Vstup/výstup,
databázové operace,
rozklad termu

Čtení ze souboru

```
process_file( Soubor ) :-
        seeing( StarySoubor ),          % zjištění aktivního proudu
        see( Soubor ),                  % otevření souboru  Soubor
        repeat,
           read( Term ),                % čtení termu  Term
           process_term( Term ),        % manipulace s termem
           Term == end_of_file,         % je konec souboru?
        !,
        seen,                           % uzavření souboru
        see( StarySoubor ).             % aktivace původního proudu

repeat.                                 % vestavěný predikát
repeat :- repeat.
```

Predikáty pro vstup a výstup

```
| ?- read(A), read( ahoj(B) ), read( [C,D] ).
|: ahoj. ahoj( petre ). [ ahoj( 'Petre!' ), jdeme ].
A = ahoj, B = petre, C = ahoj('Petre!'), D = jdeme

| ?- write(a(1)), write('.'), nl, write(a(2)), write('.'), nl.
a(1).
a(2).
yes
```

- `seeing, see, seen, read`
- `telling, tell, told, write`
- standardní vstupní a výstupní stream: `user`

Příklad: vstup/výstup

Napište predikát `uloz_do_souboru( Soubor )`, který načte několik fakt ze vstupu a uloží je do souboru Soubor.

```
| ?- uloz_do_souboru( 'soubor.pl' ).
|: fakt(mirek, 18).
|: fakt(pavel,4).
|:
yes
| ?- consult(soubor).
% consulting /home/hanka/soubor.pl...
% consulted /home/hanka/soubor.pl in module user, 0 msec
% 376 bytes
yes
| ?- listing(fakt/2). % pozor:listing/1 lze použít pouze při consult/1 (ne u compile/1)
fakt(mirek, 18).
fakt(pavel, 4).
yes
```

Implementace: vstup/výstup

```
uloz_do_souboru( Soubor ) :-
        seeing( StaryVstup ),
        telling( StaryVystup ),
        see( user ),
        tell( Soubor ),
        repeat,
           read( Term ),
           process_term( Term ),
           Term == end_of_file,
        !,
        seen,
        told,
        tell( StaryVystup ),
        see( StaryVstup ).

process_term(end_of_file) :- !.
process_term( Term ) :-
        write( Term ), write('.'), nl.
```

Databázové operace

- Databáze: specifikace množiny relací
- Prologovský program: **programová databáze**, kde jsou relace specifikovány explicitně (fakty) i implicitně (pravidly)
- Vestavěné predikáty pro změnu databáze během provádění programu:

`assert( Klauzule )`	přidání `Klauzule` do programu
`asserta( Klauzule )`	přidání na začátek
`assertz( Klauzule )`	přidání na konec
`retract( Klauzule )`	smazání klauzule unifikovatelné s `Klauzule`

- Pozor: `retract/1` lze použít pouze pro dynamické klauzule (přidané pomocí `assert`) a ne pro statické klauzule z programu
- Pozor: nadměrné použití těchto operací snižuje srozumitelnost programu

Databázové operace: příklad

Napište predikát vytvor_program/0, který načte několik klauzulí ze vstupu a uloží je do programové databáze.

```
| ?- vytvor_program.
|: fakt(pavel, 4).
|: pravidlo(X,Y) :- fakt(X,Y).
|:
yes
| ?- listing(fakt/2).
fakt(pavel, 4).
yes
| ?- listing(pravidlo/2).
pravidlo(A, B) :- fakt(A, B).
yes
| ?- clause( pravidlo(A,B), C). % clause/2 použitelný pouze pro dynamické klauzule
C = fakt(A,B) ?
yes
```

Databázové operace: implementace

```
vytvor_program :-
        seeing( StaryVstup ),
        see( user ),
        repeat,
           read( Term ),
           uloz_term( Term ),
           Term == end_of_file,
        !,
        seen,
        see( StaryVstup ).

uloz_term( end_of_file ) :- !.
uloz_term( Term ) :-
         assert( Term ).
```

Konstrukce a dekompozice termu

- Konstrukce a dekompozice termu

  `Term =.. [ Funktor | SeznamArgumentu ]`

  ```
  a(9,e) =.. [a,9,e]
  Cil =.. [ Funktor | SeznamArgumentu ], call( Cil )
  atom =.. X ⇒ X = [atom]
  ```

- Pokud chci znát pouze funktor nebo některé argumenty, pak je efektivnější:

  `functor( Term, Funktor, Arita )`

  ```
  functor( a(9,e), a, 2 )
  functor(atom,atom,0)    functor(1,1,0)
  ```

  `arg( N, Term, Argument )`

  ```
  arg( 2, a(9,e), e)
  ```

Rekurzivní rozklad termu

- Term je proměnná (`var/1`), atom nebo číslo (`atomic/1`) ⇒ konec rozkladu
- Term je seznam (`[_|_]`) ⇒

  procházení seznamu a rozklad každého prvku seznamu
- Term je složený (`=../2`, `functor/3`) ⇒

  procházení seznamu argumentů a rozklad každého argumentu
- Příklad: ground/1 uspěje, pokud v termu nejsou proměnné; jinak neuspěje

```
ground(Term) :- atomic(Term), !.          % Term je atom nebo číslo NEBO
ground(Term) :- var(Term), !, fail.       % Term není proměnná NEBO
ground([H|T]) :- !, ground(H), ground(T). % Term je seznam a ani hlava ani tělo
                                          %      neobsahují proměnné NEBO
ground(Term) :- Term =.. [ _Funktor | Argumenty ], % je Term složený
                ground( Argumenty ).              % a jeho argumenty
                                                  % neobsahují proměnné
```

```
?- ground(s(2,[a(1,3),b,c],X)).
no
```

```
?- ground(s(2,[a(1,3),b,c])).
yes
```

subterm(S,T)

Napište predikát `subterm(S,T)` pro termy S aT bez proměnných, které uspějí, pokud je S podtermem termu T. Tj. musí platit alespoň jedno z

- podterm S je právě term T NEBO
- podterm S se nachází v hlavě seznamu T NEBO
- podterm S se nachází v těle seznamu T NEBO
- T je složený term (`compound/1`), není seznam (`T\=[_|_]`), a S je podtermem některého argumentu T.
  - pokud nepoužijeme (`T\=[_|_]`), pak dotaz `:- subterm(a,[1]).` cyklí!
  - pokud nepoužijeme (`compound/1`), pak dotaz `:- subterm(1,2).` cyklí!

```
| ?- subterm(sin(3),b(c,2,[1,b],sin(3),a)).                    yes
```

subterm(S,T)

Napište predikát subterm(S,T) pro termy S aT bez proměnných, které uspějí, pokud je S podtermem termu T. Tj. musí platit alespoň jedno z

- podterm S je právě term T NEBO
- podterm S se nachází v hlavě seznamu T NEBO
- podterm S se nachází v těle seznamu T NEBO
- T je složený term (compound/1), není seznam (T\=[_|_]), a S je podtermem některého argumentu T.
  - pokud nepoužijeme (T\=[_|_]), pak dotaz `:- subterm(a,[1]).` cyklí!
  - pokud nepoužijeme (compound/1), pak dotaz `:- subterm(1,2).` cyklí!

```
| ?- subterm(sin(3),b(c,2,[1,b],sin(3),a)).                    yes
```

```
subterm(T,T) :- !.
subterm(S,[H|_]) :- subterm(S,H), !.
subterm(S,[_|T]) :- subterm(S,T),!.
subterm(S,T) :- compound(T), T\=[_|_], T=..[_|Argumenty], subterm(S,Argumenty).
```

same(A,B)

Napište predikát same(A,B), který uspěje, pokud mají termy A a B stejnou strukturu. Tj. musí platit právě jedno z

- A i B jsou proměnné NEBO
- pokud je jeden z argumentů proměnná (druhý ne), pak predikát neuspěje, NEBO
- A i B jsou atomic a unifikovatelné NEBO
- A i B jsou seznamy, pak jak jejich hlava tak jejich tělo mají stejnou strukturu NEBO
- A i B jsou složené termy se stejným funktorem a jejich argumenty mají stejnou strukturu

```
| ?- same([1,3,sin(X),s(a,3)],[1,3,sin(X),s(a,3)]).                yes
```

same(A,B)

Napište predikát same(A,B), který uspěje, pokud mají termy A a B stejnou strukturu. Tj. musí platit právě jedno z

- A i B jsou proměnné NEBO
- pokud je jeden z argumentů proměnná (druhý ne), pak predikát neuspěje, NEBO
- A i B jsou atomic a unifikovatelné NEBO
- A i B jsou seznamy, pak jak jejich hlava tak jejich tělo mají stejnou strukturu NEBO
- A i B jsou složené termy se stejným funktorem a jejich argumenty mají stejnou strukturu

```
| ?- same([1,3,sin(X),s(a,3)],[1,3,sin(X),s(a,3)]).                    yes
```

```
same(A,B) :- var(A), var(B), !.
same(A,B) :- var(A), !, fail.
same(A,B) :- var(B), !, fail.
same(A,B) :- atomic(A), atomic(B), !, A==B.
same([HA|TA],[HB|TB]) :- !, same(HA,HB), same(TA,TB).
same(A,B) :- A=..[FA|ArgA], B=..[FB|ArgB], FA==FB, same(ArgA,ArgB).
```

D.Ú. unify(A,B)

Napište predikát unify(A,B), který unifikuje termy A a B a provede zároveň *kontrolu výskytu*.

```
| ?- unify([Y,3,sin(a(3)),s(a,3)],[1,3,sin(X),s(a,3)]).
X = a(3)      Y = 1       yes
```

D.Ú. unify(A,B)

Napište predikát unify(A,B), který unifikuje termy A a B a provede zároveň *kontrolu výskytu*.

```
| ?- unify([Y,3,sin(a(3)),s(a,3)],[1,3,sin(X),s(a,3)]).
X = a(3)     Y = 1      yes
```

```
unify(A,B) :- var(A), var(B), !,  A=B.
unify(A,B) :- var(A), !, not_occurs(A,B), A=B.
unify(A,B) :- var(B), !, not_occurs(B,A), B=A.
unify(A,B) :- atomic(A), atomic(B), !, A==B.
unify([HA|TA],[HB|TB]) :- !, unify(HA,HB), unify(TA,TB).
unify(A,B) :- A=..[FA|ArgA], B=..[FB|ArgB], FA==FB, unify(ArgA,ArgB).
```

not_occurs(A,B)

Predikát not_occurs(A,B) uspěje, pokud se proměnná A nevyskytuje v termu B.

Tj. platí jedno z

- B je atom nebo číslo NEBO
- B je proměnná různá od A NEBO
- B je seznam a A se nevyskytuje ani v tělě ani v hlavě NEBO
- B je složený term a A se nevyskytuje v jeho argumentech

not_occurs(A,B)

Predikát not_occurs(A,B) uspěje, pokud se proměnná A nevyskytuje v termu B.

Tj. platí jedno z

- B je atom nebo číslo NEBO
- B je proměnná různá od A NEBO
- B je seznam a A se nevyskytuje ani v tělě ani v hlavě NEBO
- B je složený term a A se nevyskytuje v jeho argumentech

```
not_occurs(_,B) :- atomic(B), !.
not_occurs(A,B) :- var(B), !, A\==B.
not_occurs(A,[H|T]) :- !, not_occurs(A,H), not_occurs(A,T).
not_occurs(A,B) :- B=..[_|Arg], not_occurs(A,Arg).
```