Cvičení 11

11.1 Řezy

Příklad 11.1.1 Navrhněte predikát `max/3`, který uspěje, jestliže je číslo ve třetím argumentu maximem čísel z prvních dvou argumentů. Uveďte řešení bez použití řezu i s ním.

Příklad 11.1.2 Jaký je rozdíl mezi následujícími definicemi predikátů `member/2`? Ve kterých odpovědích se budou lišit?

a)
```
mem1(H, [H|_]).
mem1(H, [_|T]) :- mem1(H, T).
```

b)
```
mem2(H, [H|_]) :- !.
mem2(H, [_|T]) :- mem2(H, T).
```

c)
```
mem3(H, [K|_]) :- H == K.
mem3(H, [K|T]) :- H \== K, mem3(H, T).
```

Příklad 11.1.3 Zakreslete výpočetní SLD stromy následujícího programu na dotaz `?- b(X, Y).` Zakreslete také, kde se nacházejí větve, kterými výpočet nepokračuje, a popište, o které použití řezu se jedná (ořezání/upnutí).

a)
```
a(X) :- X = 0.
a(X) :- X = 1, !.
a(X) :- X = 2.
b(X, Y) :- a(X), a(Y).
```

b)
```
a(X) :- X = 0.
a(X) :- X = 1.
a(X) :- X = 2.
b(X, Y) :- a(X), !, a(Y).
b(X, Y) :- a(X), a(Y).
```

Příklad 11.1.4 Napište predikát `insert/3`, který do uspořádaného seznamu čísel vloží další číslo tak, aby i výslední seznam byl uspořádán. Například dotaz `?- insert(7, [1,2,3,10,12], X).` uspěje se substitucí `X = [1,2,3,7,10,12]`.

Příklad 11.1.5 Napište predikát `remove/3`, který odstraní všechny výskyty prvního argumentu ze seznamu ve druhém argumentu a výsledný seznam unifikuje do třetího argumentu.

Příklad 11.1.6 Napište predikát `intersection/3` pro výpočet průniku dvou seznamů a predikát `difference/3` pro výpočet rozdílu dvou seznamů. Můžete předpokládat, že žádný seznam neobsahuje stejný prvek vícekrát.

Příklad 11.1.7 Níže uvedený logický program obsahuje predikát `fib/2`, který počítá n-tý člen Fibonacciho posloupnosti ($n \geq 1$). Program však není plně korektní: správnou hodnotu sice

vypočítá, ale když si pak pomocí `;` vyžádáme další řešení, program se zacyklí (měl by skončit s odpovědí `no`). Přidejte do programu na správná místa operátory řezu tak, aby v tomto případu necyklil.

```
fib(N, 1) :- N =< 2.
fib(N, X) :- N1 is N - 2, fib(N1, 1, 1, X).
fib(0, _A, X, X).
fib(N, A, B, X) :- N1 is N - 1, C is A + B, fib(N1, B, C, X).
```

Příklad 11.1.8 Do programu z úlohy **??** přidejte jeden řez tak, aby výsledný program uspěl se stejným dotazem (`?- p(X, X).`) právě dvakrát.

Příklad 11.1.9 Uvažte následující program:

```
f(X) :- g(X), h(X).
f(3).
g(0).
g(1).
g(2).
h(1).
h(2).
```

Vyhodnoťte dotaz `?- f(X).` pro výše uvedený program a pro jeho 4 variace, které vzniknou výměnou příslušného pravidla za verzi s řezem uvedenou níže. Pro každý nakreslete kompletní SLD strom výpočtu se všemi řešeními.

a) `f(X) :- ! , g(X), h(X).`
b) `f(X) :- g(X), h(X), ! .`
c) `f(X) :- g(X), ! , h(X).`
d) `g(1) :- ! .`

11.2 Negace

Příklad 11.2.1 Definujte predikát `nd35/1`, který uspěje, pokud jako parametr dostane číslo, které není dělitelné číslem 3 a současně není delitelné číslem 5.

Úlohu vyřešte bez použití řezu a negace, s použitím řezu a s použitím negace.

Příklad 11.2.2 Pomocí negace a predikátu `member/2` naprogramujte predikáty `inNeither/3` a `inExactlyOne/3` takové, že

- `inNeither(X, XS, YS)` uspěje právě tehdy, když prvek `X` není v seznamu `XS` ani v seznamu `YS`;
- `inExactlyOne(X, XS, YS)` uspěje právě tehdy, když prvek `X` je v právě jednom ze seznamů `XS` a `YS`. Predikát `inExactlyOne` zkuste naprogramovat tak, aby uměl do volné proměnné `X` postupně unifikovat všechny takové prvky.

11.3 Predikáty pro všechna řešení

Příklad 11.3.1 S využitím predikátů pro všechna řešení a knihovních funkcí pro seznamy napište následující predikáty:

a) `variation3all(+List, ?Variations)`, který vygeneruje seznam všech tříprvkových variací prvků ze seznamu `List`.
b) `combination3all(+List, ?Combinations)`, který vygeneruje seznam všech tříprvkových kombinací prvků ze seznamu `List`.

Poznámka: Podívejte se i na úlohu **??**.

Příklad 11.3.2 Uvažme následující databázi faktů:

```
f(a, b).
f(a, c).
f(a, d).
f(e, c).
f(g, h).
f(g, b).
f(i, a).
```

Bez použití interpretru zjistěte, s jakou substitucí uspějí následující dotazy:

a) `?- findall(X, f(a, X), List).`
b) `?- findall(X, f(X, b), List).`
c) `?- findall(X, f(X, Y), List).`
d) `?- bagof(X, f(X, Y), List).`
e) `?- setof(X, Y ^ f(X, Y), List).`

Příklad 11.3.3 Napište predikát `subsets/2`, který pro danou množinu vygeneruje všechny její podmnožiny. Množinou rozumíme seznam, ve kterém se neopakují prvky. Na pořadí vygenerovaných množin nezáleží. Napište i predikát `isset/1`, který uspěje, když zadaný seznam korektně reprezentuje množinu (tedy neobsahuje duplicitní prvky).

11.4 Základy I/O

Příklad 11.4.1 Napište predikát `fileSum(+FileName, -Sum)`, který bude číst zadaný soubor po termech a spočítá sumu čísel v termech tvaru `s(X)`. Ostatní termy bude ignorovat.

Příklad 11.4.2 Napište predikát `contains(+FileName, +Char)`, který uspěje, pokud se v souboru se jménem `FileName` vyskytuje znak `Char`.

Příklad 11.4.3 S použitím predikátů pro získání všech řešení a predikátu `contains/2` z předchozího příkladu získejte všechny znaky uvedené v daném souboru.

Řešení

Řešení 11.1.1

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.

% efektivnejsi reseni pomoci rezu
max2(X, Y, Z) :- X >= Y, !, Z = X.
max2(X, Y, Y).
```

Následující definice není korektní:

```
max(X, Y, X) :- X >= Y, !.
max(_X, Y, Y).
```

Dotaz `?- max(2, 1, 1).` uspěje. Nedá se totiž unifikovat s hlavou první klauzule, tudíž se interpretr ani nepokusí o vyhodnocení její pravé strany a rovnou přejde na druhou klauzuli. Ta uspěje, protože její platnost není vůbec závislá na hodnotě `_X`.

Pro přehlednost pojmenujme třetí argument predikátu `Z`. Problém je v tom, že první klauzule tvrdí `X = Z` $\land$ `X >= Y` $\Rightarrow$ `true`, zatímco správná definice je `X >= Y` $\Rightarrow$ `Z = X`.

Naopak u `max2` výše tento problém není: nyní již není možné dostat se na druhou klauzuli v případě, že platí `X >= Y`, protože hlava první klauzule je vždy unifikovatelná s dotazem (a pokud `X >= Y` uspěje, řez zajistí, že se již nemůžeme dostat na druhou klauzuli).

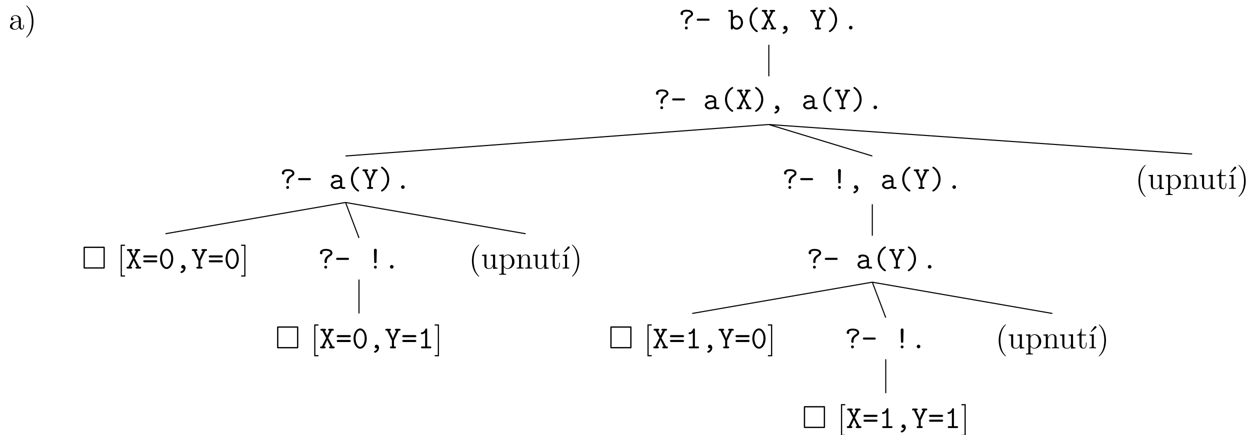
Řešení 11.1.2 Predikát `mem1/2` vyhledá všechny výskyty prvku. Při porovnávání hledaného prvku s prvky seznamu může dojít k vázání proměnných (může sloužit ke generování všech prvků seznamu).

Predikát `mem2/2` najde jenom první výskyt, také váže proměnné.

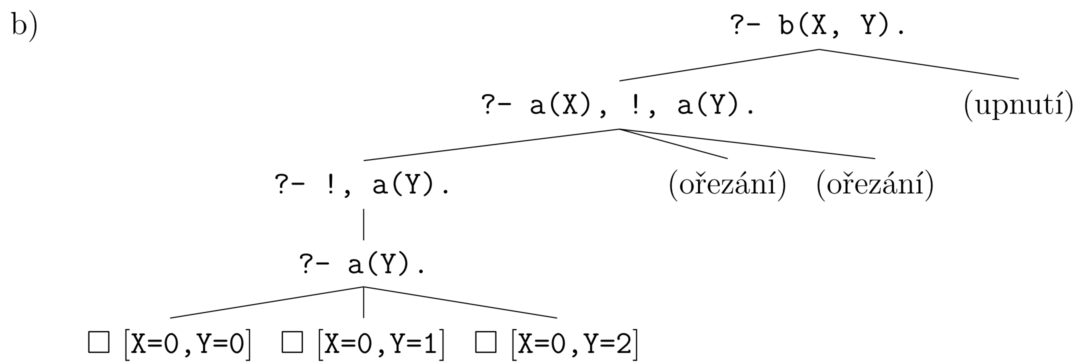
Predikát `mem3/2` najde jenom první výskyt, proměnné neváže (hledá pouze prvky, které jsou identické jako termy).

Řešení 11.1.3 V níže uvedených výpočetních stromech byly pro zpřehlednění vynechány predikáty, které pouze unifikují proměnnou s číselnou hodnotou (a uspějí s triviální substitucí čísla za onu proměnnou).

a)



b)



Řešení 11.1.4

```
insert(X, [], [X]).
insert(X, [H|T], R) :- X =< H, !, R = [X,H|T].
insert(X, [H|T], [H|T1]) :- insert(X, T, T1).
```

Všimněte si, že pokud bychom namísto `R` v druhé klauzuli umístili výraz `[X,H|T]`, máme stejný problém jako u `max/3` v příkladu 11.1.1: například dotaz `?- insert(1, [1,2], [1,2,1]).` uspěje.

Řešení 11.1.5

```
remove(_X, [], []).
remove(X, [X|S], S1) :- !, remove(X, S, S1).
remove(X, [Y|S], [Y|S1]) :- remove(X, S, S1).
```

Zvažte, proč je nutný řez a jak by vypadal výsledek bez něj.

Řešení 11.1.6 V řešení využíváme knihovní funkci `member/2`, která uspěje, když je první argument prvkem seznamu v druhém argumentu.

```
intersection([], _List, []).
intersection([H|T], List, R) :-
    member(H, List), !, R = [H|Rest], intersection(T, List, Rest).
intersection([_H|T], List, Rest) :- intersection(T, List, Rest).

difference([], _List, []).
```

```
difference([H|T], List, Rest) :-
    member(H, List), !, difference(T, List, Rest).
difference([H|T], List, [H|Rest]) :- difference(T, List, Rest).
```

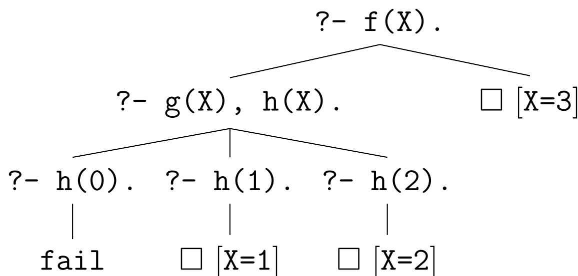
Řešení 11.1.7

```
fib(N, 1) :- N =< 2, !.
fib(N, X) :- N1 is N - 2, fib(N1, 1, 1, X).
fib(0, _A, X, X) :- !.
fib(N, A, B, X) :- N1 is N - 1, C is A + B, fib(N1, B, C, X).
```

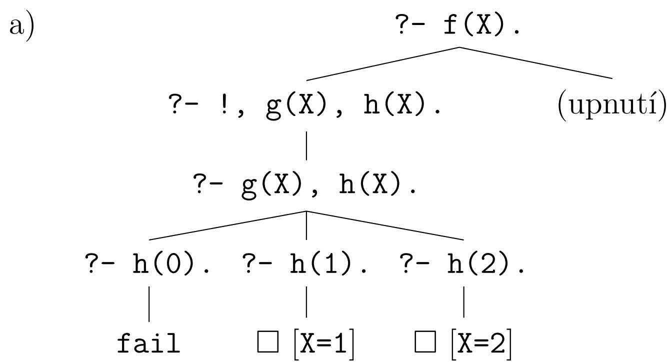
Řešení 11.1.8 Řez doplníme na konec sedmého řádku. Ten bude tedy vypadat následovně:

```
s(X) :- t(X, a), !.
```

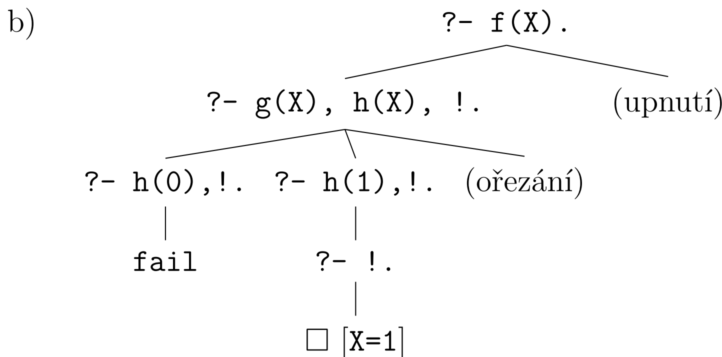
Řešení 11.1.9



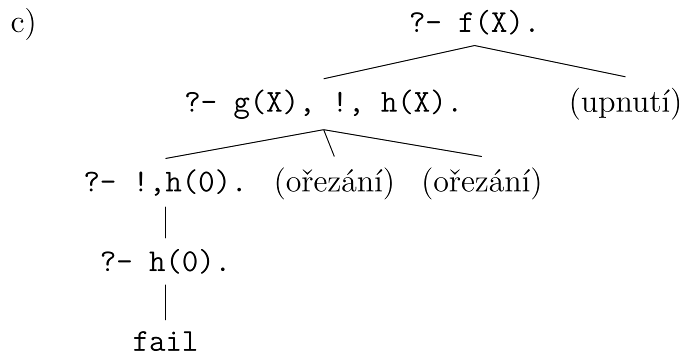
a)



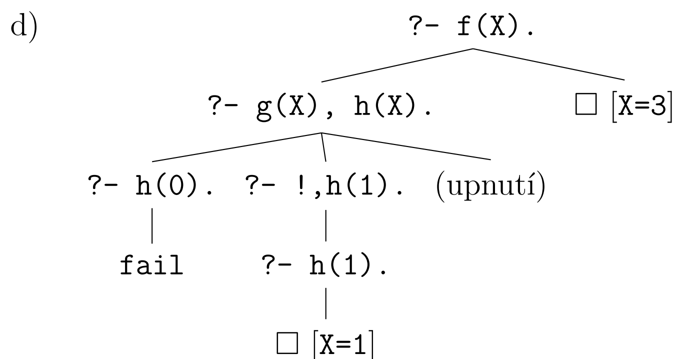
b)



c)



d)



Řešení 11.2.1

```
nd35a(X) :- X mod 5 =\= 0, X mod 3 =\= 0.

nd35b(X) :- X mod 5 =:= 0, !, fail.
nd35b(X) :- X mod 3 =:= 0, !, fail.
nd35b(X).

nd35c(X) :- \+(X mod 3 =:= 0), \+(X mod 5 =:= 0).
```

Řešení 11.3.1 Uvažme predikáty `variation3/2` a `combination3/2` z řešení úlohy **??**. Pak už je definice poměrně jednoduchá:

a)
```
variation3all(List, Y) :-
    findall(Variation3, variation3(List, Variation3), Y).
```

b)
```
combination3all(List, Y) :-
    findall(Combination3, combination3(List, Combination3), Y).
```

Řešení 11.3.2

a) `List = [b,c,d].`
b) `List = [a,g].`
c) `List = [a,a,a,e,g,g,i].`
d) `Y = a, List = [i];`
`Y = b, List = [a,g];`
`Y = c, List = [a,e];`

```
    Y = d, List = [a];
    Y = h, List = [g].
```
e) `List = [a,e,g,i].`

Řešení 11.3.3

```
subsets(X, S) :- isset(X), findall(Y, subset(X, Y), S).

isset([]).
isset([X|T]) :- \+ member(X, T), isset(T).

subset([], []).
subset([_X|T], S) :- subset(T, S).
subset([X|T], [X|S]) :- subset(T, S).
```

Řešení 11.4.1

```
fileSum(FileName, Sum) :-
    seeing(Old), seen, see(FileName),
    read(X),
    fileSumH(X, 0, Sum),
    !,
    seen, see(Old).

fileSumH(end_of_file, Sum, Sum).
fileSumH(s(N), AcSum, Sum) :-
    AcSum1 is AcSum + N,
    read(X),
    fileSumH(X, AcSum1, Sum).
fileSumH(_, AcSum, Sum) :-
    read(X),
    fileSumH(X, AcSum, Sum).
```

Řez v řešení je nutný, aby nedošlo k backtrackování zpět do `fileSumH/3` v případě, že by nějaký pozdější cíl selhal (což by způsobilo čtení z jiného proudu!).

Řešení 11.4.2 Nejdříve si zapamatujeme, který proud byl původně nastaven jako čtecí (abychom ho pak mohli znova nastavit, až skončíme). Pak proud zavřeme a otevřeme požadovaný soubor. Pak opakujeme (predikát `repeat/0`) načtení znaku, test na konec souboru a unifikaci na požadovaný znak. Jestli test na konec souboru uspěl, výpočet zařízneme, obnovíme původní vstupní proud a selžeme (znak se nenašel).

```
contains(FileName, Char) :- seeing(Old), seen, see(FileName),
    repeat, get_char(X),
    (X == end_of_file -> !, restore(Old), fail ; Char = X).

restore(Old) :- seen, see(Old).
```

Řešení 11.4.3 Stačilo by nám využít následujícího dotazu:

```
findall(Char, contains(FileName, Char), List).
```

Uvedené řešení však načítá znaky ze souboru postupně, po jednom. Navíc pro získání každého znaku vyhodnocuje znovu predikát `contains/2`. Je lepší využít efektivní knihovní funkci `read_file_to_codes/3`.