Cvičení 12

12.1 Opakování

Příklad 12.1.1 Určete, které dotazy uspějí:

a) `?- X = 1.`
b) `?- X == 1.`
c) `?- X =:= 1.`
d) `?- X is 1 + 1.`
e) `?- X = 1, X == 1.`
f) `?- X = 1 + 1, X == 2.`
g) `?- X = 1 + 1, X =:= 2.`
h) `?- g(X, z(X)) = g(Y, Z).`
i) `?- a(a, a) = X(a, a).`
j) `?- a(1, 2) = b(1, 2).`

Příklad 12.1.2 Opravte chyby v následujícím programu a vylepšete jeho nevhodné chování (bez použití CLP). Měl by fungovat správně pro celá čísla a můžete předpokládat, že první argument je vždy plně instanciován.

```
fact(N, Fact) :-
    M #= N - 1, fact(M, FactP), Fact #= N * FactP.
fact(0, 1).
```

Příklad 12.1.3 Napište predikát `merge/3`, který spojí 2 vzestupně uspořádané seznamy čísel z prvního a druhého argumentu. Výsledný vzestupně uspořádaný seznam pak unifikuje do třetího argumentu. Například dotaz `merge([1,2,4], [0,3,5], X)` uspěje se substitucí `X = [0,1,2,3,4,5]`.

Poté naprogramujte predikát `mergesort/2`, který seřadí seznam čísel z prvního argumentu podle velikosti s využitím techniky *merge sort* a výsledek unifikuje s druhým argumentem. Může se vám hodit i pomocný predikát `split/3`, který rozdělí zadaný seznam na 2 seznamy stejné délky.

Příklad 12.1.4 Napište predikát `quicksort/2`, který seřadí seznam v prvním argumentu metodou *quick sort* a výsledek unifikuje do druhého argumentu. Pravděpodobně se vám bude hodit i pomocný predikát `split/4`, který podle zadaného pivota rozdělí seznam na prvky menší než zadaný pivot a prvky větší nebo rovny než tento pivot.

Příklad 12.1.5 S využitím Prologu vyřešte Einsteinovu hádanku.

Zadání

- Je 5 domů, z nichž každý má jinou barvu.
- V každém domě žije jeden člověk, který pochází z jiného státu.
- Každý člověk pije jeden nápoj, kouří jeden druh cigaret a chová jedno zvíře.

- Žádní dva z nich nepijí stejný nápoj, nekouří stejný druh cigaret a nechovají stejné zvíře.

Nápovědy

1. Brit bydlí v červeném domě.
2. Švéd chová psa.
3. Dán pije čaj.
4. Zelený dům stojí hned nalevo od bílého.
5. Majitel zeleného domu pije kávu.
6. Ten, kdo kouří *PallMall*, chová ptáka.
7. Majitel žlutého domu kouří *Dunhill*.
8. Ten, kdo bydlí uprostřed řady domů, pije mléko.
9. Nor bydlí v prvním domě.
10. Ten, kdo kouří *Blend*, bydlí vedle toho, kdo chová kočku.
11. Ten, kdo chová koně, bydlí vedle toho, kdo kouří *Dunhill*.
12. Ten, kdo kouří *BlueMaster*, pije pivo.
13. Němec kouří *Prince*.
14. Nor bydlí vedle modrého domu.
15. Ten, kdo kouří *Blend*, má souseda, který pije vodu.

Úkol Zjistěte, kdo chová rybičky.

Postup

a) Uvažme řešení uvedené v souboru `12\_einstein.pl`. Soubor naleznete v ISu a v příloze sbírky.
b) Pochopte, jak by měl program pracovat. Vysvětlete, proč na dotaz `?- rybicky(X).` program zdánlivě cyklí.
c) Přeuspořádejte pravidla tak, aby Prolog našel řešení na tento dotaz do jedné vteřiny.
d) Odstraňte kategorizaci objektů (`barva/1`, `narod/1`, `zver/1`, `piti/1`, `kouri/1`) a požadavky na různost entit v každé kategorii. Diskutujte, v jaké situaci, by vám tato kategorizace byla prospěšná.
e) Nyní sdružte entity stejného typu do seznamů. Modifikujte program tak, aby místo predikátu `reseni/25` používal predikát `reseni/5`.
f) Podobnou modifikaci proveďte i s jednotlivými pravidly, tj. místo `ruleX/10` použijte `ruleX/2` a místo `ruleX/5` použijte `ruleX/1`.
g) Definujte pomocné predikáty `isLeftTo/4`, `isNextTo/4`, `isTogetherWith/4`, které prověřují požadované vztahy, například: `isNextTo(A, B, Acka, Bcka)` je pravdivý, pokud se objekt `A` vyskytuje v seznamu `Acka` na pozici, která sousedí s pozicí objektu `B` v seznamu `Bcka`. Tímhle způsobem přepište všechna pravidla.
h) Obdivujte krásu výsledku po vašich úpravách.

Jiná kódování Přeformulujte program tak, aby predikát `reseni/5` jako své argumenty bral seznamy objektů odpovídající jednotlivým pozicím v ulici, tj. 5 seznamů takových, že každý seznam bude obsahovat informaci o barvě domu, národnosti obyvatele, chovaném zvířeti, oblíbeném kuřivu a oblíbeném nápoji obyvatele.

Příklad 12.1.6 Napište predikát `equals/2`, který uspěje, pokud se dva zadané seznamy rovnají. V opačném případě vypíše důvod, proč je tomu tak (jeden seznam je kratší, výpis prvků,

které se nerovnají, ...). Můžete předpokládat, že seznamy neobsahují proměnné. Příklady použití najdete níže.

```
?- equals([5,3,7], [5,3,7]).
true.
?- equals([5,3,7], [5,3,7,2]).
1st list is shorter
false.
?- equals([5,3,7], [5,2,3,7]).
3 does not equal 2
false.
```

Příklad 12.1.7 Nechť je zadána databáze faktů ve tvaru `road(a, b)`, které vyjadřují, že z místa `a` existuje přímá (jednosměrná) cesta do místa `b`. Napište predikát `trip/2` tak, že dotaz `?- trip(a, b).` uspěje, jestli existuje nějaká cesta z místa `a` do místa `b`. Můžete předpokládat, že cesty netvoří cykly (tedy pokud jednou z nějakého místa odejdete, už se tam nedá vrátit).

Rozšíření 1
Upravte vaše řešení tak, že predikát `trip(a, b, S)` uspěje, jestli existuje nějaká cesta z místa `a` do `b` a `S` je seznam měst, přes které tato cesta prochází (v daném pořadí).

Rozšíření 2
Upravte vaše řešení tak, aby fungovalo i v případě, že cesty mohou tvořit cykly.

12.2 Logické programování s omezujícími podmínkami

Příklad 12.2.1 Vyřešte následující logickou úlohu. Jednotlivá slova reprezentují čísla, každé písmeno zastupuje jednu číslici, různá písmena představují různé číslice.

$$SEND + MORE = MONEY$$

Řešení využívá logické programování s omezujícími podmínkami v konečných doménách, nezapomeňte proto do programu zahrnout také načítání správné knihovny pomocí následujícího řádku:

```
:- use_module(library(clpfd)).
```

Příklad 12.2.2 Vyřešte následující algebrogram. Jednotlivá slova reprezentují čísla, každé písmeno zastupuje jednu číslici, různá písmena představují různé číslice. Nalezený výsledek přehledně vypište na obrazovku.

$$DONALD + GERALD = ROBERT$$

Příklad 12.2.3 Vyřešte pomocí Prologu následující algebrogram:

$$\begin{array}{ccccc} \mathrm{KC} & + & \mathrm{I} & = & \mathrm{OK} \\ + & & + & & + \\ \mathrm{A} & + & \mathrm{A} & = & \mathrm{KM} \\ = & & = & & = \\ \mathrm{OL} & + & \mathrm{KO} & = & \mathrm{LI} \end{array}$$

Příklad 12.2.4 Napište predikát `fact/2`, který počítá faktoriál pomocí CLP. V čem je lepší než klasicky implementovaný faktoriál?

Příklad 12.2.5 S využitím knihovny `clpfd` implementujte program, který spočítá, kolika a jakými mincemi lze vyskládat zadanou částku. Snažte se, aby řešení s menším počtem mincí byla preferována (nalezena dříve).

Příklad 12.2.6 Uvažte problém osmi dam. Cílem je umístit na šachovnici 8×8 osm dam tak, aby se žádné dvě neohrožovali. Dámy se ohrožují, pokud jsou ve stejném řádku, sloupci nebo na stejné diagonále.

S pomocí knihovny `clpfd` napište predikát `queens/3`, který dostane v prvním argumentu rozměr šachovnice, ve druhém výstupním argumentu bude jako výsledek seznam s pozicemi sloupců, do kterých třeba v jednotlivých řádcích umístit dámy, a ve třetím argumentu bude možné ovlivnit způsob hledání hodnot (predikát `labeling/2`).

Příklad výstupu:

```
?- queens(8, L, [up]).
L = [1,5,8,6,3,7,2,4]
...
```

Příklad 12.2.7 Napište program, který nalezne řešení zmenšené verze Sudoku. Hrací pole má rozměry 4×4 a je rozdělené na 4 čtverce 2×2. V každém řádku, sloupci a čtverci se musí každé z čísel 1 až 4 nacházet právě jednou. Jako rozšíření můžete přidat formátovaný výpis nalezeného řešení.

Příklad 12.2.8 Stáhněte si program `12\_sudoku.pl`. Soubor naleznete v ISu a v příloze sbírky.

a) Program spusťte a naučte se jej ovládat. Zamyslete se, jak byste funkcionalitu programu sami implementovali.
b) Prohlédněte si zdrojový kód programu a pochopte, jak funguje.
c) Modifikujte program tak, aby nalezená řešení splňovala podmínku, že na všech políčkách hlavní diagonály se vyskytuje pouze jedna hodnota.

12.3 Bilingvální opakování

Příklad 12.3.1 V jazyce Haskell naprogramujte rekurzivní funkci `app`, která se chová jako knihovní funkce `++`. Dále v jazyce Prolog naprogramujte odpovídající rekurzivní predikát `app/3`, který se chová jako knihovní predikát `append/3`.

V jazyce Haskell naprogramujte funkci `concat'`, která se chová jako knihovní funkce `concat`. Ekvivalentní predikát `concat/2` napište i v jazyce Prolog. *Pro pokročilejší:* V obou implementacích `concat` zkuste použít tail-rekurzi.

Příklad 12.3.2 V jazyce Haskell naprogramujte funkci `listAvg`, která pro neprázdný seznam čísel vrátí jeho aritmetický průměr. V jazyce Prolog naprogramujte odpovídající predikát `listAvg/2`.

Pro pokročilejší: Zkuste v obou jazycích `listAvg` implementovat tak, aby došlo jen k jednomu průchodu seznamu. V jazyce Haskell můžete zkusit použít vhodný `fold`; v jazyce Prolog akumulátory.

Příklad 12.3.3 V jazyce Haskell naprogramujte funkci `removeDups`, která všechny opakované výskyty libovolného prvku bezprostředně po sobě nahradí pouze jedním výskytem. Například

```
removeDups [1,1,1,2,2,3,1,1,4,4] = [1,2,3,1,4]
```

V jazyce Prolog naprogramujte odpovídající predikát `removeDups/2`.

Příklad 12.3.4 V jazyce Haskell naprogramujte funkci `mergeSort`, která seřadí zadaný seznam pomocí algoritmu *Merge sort*. Může se vám hodit nejprve implementovat pomocné funkce `split` a `merge`.

V jazyce Prolog naprogramujte odpovídající predikát `mergeSort/2`. Zkuste vhodným způsobem použít řezy.

Příklad 12.3.5 V jazyce Haskell naprogramujte funkci `primeFactors`, která pro zadané číslo vrátí seznam čísel, která odpovídají jeho prvočíselnému rozkladu.

Například

```
primeFactors 2 = [2]
primeFactors 4 = [2,2]
primeFactors 6 = [2,3]
primeFactors 30 = [2,3,5]
primeFactors 40 = [2,2,2,5]
```

V jazyce Prolog naprogramujte odpovídající predikát `primeFactors/2`.

Příklad 12.3.6 V jazyce Haskell naprogramujte funkci `quickSort`, která seřadí zadaný seznam pomocí algoritmu *Quick sort*.

V jazyce Prolog naprogramujte odpovídající predikát `quickSort/2`. Zkuste vhodným způsobem použít řezy.

Řešení

Řešení 12.1.1

a) Ano, dojde k unifikaci se substitucí `X = 1`.
b) Ne, `X` je neinstanciována proměnná, `1` je číslo/atom, tedy nejde o stejné termy. Unifikace se při `==` neprovádí.
c) Dojde k chybě – při aritmetickém porovnání musí být obě strany plně instanciovány.
d) Ano, dojde k aritmetickému vyhodnocení pravé strany a unifikaci výsledku `2` s proměnnou `X`.
e) Ano, po unifikaci `X` s `1` porovnání termů uspěje, protože porovnání bere ohled na dříve provedené substituce.
f) Ne, nedojde k aritmetickému vyhodnocení, a tedy tyto výrazy představují různé termy.
g) Ano, dojde k aritmetickému vyhodnocení a porovnání uspěje.
h) Ano, se substitucí `Z = z(X), X = Y`.
i) Ne, nelze použít proměnnou na místě funktoru. (I když tento cíl lze dosáhnout pomocí predikátů `functor/3`, `arg/3` nebo `=../2`).
j) Ne, jde o různé funktory.

Řešení 12.1.2 Prvním problémem je, že program nelze přeložit a způsobuje ho nesprávné použití operátoru `=`, který jenom unifikuje obě strany, ale aritmeticky nevyhodnocuje. Nahradíme ho tedy `is`.

Dále, takto napsaný program nikdy nekončí, protože vždy se použije pravidlo a k ukončujícímu faktu nikdy nedojde. Je tedy třeba prohodit fakt a pravidlo.

Teď už dostaneme výsledek. Avšak program stále nefunguje například pro dotaz `?- fact(1, 2).` V tomto případě se nikdy nepoužije fakt. Musíme tedy doplnit omezující podmínku, že první argument musí být v pravidle větší než `0`.

Zůstává poslední neduh. Pokud zadáme dotaz, Prolog se pokouší po vrácení prvního řešení nalézt další, avšak víme, že žádné další nebude existovat. Proto použijeme řez a upravíme ním fakt pro faktoriál nuly.

Výslední program bude následovný:

```
fact(0, 1) :- !.
fact(N, Fact) :- N > 0, M is N - 1, fact(M, FactP), Fact is N * FactP.
```

Řešení 12.1.3

```
merge([], X, X) :- !.
merge(X, [], X) :- !.
merge([H1|T1], [H2|T2], [H1|T]) :- H1 =< H2, !, merge(T1, [H2|T2], T).
merge([H1|T1], [H2|T2], [H2|T]) :- merge([H1|T1], T2, T).

split([], [], []).
split([X], [X], []).
split([X,Y|Rest], [X|Rest1], [Y|Rest2]) :- split(Rest, Rest1, Rest2).
```

```
mergesort([], []) :- !.
mergesort([X], [X]) :- !.
mergesort(List, Sorted) :-
    split(List, SubList1, SubList2),
    mergesort(SubList1, Sub1Sorted),
    mergesort(SubList2, Sub2Sorted),
    merge(Sub1Sorted, Sub2Sorted, Sorted).
```

Řešení 12.1.4

```
split(_Pivot, [], [], []) :- !.
split(Pivot, [Head|List], [Head|Small], Big) :-
    Head < Pivot,
    !,
    split(Pivot, List, Small, Big).
split(Pivot, [Head|List], Small, [Head|Big]) :-
    split(Pivot, List, Small, Big).

quicksort([], []).
quicksort([X|XS], Sorted) :-
    split(X, XS, Small, Big),
    quicksort(Small, SmallSorted),
    quicksort(Big, BigSorted),
    append(SmallSorted, [X|BigSorted], Sorted).
```

Řešení 12.1.5 Řešení zadaných úloh najdete v souboru *einsteinSol.pl*. Řešení využívající jiné kódování najdete v souboru `12\_einsteinSol2.pl`. Soubory naleznete v ISu a v příloze sbírky.

Řešení 12.1.6

```
equals([], []) :- !.
equals([X|T1], [X|T2]) :- !, equals(T1, T2).
equals([], _S) :- write('1st list is shorter'), !, fail.
equals(_S, []) :- write('2nd list is shorter'), !, fail.
equals([X|_T1], [Y|_T2]) :- write(X), write(' does not equal '),
    write(Y), !, fail.
```

Pro úplnost dodejme, že jestli není vyžadován důvod nerovnosti, vystačili bychom si s pouhým faktem `equals(S, S).` (avšak pouze v případě, že `S` neobsahuje proměnné).

Řešení 12.1.7 Úloha bez rozšíření je v podstatě ekvivalentní úloze o rodinných vztazích z dřívějších cvičení.

```
trip(A, A) :- !.
trip(A, B) :- road(A, X), trip(X, B).
```

Následující řešení zahrnuje první rozšíření:

```
trip(A, A, [A]).
trip(A, B, [A|S]) :- road(A, X), trip(X, B, S).
```

Prezentované řešení pro druhé rozšíření využívá tzv. dynamické klauzule. Ty dovolují měnit programovou databázi za běhu přidáváním (`assert/1`) nebo odebíráním (`retract/1`) nových klauzulí. Alternativním řešením by bylo přidat další argument – seznam navštívených míst.

```
:- dynamic visited/1.
trip(A, A, [A]).
trip(A, B, [A|S]) :- assert(visited(A)), road(A, X),
    \+ visited(X), trip(X, B, S).
trip(A, _B, _S) :- retract(visited(A)), fail.
```

Řešení 12.2.1 Řešení definuje predikát `puzzle/1`, který má na vstupu term představující náš algebrogram. Výpočet spustíte třeba dotazem `?- puzzle(X).`

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    M #\= 0, S #\= 0,
    all_different(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    label(Vars).
```

Řešení 12.2.2 Řešení obsahuje pomocný predikát `printAll/1`, který vypíše všechny prvky zadaného seznamu a pak zalomí řádek. Samotný součet využívá proměnné `Donald`, `Gerald` a `Robert`, které jsou definovány pomocí predikátu `scalar_product/4`.

```
puzzle2([D,O,N,A,L,G,E,R,B,T]) :-
    [O,N,A,L,E,B,T] ins 0..9,
    [D,G,R] ins 1..9,
    [Donald, Gerald, Robert] ins 0..1000000,
    all_distinct([D,O,N,A,L,G,E,R,B,T]),
    scalar_product([100000,10000,1000,100,10,1], [D,O,N,A,L,D],
                   #=, Donald),
    scalar_product([100000,10000,1000,100,10,1], [G,E,R,A,L,D],
                   #=, Gerald),
    scalar_product([100000,10000,1000,100,10,1], [R,O,B,E,R,T],
                   #=, Robert),
    Donald + Gerald #= Robert,
    label([D,O,N,A,L,G,E,R,B,T]),
    printAll([D,O,N,A,L,D]),
    print(+), nl,
    printAll([G,E,R,A,L,D]),
    print(=), nl,
    printAll([R,O,B,E,R,T]).

printAll([]) :- nl.
printAll([H|T]) :- print(H), printAll(T).
```

Řešení 12.2.3

```
puzzle3([K,C,I,O,A,M,L]) :-
    [K,I,O,A,L] ins 1..9,
    [C,M] ins 0..9,
    all_distinct([K,C,I,O,A,M,L]),
    10*K + C + I #= 10*O + K,
    A + A #= 10*K + M,
    10*O + L + 10*K + O #= 10*L + I,
    10*K + C + A #= 10*O + L,
    I + A #= 10*K + O,
    10*O + K + 10*K + M #= 10*L + I,
    label([K,C,I,O,A,M,L]).
```

Řešení 12.2.4

```
fact(0, 1).
fact(N, F) :- N #> 0, N1 #= N - 1, F #= N * F1, fact(N1, F1).
```

Výhodou této formulace je, že je intuitivní a zároveň silnější než běžná definice, protože můžeme pokládat dokonce i dotazy jako `?- fact(N, 120).` nebo dokonce `?- fact(X, Y).` Také pořadí podcílů je volnější.

Řešení 12.2.5 Seznam `Value` představuje hodnoty mincí, které máme k dispozici. Seznam `Count` představuje množství mincí jednotlivých hodnot, které potřebujeme na získání sumy `Sum`. Řešení můžeme odzkoušet třeba na dotazu `?- coins([1,2,5,10,20,50], 174, X).`

```
coins(Value, Sum, Count) :-
    length(Value, Len),
    length(Count, Len),
    Count ins 0..Sum,
    scalar_product(Value, Count, #=, Sum),
    label(Count).
```

Řešení lze vylepšit tak, abychom nejdříve získali řešení s nejmenším počtem mincí. Na to si zavedeme pomocnou proměnnou `Coins` s celkovým počtem mincí a pomocí predikátu `labeling/2` řekneme, že jí chceme minimalizovat:

```
coins(Value, Sum, Count) :-
    length(Value, Len),
    length(Count, Len),
    Count ins 0..Sum,
    scalar_product(Value, Count, #=, Sum),
    sum(Count, #=, Coins),
    labeling([min(Coins)], Count).
```

Řešení 12.2.6

```
:- use_module(library(clpfd)).
```

```
queens(N, L, Type) :-
   length(L, N),
   L ins 1..N,
   constr_all(L),
   labeling(Type, L).

constr_all([]).
constr_all([X|Xs]) :- constr_between(X, Xs, 1), constr_all(Xs).

constr_between(_, [], _).
constr_between(X, [Y|Ys], N) :-
   no_threat(X, Y, N),
   N1 is N + 1,
   constr_between(X, Ys, N1).

no_threat(X, Y, I):- X #\= Y, X + I #\= Y, X - I #\= Y.
```

Řešení 12.2.7

```
sudoku4([A1,A2,A3,A4,
         B1,B2,B3,B4,
         C1,C2,C3,C4,
         D1,D2,D3,D4]) :-
    Values = [A1,A2,A3,A4,B1,B2,B3,B4,C1,C2,C3,C4,D1,D2,D3,D4],
    Values ins 1..4,
    all_distinct([A1,A2,A3,A4]),
    all_distinct([B1,B2,B3,B4]),
    all_distinct([C1,C2,C3,C4]),
    all_distinct([D1,D2,D3,D4]),
    all_distinct([A1,B1,C1,D1]),
    all_distinct([A2,B2,C2,D2]),
    all_distinct([A3,B3,C3,D3]),
    all_distinct([A4,B4,C4,D4]),
    all_distinct([A1,A2,B1,B2]),
    all_distinct([A3,A4,B3,B4]),
    all_distinct([C1,C2,D1,D2]),
    all_distinct([C3,C4,D3,D4]),
    label(Values),
    printSudoku(Values).

printSudoku([]) :- print(+-+-+-+-+), nl.
printSudoku([V1,V2,V3,V4|Rest]) :-
    print('+-+-+-+-+'), nl,
    print('|'), print(V1), print('|'), print(V2), print('|'),
    print(V3), print('|'), print(V4), print('|'), nl,
    printSudoku(Rest).
```