotenie/rozdiel aj
akumulatorom, DC: relacne vyrazy s mnozinami, (nejasnosti okolo operatorov)
trie ako predkompilovany vyhladavaci strom pre AC alg., d.s. v tvare
vnorenych zoznamov, vyhladavanie, DC: rozmysliet konstrukciu.
5. cv. Po 25.10: suplovane: grafy (?)
```

pa: odpadlo.

6. cv. Po 1.11: precvicovanie vstavanych predikatov z prednasky:
- @<, ... neprecvicovane, pouzitie v triedeni
- xref: Pre dany term (s premennymi) vytvorit zoznam vsetkych funkcnych symbolov (s cetnostami v tvare F/N). Atomy su s cetnostou 0, premenne sa ignoruju. (pouzije sa funktor, =.., var, atomic, number). Vytvaranie zoznamu zdola, s pouzitim is_set. Varianty: dtto, ale aj s poctom vyskytov: namiesto is_set pouzit "is_bag" , ktore as postara aj o vyskyty. Rez: Klauzula, ktora sa stara o zlozene termy v troch variantach: 1) pomocou compound, 2) pomocou negativnych testov (notvar(T), \+ atomic(T), ...) 3) pomocou rezu v minulych klauzulach
- == : len idea: test vyskytu premennej v terme
- name: generovanie symbolov, pr.: generovanie mien premennych, ak chceme k riedkej reprezentacii termu vygenerovat vyraz s x1, x2, ...
- operatory: navrhnute operatory pre boolovske vyrazy
- precvicenie setof: prienik, (zjednotenie, rozdiel) mnozin napisany pomocou setof. Varianta: zjednotenie pre multimnoziny.

Pa: dtto

7.cv. Po 8.11. map, fold (nazývané na prednáške zpracuj) pre binárne stromy. Predikát size ako aplikácia fold. DC: map pomocou fold, map a fold pre n-árne stromy; pokus o väčší príklad: generovanie vyhľadávacieho stromu Aho-Corasicková.

Pa 12.11: grafy: dve reprezentacie, v d.š, interface vrchol a hrana, moznost reprezentacie v db, obsahuje meno grafu /nepoužívať/. Nadstavba: stupen, susedia vrcholu; prevody reprezentacii - pomocou setof. Problemy s dvomi reprzentaciami: interface ku konkretnej reprezentacii, alebo viac interface /a mozmost prevodu/, alebo lepsie /dynamicke/ typovanie: data si nesu typ sebou: potom interface pozna konkretny typ argumentu /d.s./

8.cv 15.11. Po ukoncenie Prologu: pisomka: z trie vytvorit usporiadany zoznam slov /tj. zoznamov pismen/. Potom Scheme: ZAklad rekurzie: sucet zoznamu, abstrakcia tejto funkcie, zavedenie cons, car, cdr: map, fold, faktorial pomocou enum /prevod intervalu daneho cislami na zoznam/ a fold. DC: map pomocou fold, zobecnenie enum: s parametrom pre "krok" /a testovanie konca/
Q: co je na FP deklarativneho
(pa): pisomka z Prologu; zoznamy, map, transpozicia matice, ...

9.cv. 22.11. otazky: apply, eval, '(1 2) vs (list 1 2): '(1 2) znamena d.s a nie funkciu k vyhodnoteniu, apply umoznuje volanie funkcii z neznamym poctom (tj. dynamicky zadanym) argumentov; transpozicia matice, map pre matice, map pre

PISOMKOVe priklady 15. a 19.11.1999:
Za neuspesne napisanie pisomky mate odovdzat niekolko prikladov. Vyberte si z lahkych zapoctovych (napr. 201,203,205,210,320,321...). Odovzdajte na papieri, za dva tyzdne (6.12. resp. 10.12.)

10-8 bodov: ziadny
  (typicky technicke chyby)
7-4 body: odovzdat jeden lahky zapoctovy priklad
  (typicke chyby v toku dat a predavani argumentov)
3-0 bodov alebo neucast: dva priklady a pisomkovy, spolu tri.
  (typicky nepochopenie rekurzie)

(po): Je dana datova struktura trie, ktora reprezentuje niekolko postupnosti pismen. Moze byt prazdna, konkretne atom void. Alebo neprazdna s funkcnym symbolom t/4 a argumentami lavy podstrom (podtrie), znak, vnoreny podstrom a pravy podstrom. Predpokladame a pouzivame usporiadanie na pismenach. Reprezentacia zoznamu postupnosti pomocou trie je takato: prazdny zoznam je void. Ak mame neprazdny zoznam, vyberieme si nejake pociatocne pismeno lub. postupnosti. Trie bude s funktorom t/4 a argumenty budu urcene takto. Vsetky postupnosti zacinajuce mensim pismenom su v lavom podstrome, vsetky zacinajuce vacsim pismenom su v pravom podstrome. Znak, tj. 2. arg., je vybrane pismeno. Vo vnoremom podstrome su reprezentovane vsetky postupnosti zacinajuce vybranym pismenom, ale bez tohoto pismena. Na vyvazenosti nezalezi. (V takto navrhnutej datovej strukture nemame explicitne urceny koniec kazdej postupnosti a preto nedokazeme reprezentovat sucasne postupnost aj jej prefix). Ulohou je k danemu trie vyrobit (usporiadany) zoznam postupnosti' pismen, ktory trie reprezentuje.

Idea: trie je vyhladavaci strom, v ktorom je treti argument naviac. Spracovanie je rekurzivne, pricom prazdny vnoreny podtrie sa osetri zvlast.

```
%main: prevod(Trie,Sez)
prevod(void,[]).          % odpoveda prazdnemu zoznamu postupnosti
prevod(t(L,X,void,R),V):- % odpoveda prazdnej postupnosti
  prevod(L,VL), prevod(R,VR),
  append([[X]],VR,V1), append(VL,V1,V).
prevod(t(L,X,P,R),V):-
  prevod(L,VL), prevod(R,VR),
  prevod(P,VP), pridej_prefix([X],P,VP),
  append(VP,VR,V1), append(VL,V1,V).
pridej_prefix(_,[],[]).
pridej_prefix(Pref,[X|L],[X0|L0]):-
  append(Pref,X,X0),
  pridej_prefix(Pref,L,L0).
```

(pi):
Je dany n-arny strom s funktorom s/2, kde argumenty su symbol a zoznam postromov v zhodnej reprezentacii. V zozname postromov sa moze vyskytovat spec. symbol void. Pre dany strom chceme vydat v jednom (usporiadanom) zozname vsetky cesty z vrcholu do kazdeho z vrcholov, listovych i medzilahlych. Kazda cesta je zoznam, ktory bude obsahovat dvojice: symbol vo vrchole podstromu a poradove cislo podstromu v zozname. Spec. symboly void v zozname podstromov pocitame do poradia, ale ony netvoria ziadne cesty vo vystupe. Pre koren na poradi nezalezi, napr. chceme na vystupe 0. Listy su podstromy, ktore maju zoznam symov prazdny. Nepouzivat setof a bagof.

Idea: Zapamatame si prefix, ktory popisuje cestu k aktualnym vrcholom v strome a ktory musime pridat ku kazdej ceste v aktualnych stromoch. Dalej si pamatame poradove cislo aktualneho podstromu, ktore sa inicializuje pri vstupe do zoznamu postromov.

Varianta: bez pamatania prefixu, ale zoznam vysledkov z podstromu musime spracovat pomocou pridej_prefix, viz minuly priklad.

```
% cesty(strom,cesty)
cesty(Strom,Cesty):-              % init
  cestyl([Strom],0, Pref, Cesty). % zoznam postromov, poradie, prefix,
                                        % Out
cestyl([], _N,_P,[]).   % vysledok skladanim medzivysledkov
cestyl([s(Sym,Pod)|Ss],N,P,C):-
  append(P,[Sym-N],P1), % zmena prefixu pre podstromy
  cestyl(Pod,1,P1,C1),  % podstromy akt. stromu
  N1 is N+1,
  cestyl(Ss, N1, P, C2 ), % bratia akt. stromu
  append([P1|C1],C2,C). % spojenie vysledkov, v poradi
cestyl([void|Ss],N,P,C):-
 %  append(P,[Sym-N],P1), % zmena prefixu pre podstromy
 % cestyl(Pod,1,P1,C1),  % podstromy akt. stromu
  N1 is N+1,
  cestyl(Ss, N1, P, C ). % bratia akt. stromu
 % append([P1|C1],C2,C). % spojenie vysledkov, v poradi

volanie:
?- cesty(s(a,[s(b,[]),void,s(c,[])]), C).
C=[[a-0],[a-0,b-1],[a-0,c-3]]

-------------
- abstrakcia a parametrizácia: zobecnenia faktoriálu (a čo to stojí!?):
dolná hranica, zoznam čísel (generovanie zoznamu: aritm. post., ľub.),
abstrahovanie od násobenia (-> foldl); stojí to: predávanie hodnot,
generovanie ( a iné pomocné), predávanie operácie ...
- program ako data, data ako program
- is-set ako interpret, varianty: multimnožiny, usporiadané množiny
-------------
2.cv; 1.10.1996
pr: % rodic/2
rodic(adam,bozena).
rodic(adam,bohus).
muz(adam).
zena(bozena).

- definujte je_rodic/1 ak má nejaké dieťa:
je_rodic(X):-rodic(X,_).
% _ je anonymná premenná

- zistite, či relácia/procedúra rodic/2 je definovaná
je_def_rodic :- rodic(_,_).
/* ako ukažka, že v jednom cieli môže byť viac (nezaviských)
     anonymných premenných */

- mama/2:
mama(X,Y):- rodic(X,Y),zena(X).

- (nevlastní) súrodenci
sourozenec(X,Y):-
     rodic(R,X),
```

```
     rodic(R,Y),
     X \= Y.     % X je rôzne od Y

- vlastní súrodenci
a) vl_sourozenci1(X,Y):-
     rodic(R1,X), rodic(R1,Y), X  \= Y,
     rodic(R2,X), rodic(R2,Y), R1 \= R2.
b) vl_sourozenci2(X,Y):-
     otec(R1,X), otec(R1,Y), X  \= Y,
     mama(R2,X), mama(R2,Y).

-DC: bratranec/sestřenice

----- 3.cv: 8.10.1996
definicia pristupovych procedur k datovym typom:
  d.s. datum(Rok,Mes,Den)   napr. datum(1996,10,1)
pristupove proc: get, set, make:
% get___
  get_rok(datum(R,_,_),R).
  get_mes(datum(_,M,_),M).
  get_den(datum(_,_,D),D).
% set___
% musia mat tri arg.: stary datum, novy datum a menenu informaciu
  set_rok(R,datum(_,M,D),datum(R,M,D)).
  set_mes...     % analogicky
% make_datum: vytvorenie zo zloziek:
  make_datum(R,M,D,datum(R,M,D)).
pr. pouzitia:
za_rok(Dnes,Za):-get_rok(Dnes,R), R1 is R+1, set_rok(R1,Dnes,Za).

---unárna reprezentácia čísel
0   ...   0
s(X)...   n+1, kde X repr. n

- test typu unárneho čísla (bolo na pred.)
     idea: rekurziou podľa štruktúry unárneho čísla
je_unarne_cislo(0).
je_unarne_cislo(s(X)) :- je_unarne_cislo(X).
! poučenie každá rekurzia musí končiť (faktom)

- sčítanie
a) rekurzia podľa prvého arg. (je možná aj podľa druhého)
     výsledok vzniká skladaním substitúcií
   plus1(  0, Y, Y).
   plus1(s(X),Y,s(Z)) :- plus1(X,Y,Z).
   volanie: ?- plus1( s(s(s(0))), s(s(0)), Z)
           Z  = s(s(s(s(s(0)))))
   /* Z vzniká skladaním substitúcií, postupne v rekurzii:
     ?- plus1( s(s(s(0))), s(s(0)), Z)
          Z = s(Z1), taky X1 = s(s(0)), Y1 = s(s(0))
     ?- plus1(  s(s(0)), s(s(0)), Z1) /* tj. plus1(X1,Y1,Z1) */
          Z1 = s(Z2), X2 = s(0), Y2=Y1=s(s(0)), proto
```

```
        Z = s(s(Z2))
    ?- plus1(    s(0), s(s(0)), Z2)
        Z2 = s(Z3), X3 = 0, Y3=Y2, proto
        Z = s(s(s(Z3)))
    ?- plus1(      0, s(s(0)), Z3)
        Z4 = Y4 = Y3 = s(s(0)), proto
        Z = s(s(s(s(s(0)))))
    konec

b) rekurzia podľa prvého arg., výsledok vzniká akumuláciou
   plus2(  0, Y, Y).
   plus2(s(X),Y,Z) :- plus2(X, s(Y), Z).
   volanie ako minule
   /* Y je akumulátor, v ňom sa akumuluje správne hodnota a na
     konci sa predá na výstup, Z je výstupná hodnota a počas
     rekurzie sa nemení */

- odčítanie
a) minus1(X,Y,Z):- plus1(Y,Z,X).
     prologovské pravidlá sú viacsmerové (občas).
   minus1a(X,Y,Z):-plus1(Z,Y,X).
     obdobne ide použiť plus2
b) priamo, rekurzia podľa druhého arg.
     minus2(X,0,X).
     minus2(s(X),s(Y),Z) :- minus2(X,Y,Z).
- nasobenie - pomocou scitania

-- poučenia: (z 3. cv)
- tok dát
- definícia "typov"
- štrukturálna rekurzia
- dva spôsoby získania výsledku
  - skladaním substitúcií; výpočtom z medzivýsledkov
  - akumulácia v určitom argumente

----------------------------------------------
--- 4.cv. 15.10.1996
- opakovanie z prednášok:
  append([],L,L).
  append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

operacia so zoznamami:
-- prefix(?Pref,+Zoz):- Pref je prefix zoznamu Zoz.
a) prefix([],_).
   prefix([X|Xs],[Z|Zs]):-prefix(Xs,Zs).
b) prefix(P,Z):- append(P,_,Z).

-- suffix(?S,+Z):- S je suffix (pripona) Z
a) suffix(S,S).
   suffix(S,[_Z|Zs]):-suffix(S,Zs).
b) suffix(S,Z):- append(_,S,Z).
   -- rekurzívny prefix/2 a suffix/2 je vlastne proojekcia na prvý a
```

```
tretí, resp. druhý a tretí argument u append/3.

-- podzoznam(?P,+Z):- P je podoznam Z
podzoznam(P,Z):- append(_,X,Z),append(P,_,X).

-- DC vybraná podpostupnosť, tj. vlastne všetky podmnožiny.
   subset(Sub, L):- ...

-- mappend(ZZ,Z):- ZZ je zoznam zoznamov, vsetky prvky ZZ (,ktore
su zoznamy,) chcem spojit do jednoho zoznamu Z v danom poradi.
Je to zobecnenie binárneho append na n-ticu vstupov.
mappend([],[]).
mappend([X|Xs],V):- mappend(Xs,V1), append(X,V1,V).
     %  v  tomto  pripade,   pretoze  vysledok  vznika  skladanim
substitucii, by rekurzivna klauzula mohla mat aj prehodene ciele.
Tretí argument by v tom prípade bol neukončený zoznam , tj mal na konci
volnú premennú, do ktorej by sa v rekurzii sunstutuovalo a nakoniec  by
sa uzavrela prázdnym zoznamom. Napr. pri volaní
?- mappend([[a],[b,c],[],[d]],L ) by L nadobúdalo postupne hodnoty
L = L'
L = [a|L'']
L = [a,b,c|L''']
L = [a,b,c|L'''']
L = [a,b,c,d|L''''']
L = [a,b,c,d]

-- DC: mappend pomocou akumulatora.

-- scitanie cisel v zozname:
% sum(Xs,S):- S je sucet/suma cisel zo zoznamu Xs
sum([],0).
sum([X|Xs],S):- sum(Xs,S1), S is S1 + X.

-- abstrakcia operácií v štrukturálnej rekurzii:
! vsimnite si "strukturalnu" podobnost medzi sum a mappend, lisia
sa  len   operaciou,  ktorou  sa   spracovavaju  medzivysledky  a
vysledkom spracovania prazdneho zoznamu.

-- flatten:  pozbieranie prvkov zo zoznamov  bez ohladu na uroven
vnorenia
flatten([],[]).  % spracovanie prazdneho zoznamu
flatten([H|T],V):- flatten(H,VH),  % spracovanie neprazd. zoznamu
                   flatten(T,VT),
                   appent(VH,VT,V).
flatten(A,[A]):-             % spracovanie jednoho prvku
     A \= [], A \= [_|_].    % ktory neni zoznam !

pr. volania:
flatten([1,[],[[a],b],[2]],V).
     V = [1,a,b,2];
     no
  - uvedomte si dolezitost testov v poslednej klauzuli
```

```
  - tento predikat nema jednoducho popisatelny "typ"

-- rekurzia cez viac vstupných argumentov.
  zip/3: dostáva dva vstupné zoznamy zhodnej dĺžky, vráti zoznam dvojíc.
  Dvojice sú pre naše účely štruktúry s funkčným symbolom - (mínus),
ktorý budeme zapisovať operátorovo, tj. X-Y namiesto -(X,Y).
  zip([],    [],    []).
  zip([X|Xs],[Y|Ys],[X-Y|Zs]):- zip(Xs,Ys,Zs).
  - predpokladáme (a overujeme), že zoznamy sú rovnako dlhé.
  - ak by sme chceli variantu, ktorá vydá výsledok podľa dĺžky
kratšieho, nahradili by sme koncové podmienky:
  zip([],_,[]).
  zip(X,[],[]):- X\=[].
  - Podmienka X\=[] zabezbečuje, aby pre roznako dlhé zoznamy (volanie
?-zip([],[],[]) to neuspelo dvakrát.
  - abstrakcia: predikát zip/3 vydáva pevnú datovú štruktúru. Ale
operácia, ktorou sa majú spojiť dve hlavy, vlastne nezávisí na rekurzii,
a mohla by sa predávať ako parameter. (Zatiaľ na to nemáme technické
prostriedky.) Procedúra by vyzerala:
*  zip(Oper,[],    [],    []).
*  zip(Oper,[X|Xs],[Y|Ys],[Z|Zs]):-
*    "Oper"(X,Y,Z),  % chybná syntax
*    zip(Oper,Xs,Ys,Zs).
Pre každú konkrétnu operáciu môžeme napísať variantu predikátu zip.
V kontrétnom prípade je operácia definovaná faktom mk_dvoj(X,Y,X-Y).

-- analogicky: dva výstupné zoznamy, napr. rozdelenie zoznamy na dva
zoznamy zhruba rovnakej dĺžky. Sú dve metódy delenia: a) sudé - liché,
b) prvá časť, druhá časť. Druhá metóda má výhodu, že zachováva poradie,
hodí sa teda pre usporiadané zoznamy (, keď to rekurzia o nich
predpokladá)

a) rozdel1([],[],[]).
   rozdel1([X],[X],[]).
   rozdel1([X1,X2|Xs],[X1|Ys],[X2|Zs]):- rozdel1(Xs,Ys,Zs).

  - dá sa implementovať aj trochu iným spôsobom, keď poradie výsledných
zoznamov prehadzujeme. Ktorý spôsob je (časovo) efektívnejší by sa
muselo zmerať.

a2) rozdel2([],[],[]).
    rozdel2([X|Xs],[X|Ys],Zs):-rozdel2(Xs,Zs,Ys).

b) Pri delení na prvú a druhú časť je možné spočítať dĺžku a tu potom
použiť na určťenie počtu prvkov v oboch častiach. (viz neskôr) Iná možnosť
je použiť
vstupný zoznam aj ako čítač dĺžky (v zvláštnom argumente). Ak z neho
budeme uberať prvky po dvojiciach, potom počet rekurzívnych vnorení bude
odpovedať počtu prvkov prvého zoznamu. Prípadný nepárny prvok necháme v
druhom zozname. Inicializácia čítača je v úvodnej klauzuli.
   rozdel3(Xs,Ys,Zs):-rozdel3_1(Xs,Ys,Zs,Zs).
   rozdel3_1(Xs,[],Xs,[]).
```

```
   rozdel3_1(Xs,[],Xs,[_Y]).
   rozdel3_1([X|Xs],[X|Ys],Zs,[_,_|Us],):-rozdel3_1(Xs,Ys,Zs,Us).
```

- DC: Ako máme zmeniť program, ak chceme, aby nepárny prvok bol v prvej časti?

-- transpozícia matice

Matica je zoznam zoznamov, pr jednoduchosť predpokladáme, že matica je korektne zarovnaná.

Je niekoľko možností, ako napísať transpozíciu matice.

a) Z danej matice vyberieme z každého riadku prvý prvok, tie dáme do zoznamu, ktorý bude tvoriť prvý riadok výsledku. ďalej pre rekurziu pripravíme "zostávajúcu" maticu, tj. čo ostane po ubraní prvých prvkov. Konkrétne chceme do zoznamu pozbierať telá všetnkých riadov matice, a výsledok spracovať rekurzívne. Obidva medzivýsledky môžeme získať počas jednoho prechodu maticou. Pri tomto spracovaní je vonkajšia rekurzia cez dĺžku riadkov, vnútorná cez počet riadkov.

b) Iný možnosť je prehodiť poradie rekurzií. Vonkajšia by spracovala vždy jeden riadok povodnej matice, vnútorná by zabezpečila prechod riadkom a pridanie každého prvku do odpovedajúceho zoznamu. Pretože dĺžka riadkov výsledku sa postupne zväčšuje, musíme si dať pozor, aby sme nezvýšili zložitosť (z O(n^2) na O(n^3)).

```
a) transp([[]|_],[]). % prázdna matica je zoznam prázdnych riadkov
   transp(M,[R|V]):-
     vytvor_radky(M,R,M1),
     transp(M1,V).
   vytvor_radky([],[],[]).
   vytvor_radky([[H|T]|M],[H|R],[T|M1]):-
     vytvor_radky(M,R,M1).

b) t2([R],V):- listify(R,M).
   t2([R|M],V):- t2(M,V1), zip(mk_list,R,V1,V).
   mk_list(H,T,[H|T]).
   listify([],[]).
   listify([H|T],[[H]|V]):-listify(T,V).
```

- použil som zobecnený zip s operáciou mk_list. Konkrétna verzia sa získa rozpisom.

- procedúra listify/2 vyrobí zoznam jednoprvkových zoznamov, tj. transpozíciu jednoho vektora. Podobne ako u zip/3 je možné z nej abstrahovať operáciu, ktorá vyjadruje reláciu jednoho prvku vstupu a výstupu. V tomto prípade to bola relácia mk_jednoprvkový(H,[H]).

-- kombinácia a spol. bez počítania

Kombinácie (a variácie) budeme vyberať z explicitne daného zoznamu prvkov - báze. V prípade kombinácií nezáleží na poradí, tj. môžeme si ho zvoliť. Vyberieme si poradie podľa báze. Počet prvkov výstupu môže byť daný číslom (viz nižšie), ale aj zoznamom s voľnými premennými príslušnej dĺžky. Dve rekurzívne klauzule umožňujú (nedeterministický) výber, či nejaký prvok do výsledku pridáme alebo nie.

```
  komb(_,[]).
```

```
  komb([X|Xs],[X|Ks]):- komb(X,Ks).
  komb([X|Xs],   Ks) :- komb(X,Ks).
alebo
  komb2(_,[]).
  komb2([X|Xs],[K|Ks]):- X=K, komb2(Xs,Ks)
                       ; komb2(Xs,[K|Ks]).

  - DC: bez opakovania

-- variacie bez opakovania:
/* vari(K,N,V):- pomocou backtrackingu vydava vo V vsetky
 * variacie o K prvkoch bez opakovania zo zoznamu N
 */
     vari(0,_N,[]).
     vari(K,N,[V|Vs]):- K>0, K1 is K-1,
           delete(V,N,N1),
           vari(K1,N1,Vs).

-- DC: variacie s opakovanim
-- ? kombinacie s/bez opakovania

--- 22.10.1996
DC: kombinacie bez opakovania:
% komb(K, N, V). N je zoznam prvkov, z ktorých vyberame K prvkov.
Výsledky sa vydavajú backtrackingom v premennej V.
     komb(0,_N,[]).
     komb(K,N,[V|Vs]):- K>0, K1 is K-1,
           append(_,[V|N1],N),     % podobne variaciam
           komb(K1,N1,Vs).
iná možnosť:
     komb1(0,_N,[]).
     komb1(K,[N|Ns],[N|Vs]):- K>0, K1 is K-1,
           komb1(K1,Ns,Vs).   % prvok N sme vybrali
     komb1(K,[_N|Ns],Vs):- K>0,
           komb1(K,Ns,Vs).    % prvok _N sme nevybrali

- fibonaciho cisla
  klasická definícia:
  fib1(N,F):-fib1a(N,F,F1). % N-té a (N-1)-té
  fib1a(1,1,_).
  fib1a(2,1,1).
  fib1a(N,F,F1):- N>2, N1 is N-1, fib1a(N1,F1,F2), F is F1+F2.

  DC: fib cisla akumulátorom, odspodu
  - a mnoho iných možností: po dvoch ...

  - logaritmicky:
    (funguje to, len ak sú počiatočné hodnoty 1 a 1, obecnejšie, viz
nižšie)
  Ūvaha: f_n= 1* f_{n-1} + 1* f_{n-2}
            = f_2 * f_{n-1} + f_1 * f_{n-2}
            = f_3 * f_{n-2} + f_2 * f_{n-3} = ...
```

```
           = f_i * f_{n-i+1} + f_{i-1} * f_{n-i}
  Špeciálne: (ak som to dobre spočítal)
   f_{2n+1} = f_{n+1}^2 + f_{n}^2
   f_{2n}   = f_{n+1}*f_n + f_{n} * f_n * f_{n-1}
            = f_n^2 + 2 * f_n * f_{n-1}
            = 2 * f_{n+1} * f_n - f_{n}^2

  potom v Prologu: (neodladené)
  fib2a(1,1,0).
  fib2a(2,1,1).
  fib2a(N,F,F1):- N>2, N1 is N//2,
               (N =:= 2*N1 -> fib2a(N1,F2,F3),  % alebo is/2 v podm.
                              F is F2^2+2*F2*F3,
                              F1 is F2^2 + F3^2  % ??
                            ; N2 is N1 +1,
                              fib2a(N2,F2,F3),
                              F is F2^2*F3^2,
                              F1 is 2*F2*F3 - F3^2
               ).

  - pôvodná myšlienka: násobenie matíc:
    <f_{n+1}, f_n> =  / 1  1 \  f_n
                      |      |
                      \ 1  0 /  f_{n-1}
    a násobiť rýchle matice už viete.
    - táto myšlienka (resp. spočítanie prvkov n-tej mocniny matice) sa
musí použiť, ak počiatočné hodnoty nie sú 1 a 1.

  - (použiť Knuth-Bendixov alg. na zistenie/overenie týchto vzorcov)

- mergesort - na prednáške
  - rozdelenie zoznamu na sude a liche

DC: rozdelenie usporiadaneho zoznamu na dve zhodne časti

-- (21.10.1999, pre Hrůzu, nedokončené)
-- príprava: Cvičenie: aritmetika, práca s maticami, dynamické
   programovanie
- úloha: Je daný (konvexný) n-uholník. Má sa nájsť jeho triangulácia,
ktorá má minimálny súčet dĺžok hrán.
  Idea: pre dva vrcholy i,j (v danom poradi) počítame hodnotu minimálnej
triangulácie od i do j. Az máme min. hodnotu pre (1,n), sme hotoví.
Minimálnu hodnotu pre (i,j) spočítame za predpokladu, že už poznáme
hodnoty pre kratšie podúseky takto:
  značenie: t(i,j) je hodnota min. triangulácie medzi i,j včetne hrany
(i,j). ďalej dĺžka hrany medzi i,j je d(i,j), v zmysle triangulácie, tj.
d(i,i+1)=0. Počiatočné podmienky: t(i,i+1)=0. (Možno som si neuvedomil
a nepopísal správnu okrajovú podmienku pre druhý okraj, tj. d(1,n)=0)
  Rekurzívny vzťah: t(i,j) = min(k=i+1 to j-1; t(i,k)+t(k,j)+d(i,j))
  Dynamické programovanie znamená, že opakujuce sa úlohy t(i,j) sa
nebudú znovu počítať (rekurziou).
  Dve možnosti naprogramovania: (a) explicitne počítanie matice, vonkajší
```

cyklus cez vzdialenost i-j, vnútorný cez i. (b) počítanie rekurziou, ale spočítané výsledky pre t(i,j) sa predávajú v akumulátore.

  - program pre (b)

%mt(vrcholy, hodnota, Stav)

- d.š.: vrcholy: zoznam súradníc X-Y, stavy sú zoznamy prvkov tvaru p(i,j,hodnota, deliaci_bod), deliaci bod je číslo vrcholu, pre ktoré s nadobudlo minima. Stavy sa predávajú ako akumulátor, vlastne explicitne ošetrujeme tabeláciu.

```
  mt(Vrch, N, H, S0) :- mt1(Vrch,1,N,H,[],S0).
  % mt1/4 počíta minimálnu hodnotu H triangulácie medzi i,j
  mt1(Vs,I,J,H,S1,S0):- K is I+1, mt2(Vs,I,J,K,0,H,S1,S0).
  % mt2/8 počíta vnútorný cyklus pre minimum cez K, 5. arg. je
akumulátor, dosial spocitané minimum
  mt2(Vs,I,J,J,)
  ...
```

-------------- 29.10.1996

DC:

```
rozdel(I,I1,I2):- length(I,N), N1 is N div 2, rozdeln(N1,I,I1,I2).
rozdeln(0,I,[],I).
rozdeln(N,[I|Is],[I|I1s],I2):- N>0, N1 is N-1,
     rozdeln(N1, Is, I1s, I2).
```

- vybudovanie vyvazeneho stromu z usporiadaneho zoznamu:

```
sez2str([],void).
sez2str([X],t(void,X,void)).
sez2str(Sez,t(Str1,X,Str2)):-
     rozdel(Sez,Sez1,[X|Sez2]),
     sez2str(Sez1,Str1),
     sez2str(Sez2,Str2).
```

písomka:

1. Daný je binarny vyhľadavací strom. Chceme získať usporiadaný zoznam všetkých prvkov zo stromu pri prechode zľava doprava.

```
  ako obvykle: strom je prazdny - void
                  alebo neprázdny - t(L,X,R)
```

2. Daná je vyhľadávacia štruktúra trie, v ktorej sú uschované postupnosti.

trie môže byť prazdne - void, alebo neprázdne t(L,X,N,R), kde v L sú všetky postupnosti začínajúce prvkom menším než X, R absahuje postupnosti začínajúce prvkom väčším než X a N obsahuje postupnosti začínajúce prvkom X, ktoré sú bez uvodného X.

(trie je niekoľkourovnôvý vyhľadávací strom, v ktorom sa vyhľadáva najprv podľa prveho prvku s prechodmi do L a R, až sa najde prvé písmeno X, prejde sa do štruktúry N a hľadá sa zbytok postupnosti bez X.)

trie pre [1],[3,2],[3,4],[5]

poznámka: takéto štruktúry obvykle nepíšete ručne do zdrojáku ale generujú sa ako (medzi)výsledky programu.

```
  t( t( void, 1, void, void),      % [1]
     3, t( void,2,void,            % [3,2]
```

```
        t(void,4,void,void))    % [...,4]
    t( void, 5, void, void))     % [5]
```

--- 5.11.1996 : grafy

--- 12.11.1996
zadávanie zápočtových príkladov

--- 19.11.1996
Analýza vstupu:
Je zadaný zoznam vstupných symbolov v predspracovanej podobe.

```
Vstupné symboly sú štruktúry:      c(Cislo)
                                   a(Atom)
                                   '('
                                   ')'
```

Zoznam obsahuje zápis úplne uzátvorkovaného termu s binárnymi operátromi. Každý atom môže byť operátor a pretože výraz je úplne uzátvorkovaný, nepotrebujeme precedencie.
Pr. vstupu [c(1),a(+),'(',a(b),a(-),c(2),')']
chceme výraz analyzovať na na výstup vydať nejakú jeho strukturovanú reprezentáciu, napr.
1) term 1+(b-2)
2) prefixný zápis [+, 1, [-, b, 2]]

--- 26.11.1996
DC: plánovanie procesov
Je k dispozícii jeden procesor, na ktorom máte vykonať niekoľko úloh. V jednom čase smie byť na procesore vykonávaná len jedna úloha. U každej úlohy je daná dĺžka trvania a čas dokončenia, ak tento čas prekročíte, platíte pokutu. Časy sú zadané prirodzenými číslami v časových jednotkách. Program má naplánovať úlohy tak, aby celková pokuta za všetky úlohy bola minimálna.
Pokuta za jednu úlohu je lineárna funkcia, za jednotku oneskorenia platíte jednu (korunu).
Problém riešte prehľadávaním možných rozvrhov s priebežným počítaním ceny.

Ťažšia varianta 1: je daná celková pokuta, ktorú nesmiete prekročiť. Túto informáciu sa snažíte použiť počas výpočtu.

Ťažšia varianta 2: U každej úlohy je stanovené iné penále za jednotku času.

Ľahšia varianta 3: Pokuta je stanovená jednorázovo za prekročenie. Snažíme sa minimalizovať počet úloh, ktoré sa nestihnú.

Začiatok nakoniec
1. cv.
opakovanie OOP

```
dedičnosť, zapuzdrenie, polymorfizmus
reprezentacia Tabulky Virtualnych Metod
staticke a dynamické alokacia, priradenie a zmena struktúry
koercia, preťaženie, polymorfizmus podtypový a parametrický
zmena typu za behu (explicitná)


2.cv (1.10.1996)
príklad: univerzálny vypisovač a čítač štruktúr
Vytvoriť (pod)systém, ktorý umožní vypísať ľubovoľnú d.š. do
textového súboru tak, aby sa dala spätne zrekonštruovať.
omedzenia: typy sú známe (Pascal je staticky/syntakticky typovaný)
- vypis elementárnych dát: "typ hodnota"
- výpis polí: "pole velkost hodnoty"
  - čo s neinicializovanými časťami ?
- výpis záznamov: "typ hodnoty"
  - typ musí byt známy programu, ktorý to číta
- výpis pointrov: to je to zaujímavé
  - viac pointrov môže viesť na ten istý objekt
  - dve možnosti:
     - ukazatele v súbore na pevnú adresu (nepoužiteľné)
     - mená objektov a ukazatele sa nahradia menami (aj číslo je
           meno)
  - objekt sa pri prvom výskyte ukazateľa naň zaregistruje v
     globálnej tabuľke "adresa, meno" , vypíše sa meno; a pri
     každom výpise ukazateľa najprv prezeráme tabuľku, či tam už
     pointer není.
  - čítanie: dvojprechodové
     - v prvom prechode alokujeme pamäť, tvoríme dvojice
           "adresa,meno" a čítame nepointrové hodnoty, v druhom
           doplňujeme pointry. V tabulke si môžeme držať adresy,
           kde potrebujeme doplniť pointer, až ho budeme poznať.
  - lepšie riešenie: jednoprechodové čítanie
     - potrebujeme najprv alokovať všetkých následníkov nejakého
           objektu. Ale je problém s cyklickými štruktúrami.
     - riešenie: objekt rozdelíme na dve časti - "hlavička"
           obsahuje info nutné pre alokáciu (typ a veľkosť) a
           vypisuje sa pri prechádzaní do hĺbky v pre-order a
           vlastné data sa vypisujú v post-order. Všetko klape.
     - čítanie: podľa hlavičky naalokujeme a zaradíme do tabuľky,
           potom čítame podriadené objekty a datami objekt
           naplníme až nakoniec, keď už sú známe aj hodnoty
           pointrov.
- zanorenie do OOP
  - všetky objekty v systéme sú potomkami jednej univerzálnej
     triedy s virtuálnymi procedúrami
     - zaregistruj_sa do tabuľky
     - vypíš_sa: zaregistruj_sa, vypíš hlavičku, vypíš
           pointrových následníkov, vypíš seba/svoje položky
           (prípadne pomocou volania výpisu (len) položiek predka
           v hierarchii)
  - podobne sa číta (??) : hlavička udáva typ, takže po alokácii
```

```
     by sa už dala volať virtuálna procedúra, ktorá to načíta.
- prechod štruktúry
     - do hĺbky: obracanie pointrov (s markovacími bitmi)
     - do "šírky": alg. slepého
     - -''- : alg. jednookého - lokálne do hĺbky s omedzeným
           zásobníkom, dôležitá optimalizácia posledného volania
- použitie
     - (kopírujúci) garbage collector
     - univerzálny výpis štruktúr, ktorý sa dá prezerať a
           editovať, takto sa dajú pripraviť počiatočné alebo
           pevné štruktúry (pre programy riadené datami)
       - pr: štruktúra menu namiesto alokácie v programe.
           a) objekty, ktoré si predávajú riadenie
                 vzniknú explicitnou konštrukciou
                 vzniknú pomocou vnorených konštruktorov
           b) špec. jazyk, ktorý sa skompiluje
           c) viz. - datové štruktúru popídané v datách programu
                 (samozrejme ovládacie procedúry musia byť v
                 programe)


--- zbytky:
- sčítanie výrazu:
je daná reprezentácia výrazu  pomocou unárnych čísel: 3+3+2,
resp. +(3,+(3,2))
a)   % pravoasociatívne, strukturálne podľa 1.arg, "výpočet
     % z medzivýsledkov"
     secti(s(X),s(X)).
     secti(0,0).
     secti(+(I,Is),V0) :- secti(Is,V1), plus(V1,I,V0).
a1) DC - pre ľavoasociatívnu d.š.
b)   % ľubovoľne asociatívne
     ...
     secti(+(I1,I2),V) :-
           secti(I1,V1),
           secti(I2,V2),
           plus(V1,V2,V).
c)   % akumulácia, pre ľubovoľne asociatívnu d.š.
     secti(I,V):- secti_1(I,0,V).
     secti_1(0,V,V).
     secti_1(s(I),V1,V0) :- plus(s(I),V1,V0).
     secti_1(+(I1,I2),V1,V0) :-
           secti_1(I1, V1, V2),
           secti_1(I2, V2, V0).
c1) DC - pre pravo/ľavo asoc. štruktúru sčítanie akumuláciou.

---pre danú premennú (z výrazu) chceme vypustiť všetky faktory
a zistiť súčet jej koeficientov.
% secti(+Prem, +Výraz, -Koef, -Výraz)
! výraz sa nedá meniť na miest, musíme ho zmeniť novým zložením z
     podčastí
idea: rekurzia podľa 2. arg. (podvýrazov)
```

```
% ukončenie
secti1(_X, 0, 0, 0).
secti(X, +(*(C,X),V)

--- Rok 1995
--- z cvičení:
--- 18.10.1995, 4. cv.
- zložené datové typy
- pr: datum
  - správne navrhnutý typ:
     datum(Rok,Mesiac,Den)  % alebo v opačnom poradí
    položky sú určené polohou, nie menom
  - nesprávne
     [Rok, Mesiac, Den]      % unifikuje sa s kdejakou trojicou
     datum([Rok, Mesiac, Den])
     datum(rok(Rok),Mesiac(Mesiac),den(Den))
       % zbytočne zložité a neefektívne

- k typu patrí interface:
  - môže sprostredkovať výber, ukladanie a zmeny d.š. podľa mena
     položky
  make_datum(R,M,D,Datum) :- Datum = datum(R,M,D).
  get_datum(den, Datum, D):- Datum = datum(_,_,D).

  % set_datum(den, D, Datum, Novy_Datum) - v Datume zmeni den na
  % D, výsledok musí byť nová štruktúra Nový_Datum; ! v Prologu
  % není zmena na mieste (tj. deštruktívne priradenie)
  set_datum(den, D, datum(R,M,_), datum(R,M,D))
  ...   % get_ a set_ analogicky pre mesiac, rok


--- 25.10.1995, 5. cv
- návrh boolovských výrazov
  - konštanty ano/0, ne/0
  - spojky and/2, or/2, non/1, ...
- test ,,typu" boolovského výrazu (is_bv/1)
  - pomocou štrukturálnej rekurzie
- vyhodnotenie b.v. - na ano, ne (eval_bv/2)
  - rozšírenie testu typu o argument a procedúry na jeho
     ošetrovanie
  - vyhodnocovanie spojek - tabulkou:
    a) rozbor prípadov
    b) skratky, problémy s rezom
  - prevod zložitých spojek na jednoduchšie
    - implikácia na non a or
    - ekvivalencia na dve implikácie
      - ! z hľadiska efektivity: nevyhodnocovať podvýrazy dvakrát
- zavedenie premenných do reprezentácie b.v.
  var(I) ... var je nový tag, aby sme mohli mať aj premenné ano,
     ne, and ...
  - rozšírenie is_bv - pridaním klauzule
  - rozšírenie eval_bv
```

```
    - pridaním argumentu - kontextu, ktorý prenáša hodnoty
      premenných (ako zoznam dvojíc)
- DC (domáce cvičenie)
  prevod b.v. do konjunktívnej normálnej formy
- union/3 - zjednotenie množín reprezentovaných zoznamami
  - tri varianty analogické prieniku z prednášok
    a) deklaratívna
    b) s rezom
    c) s -> (if_then_else)
- lepšia reprezentácia množím - usporiadané zoznamy
  - union je v takom prípade merge/3
  (môžete si skúsiť pomocou merge napísať mergesort/2)
- množinové vyhodnocovanie: is_set/2 (narychlo)
  - ako v Pascale: + je zjednotenie, * prienik, - rozdiel

% budiž is_set tiež vhodný binárny operátor
  M is_set M1+M2 :- V1 is_set M1,   % rekurzia,
                    V2 is_set M2,   % prirodzene.
                    union(V1,V2,M). % (post)spracovanie výsledkov
  M is_set M1*M2 :- ... ,
                    intersection(V1,V2,M).
  M is_set M     :- is_list(M).  % test ,,typu", zvládnete?
% M is_set ...    môžeme rozširovať o ďalšie operácie ...
  ?- X is_set ([1,2]+[3,4])*[4,1].
     X = [1,4]  % odpoveď, v nejakom poradí

  (22.10'99)
  - mnozinovy rozdiel:
  rozdil1([],_Ys,[]).
  rozdil1([X|Xs],Ys,Zs):-
    (member(X,Ys)-> Zs=Zs1 ; Zs=[X|Zs1]),
    rozdil1(Xs,Ys,Zs1).

  - pomocou akumulatora:
  rozdil2(Xs,Ys,Zs):-rozdil2x(Xs,Ys,[],Zs).
  rozdil2([],_Ys,Zs,Zs).
  rozdil2([X|Xs],Ys,A1,Zs):-
    (member(X,Ys)-> A2=A1 ; A2=[X|A1]),
    rozdil1(Xs,Ys,A2,Zs).

  - vsimnite si, ze v prvom pripade je zoznam zvasceny
o X v hlave (virtualne), v druhom pripade v rekurzivnom volani.
  - v pripade zjednotenia mozeme pouzit Ys ako akumulator
(pretoze ma spravnu pociatocnu hodnotu)
  - add: symetricky rozdiel ako prevedenie na minuly pripad
syntax pre symetricky rozdiel: --(X,Y)
  is_set(V, --(X,Y)):- is_set(V,(X+Y)-(X*Y)). % samozrejme neefektivne.


--- 1.11.1995, 6. cv.
- grafy:
  dve rozumné reprezantácie
```

```
  (jedna nerozumná - v databázi)
  - vrchol(Graf, V), hrana(Graf, X, Y)
    - nemá slúžiť ako reprezentácia, ale ako interface
- naprogramovanie interface k daným dvom reprezentáciám
- prevody medzi preprezentáciami
  - niektoré operácie sa lepšie robia v jednej reprezentácii, iné
     v druhej; niekedy je efektívnejšie zmeniť reprezentáciu,
     vykonať operáciu a vrátiť reprezentáciu
- minimálna kostra:
  - rozšírenie druhej reprezentácie o ohodnotenie hrán
  - algoritmus:
    usporiadanie hrán (preskočené)
    a potom pridávanie
    - pomocný predikát:
    % cesta(Z, Do, Graf, Cesta) cesta je dosial prejdená časť
    cesta(X, X, _, _).
    cesta(X, Y, G, C):-
      not(member(X,C)),            % aby sme sa nezacyklili
      (hrana(G,X,Z); hrana(G,Z,X)) % nájdenie susedov
      cesta(Z,Y,G,[Z|C]).          % rekurzia
    - volanie: ?- cesta(1,99, graf2(...), [1]).
    - prehl. do hlbky, s testom proti zacykleniu - C
    - ak sa chceme dozvediet cestu, musíme pridať jeden
      (výstupný) argument

--- 8.11.1995, 7. cv.
--- Príklad :
Dijkstrov algoritmus nájdenia najkratších ciest z daného vrcholu
do všetkých ostatných

- reprezentácia grafu
  - nepodstatná, podstatný je interface
  - napr. funktor graf/2,
    graf(ZoznamVrcholov, ZoznamHran)
    - ZoznamHran -> zoznam trojíc Vrchol-Vrchol-CenaHrany

- interface nízky:     hrana(Graf, X,Y,H)
  vydáva pri backtrackingu postupne všetky hrany
     Musí odpovedať zvolenej reprezentácii

- interface vyšší:
  okolie(Graf, X, L) :-      % okolie vrcholu X v grafe G bude v L
    setof(Y-H, hrana(Graf, X,Y,H), L).

- vlastný alg.
?- dijkstra(Graf, Vrchol, Výsledný_zoznam).
   Výsledný_zoznam je v tvare zoznamu dvojíc: Vrchol-Vzdialenosť

dijkstra(G,V,L):-      % interface na predikát, ktorý bude
  d(G,[],[V-0],L).     % počítať

- Idey výpočtu: d(Graf, Uzavrete, Otvorene, L)
```

```
  - Graf je vstup
  - L je výstup
  - stav prehľadávania grafu je možné popísať dvomi stavovými
premennými:
  množina uzavretých vrcholov - sú známe konkrétne ceny do V
  množina otvorených vrcholov - sú známe odhady ceny
  (v grafe môžu byť aj ďalšie vrcholy - ešte neznáme)

  rekurzívna klauzula pre d/4 vykoná ,,lokálnu" zmenu stavu:
  - vyberie a expanduje (rozvinie) jeden vrchol a podľa jeho
okolia upraví stavové premenné

d(G,L,[],L).      % prehľadávame všetky vrcholy,
                  % končíme, až nie je žiadny otvorený
d(G, Uz, Otv, L):-      % vlastná rekurzia
     zvol_v(Otv, V-H, Otv1), % voľba vrcholu, zvyšok otv. budeme
                             % potrebovať (v Otv1)
     okolie(G, V, L1),       % expanzia zvoleného V
     vypust(L1, Uz, L2)      % z okolia L1 vypustíme uzavreté
     zluc(L2, Otv1, Otv2, H), % nové otvorené Otv2
     d(G, [V-H|Uz], Otv2, L). % ďalšia iterácia s novými hodnotami

- poznámky:
  popísaný algoritmus je obecný a podľa voľby vrcholu je možné
    implementovať rôzne stratégie prehľadávania grafu:
  - prioritné (to je aj naše)
  - do hĺbky, do šírky, ...

,,pomocné predikáty"
zvol_v([X],X,[]).      % X = V-H
zvol_v([V1-H1|L1], V-H, L):-
  zvol_v(L1, V2-H2, L2),     % rekurzia
  ( H2 =< H1 ->              % porovnanie ako test v if->then;else
     V = V2, H = H2, L=[V1-H1|L2]      % ,,výstupné" unifikácie
  ;  V = V1, H = H1, L=[V2-H2|L2]      % ,,výstupné" unifikácie
  ).

vypust([], _, []).           % v podstate množinový rozdiel
vypust([V-H|L1], L2, L3):-
  ( member(V-_,L2) ->
    L3 = L4            % výstupné unifikácie, vznikajú skladaním
  ; L3 = [V-H|L4]      % substitúcií, L4 sa spočíta v rekurzii
  ),
  vypust(L1,L2,L4).

zluc([],L,L,_).        % podobné množinovému zjednoteniu
zluc([X1|L1],L2,L3,H):- % H je ,,kontext", cena ,,aktuálneho" vrch.
  X1 = V1-H1,
  ( delete(V1-H2, L2, L4) -> % test a príprava menšieho L2 do L4
    L3 = [V1-Hmin|L5],       % výstupná unifikácia
    Hmin is min(H1+H,H2)    % voľba novej vzdialenosti (minima)
                            % min je v LPA aj funkcia
```

```
  ; L4 = L2             % delete neuspelo
    L3 = [X1|L5]        % výstupná unifikácia
  ),
  zluc(L1,L4,L5,H).
```

- zapisovanie predikátov v takom tvare, že rekurzívne volanie je
posledným cieľom v klauzuli, je vhodné, pretože umožní
interpretu v niektorých prípadoch zmeniť rekurziu (tj. rekurzívnu
prácu so zásobníkom) na iteráciu (tj. prácu s niekoľkými
,,stavovými" premennými v konštantnej pamäti)

- Grafika v LPA Prologu
Pred spustením prologu z úrovne systému:

```
C:> gfx               inštalovanie podpory
C:> exit              odinštalovanie ...
```

v Prologu: načítať súbory: z adresára ...\prolog\gfx\

```
?- ensure_loaded(grafix),    % grafika
   ensure_loaded(turtle).    % ,,želví" grafika a la LOGO

?- go.     % spustenie želví grafiky
```

- želva má určitú aktuálnu polohu, smer a farbu pera:
  príkaz go okrem spustenia nastaví štandardnú polohu:
  na začiatku je poloha (0,0) - stred obrazovky
                smer 0 - na sever, nahor
                farba pera 15 - biela
- príkazy želví grafiky:

```
  go
  draw(Vzdialenost)    % nakreslí čiaru z akt. pozície v akt.
                             smere akt. perom
  right(Uhol)          % zmeni akt. smer doprava o daný
                             počet stupňov
  left(Uhol)           % analogicky, ale otáča sa doľava
  do(N, Cieľ)          % opakuje Cieľ N-krát
  for(Index, Od, Do, Krok, Cieľ)
                       % opakuje Cieľ pre rôzne hodnoty premennej
                             Index
  pen( Farba)          % nastaví farbu pera, 1..16
  move(Vzdialenosť)    % pohyb bez kreslenia
  goto(X,Y)            % presun na danú polohu
  head(Smer)           % otočenie do daného smeru
```

----- prevzate z prednasok
--- Reprezentácie grafov
     - a použitie bagof ...
- ukážeme si tri možné reprezentácie grafov, ktoré sa dajú
používať pre grafy s ohodnotením aj bez ohodnotenie vrcholov
a hrán. Sú možné aj iné reprezentácie, ale obecne, pretože nemáte
polia, niektoré reprezentácie sú neprirodzené.
1) graf je reprezentovaný štruktúrov, ktorá obsahuje zoznam
   vrcholov a zoznam hrán

```
   (použité funkčné symboly "graf", "-" a "hr" sú zvolené
    ľubovolne, ale používať sa musia konzistentne)
     d.š.: graf([Vrchol-Hodnota], [hr(Vrch1,Vrch2,Hodnota)])
     pr. graf([a-1, b-2, c-4],
              [hr(a,b,11), hr(a,c,1.1), hr(b,c, "Hodnota")] )

2) graf je reprezentovaný ako zoznam trojíc (alebo dvojíc - bez
ohodnotenia) Vrchol, Hodnota, Zoznam_susedov
  d.š.: [Vrch1 - Hodnota_vrcholu - [Vrch2 - Hodn_hrany]]
  pr.: [a-1-[b-11,c-1.1],
        b-2-[c- "Hodnota"],
        c-[]]

3) graf je uložený v databáze. Toto riešenie je nevýhodné,
pretože zmena alebo vytvorenie grafu znamená zásah do databáze.
Na druhej strane, myšlienku z tejto reprezentácie môžeme použiť
ako (nízky) interface nezávislý na konkrétnej reprezentácii.
V tekom prípade musíme napísať procedúry, ktoré umožnia prístup
k skutočnej reprezentácii cez zvolený interface.
     % reprezentácia: (nepoužívať)
     vrchol(g1, a, 1).
     vrchol(g1, b, 2).
     hrana(g1, a, b, 1.1).

  - interface:
     vrchol(+Graf, ?Vrch1, ?Hodnota_vrcholu).
     hrana(+Graf, ?Vrch1, ?Vrch2, ?Hodnota_hrany).
   graf je vstupný parameter, je to reprezentácia podľa 1) alebo
     2)
   výsledky sa vracajú backtrackingom

   - implementácia interface pre repr. 2):
     vrchol(G,V,H):-
           member(V - H - _, G).
     hrana(G,V1,V2,H):-
           member(V1-_-Vs, G),      % určí / hľadá prvý vrchol
           member(V2-H, Vs).        % -''- druhý a hodnotu hrany
     % ak nemáme ohodnotenie v repr.
     vrchol(G,V):-
           member(V-_,G).
     hrana(G,V1,V2):-
           member(V1-Vs,G),
           member(V2,Vs).

 - na základe nízkeho interface môžeme vybudovať ďalšie pomocné
procedúry, už nezávislé na interface
Procedúry vrchol a hrana (nízky interface) vracajú výsledky
pomocou backtrackingu. Z jednotlivých výsledkov potrebujeme
vytvoriť datové štruktúry, preto použijeme setof, bagof...

   - okolie: k vrcholu vytvorí zoznam jeho susedov
     % okolie(Graf, Vrch, Sousede):- S je zoznam všetkých
```

```
          susedov vrcholu Vrch
      okolie(G,V,S):- bagof(V2,hrana(G,V,V2), S).
     - volanie:
      ?- okolie(..., V,S).
            V=a   S=[b,c];
            V=b   S=[c];
            no                % riesenie V=c nevydá, pretože
                                    S sa nevracia prázdne

   - repr2: vytvorenie reprezentácie 2) z interface
      % repr2(+G,-R):- R je vytvorená reprezentácia grafu G
      repr2(G,R):-
        bagof(V1-S, (vrchol(G,V1),
                     (okolie(G,V1,S) -> true
                                     ;  S = [] )),
              R).
      - pozn.:
        - bez ohodnotenia
        - 2. argument, cieľ, je zložený
        - je ošetrený prípad, ak S = []

   - vytvorenie reprezentácie 1) z interface, príklad použitia
            existenčných premenných
      % repr1(G,R):- R je reprezentácia grafu G
      repr1(G, graf(Vs,Hs)):-
            bagof(V, HV^vrchol(G,V,HV), Vs),
            bagof(hr(V1,V2), HH^hrana(G,V1,V2,HH), Hs).

- príklad: Dijkstrov algoritmus
% dijkstra(+G,+V,-L):- pre každý vrchol V2 v grafe G spočíta
% vzdialenost od V  a vráti zoznam L dvojích V2-Vzdialenost
dijkstra(G,V,L) :-
     d(G,[], [V-0], L).
d(G,Uz,Otv,L):-
     zvol_v(otv, V-H, Otv1),
     okolie(G,V,L1),
     vypust(L1,Uz,L2),
     zluc(L2,Otv1,Otv2, H),
     d(G,[V-H|Uz], Otv2,L).
d(G,L,[],L).     % ak sa nepodarí zvolit vrchol

% zvol_v :- vyberie vrchol s minimalnou docasnou hodnotou
zvol_v([X],X,[]).
zvol_v([V1-H1|L1], V-H,L):-
     zvol_v(L1,V2-H2,L2),
     ( H2 =< H1 -> V = V2, H = H2, L=[V1-H1|L2]
                 ; V = V1, H = H1, L=[V2-H2|L2]
     ).

% množinový rozdiel, bez ohladu na hodnotu
vypust([],_,[]).
vypust([V-H|L1],L2,L3):-
```

```
     ( member( V-_,L2) -> L3 = L4
                        ;  L3 = [V-H|L4] ),
     vypust(L1,L2,L4).

% cosi ako zjednotenie
zluc([],L,L,_).
zluc([X1|L1],L2,L3,H):-
     X1 = V1-H1,
     ( delete(V1-H2,L2,L4) ->
           L3 = [V1-HM|L5], HM is min(H1, H2+H)
         ; L3 = [X1|L5], L4 = L2),
     zluc(L1,L4,L5,H).
---------------------------------
Dodatky:
A. reprezentácia dát v algoritme Aho-Corasicková
B. fibonacciho čísla ...

A. Reprezentácia dát v algoritme Aho-Corassicková
(autotest na 2. prednáške)
  Reprezentácia stroja sa skladá z doprednej funkcie F, spätnej funkcie
g a výstupnej fce. out. Stavy sú implicitné, číslované.
Sú tri možné reprezentácie: 1) strukturovaná, 2) bytecode a 3)
vykonateľný kód. štrukturovaná má niekoľko možností konkrétnej
štruktýry: a) riedky vektor b) hustý
vektor, c) dvojvrstevný vektor (podľa stavov a abecedy), d) pointrová
štruktúra (pointry ako viazané premenné - hack), e) štrukturovaná podľa
f (trie)
  príklad: vzorky he, her, his, she
stavy 0h1e2"he"r3"her", 1i4s5"his", 0s6h7e8"she,he", g:5->6, 7->1, 8->2
  Bytekód je symbolický (v C++ možné), štrukturovaný, a s symbolickými
pointrami (nemáme pole).

reprezentácie:
1) dvojúrovňový zoznam, prvá úroveň podľa stavu, druhá podľa písmena
(riedka). $íslo stavu je meno, na ktoré sa odkazujeme.
% d.š. [stav(Meno,Následníci,G, [Out_vzorky])]
[stav(0,[h-1,s-6],0,[]),
 stav(1,[e-2,i-4],0,[]),
 stav(2,[r-3],0,["he"]),
 stav(3,[],0,["her"]),
  ...
 stav(8,[],2,["she","he"]) ]

2) zcela rozpísané:
- toto je "interface", impl. môže to byť v DB alebo v d.š (ako u grafov)
f(0,h,1). f(0,s,6). f(1,e,2). ...
g(0,0). g(1,0). ... g(5,6). g(6,0). g(7,1). g(8,2). % g(0,0) sa nevyužije
% prípadne g(_,0). % nutné ošetriť v programe
% vlastný interpret; volanie ?- ac(0,Chars, Out). ) ako poč. stav
ac(S,[C|Cs], Out):- f(S,C,S1),!, Out=[O1|Out1], ac(S1,Cs,Out1).
ac(0,[C|Cs], [[]|Out]):- /* g(0,0)) špeciálne, pretože uberám písmeno*/
                        ac(0,Cs,Out).
```

```
ac(S,Cs,Out):- S\=0,g(S,S1),ac(S1,Cs,Out).

3) vykonateľný kód: (Prolog-ify)
 - pre jednoduchosť: ku každému znaku vstupu bude zoznam nájdených
vzorkov na tomto znaku.
ac(Chars,Out):- f_0(Chs,Out).
f_0([],[]). %ukončenie
f_0([C|Cs],Out):- C=h, f_1(Cs,Out)
                ; C=s, f_6(Cs,Out).
f_0([_|Cs],[[]|Out]):- /* _\=h,s */
                       f_0(Cs,Out).
f_4([s|Cs],[[his]|Out]) :- f_5(Cs,Out).
f_5(Cs,Out) :- g_5(Cs,Out).
...
g_5(Cs,Out):- f_6(Cs,Out).
...

4) previazaná (cyklická) štruktúra
 - pointry sú "priame", číslo stavu nepotrebujeme
S0=s([h-S1,s-S6],S0,[]),
S1=s([e-S2,i-S4],S0,[]), ...
Strom=[S0,S1,S2,...] % potrebujeme
volanie: ?- ac(S0, Chars,Out).
ac(S,[C|Cs],Out):- next(S,C,S1,Out1), append(Out1,Out2, Out),
  ac(S\,Cs,Out2). % rozdielové zoznamy (pre zmenu)
ac(_,[],[]).
next(s(Next,G,Out),C,S1,Out1):-
  member(C-S1,Next) -> Out1= ...
                     ; next(G,C,S1,Out1).
```

B. Fibonacciho čísla.
Exponenciálne. Zapamätanie posledného: lineárne - zdola, zhora. Omedzenie dvoch počiatočných podmienok - stačí jedna (s dvoma hodnotami - jedna je umelá, dopočítaná). Počítanie po dvoch (s výberom na konci alebo začiatku). Pomocou násobenia matíc: logaritmické!, motivačne: počítania súbežne spredu aj zozadu.

- iterátory, for -cyklus ,...
- program ako data, a naopak ...