Cvičení 10

10.1 Číselné operace

Příklad 10.1.1 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 = 1 + 1`
b) `1 + 1 = 1 + 1`
c) `2 is 1 + 1`
d) `1 + 1 is 1 + 1`
e) `X = 1 + 1`
f) `X is 1 + 1`
g) `2 is 1 + X`
h) `X = 1, 2 is 1 + X`

Příklad 10.1.2 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `X = 2`
b) `X == 2`
c) `X = 2, X == 2`
d) `X =:= 2`
e) `1 + 1 == 2`
f) `1 + 1 =:= 2`
g) `2 =:= 1 + 1`

Příklad 10.1.3 Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 < 2 + 1`
b) `1 + 2 =< 2 + 1`
c) `1 + 2 >= 1`
d) `2 * 3 > 3 * 1.5`
e) `1 + 1 \== 2`
f) `1 + 1 =\= 2`
g) `1 + 2 =\= 2`

Příklad 10.1.4 Jak se liší následující výrazy? Které by v případě dotazu uspěly, které ne, a které nejsou syntakticky správně? Při korektních unifikacích určete i výslednou substituci.

a) `X = Y + 1`
b) `X is Y + 1`
c) `X = Y`
d) `X == Y`
e) `1 + 1 = 2`
f) `2 = 1 + 1`
g) `1 + 1 = 1 + 1`
h) `2 is 1 + 1`

i) `1 + 1 is 1 + 1`
j) `1 + 2 =:= 2 + 1`
k) `X \== Y`
l) `X =\= Y`
m) `1 + 2 =\= 1 - 2`
n) `1 <= 2`
o) `1 =< 2`
p) `sin(X) is sin(2)`
q) `sin(X) = sin(2 + Y)`
r) `sin(X) =:= sin(2 + Y)`

Příklad 10.1.5 Rozhodněte, které z následujících dotazů uspějí a které ne.

a) `?- 15 is 3 * 5.`
b) `?- 14 =\= 3 * 5.`
c) `?- 15 = 3 * 5.`
d) `?- 15 == 3 * 5.`
e) `?- 15 =\= 3 * 5.`
f) `?- 15 =:= 3 * 5.`
g) `?- 3 * 5 == 3 * 5.`
h) `?- 3 * 5 == 5 * 3.`
i) `?- 3 * 5 =:= 3 * 5.`
j) `?- 10 - 3 =\= 9 - 2.`

Příklad 10.1.6 Napište predikát `convert/2`, který převede svůj první argument (přirozené číslo) na reprezentaci daného čísla pomocí následníků. Například `?- convert(3, X).` uspěje s unifikací `X = s(s(s(0)))`.

Příklad 10.1.7 Napište predikát `firstnums/2`, který spočítá součet prvních `N` přirozených čísel. Například dotaz `?- firstnums(5, X).` uspěje s unifikací `X = 15`.

Příklad 10.1.8 Implementujte predikát `fact/2`, který při dotazu `fact(`m`, `n`)` uspěje, pokud $m > 0$ a $n = m!$. V ostatních případech může interpret cyklit.

Predikát by měl rovněž fungovat při volání ve tvaru `fact(`n`, Res)`, kde `Res` je volná proměnná, a unifikovat tuto proměnnou s $n!$.

Příklad 10.1.9 Napište predikát `powertwo/1`, který uspěje, pouze když číslo zadané jako argument je mocninou dvou. Využijte fakt, že mocniny dvou lze opakovaně beze zbytku dělit dvěma, dokud nedostaneme jedna. Můžete využít binární operátory `mod` pro modulo a `div` pro celočíselné dělení.

Příklad 10.1.10 Naprogramujte predikát `prime/1`, který uspěje, pokud `Num` je prvočíslo. K řešení lze použít naivní metodu testování zbytku po dělení všemi čísly od 2 po zadané číslo `Num`.

Příklad 10.1.11 Napište predikát `gcd/3`, který spočítá největšího společného dělitele dvou přirozených čísel. Například dotaz `?- gcd(21, 49, X).` uspěje s unifikací `X = 7`.

Využijte Euklidův algoritmus, který je založen na pozorování, že největší společný dělitel čísel a, b (kde $a > b$) je stejný jako největší společný dělitel čísel $(a - b)$, b.

Příklad 10.1.12 Napište predikát `dsum/2`, který spočítá ciferný součet zadaného přirozeného čísla. Například dotaz `?- dsum(12345, X).` uspěje s unifikací `X = 15`.

Příklad 10.1.13 Zamyslete se a zdůvodněte, proč následující logický program není vhodný, i když jeho výsledek je vždy korektní. Program počítá n-tý člen Fibonacciho posloupnosti přímo podle definice.

```
fib(0, 0).
fib(1, 1).
fib(N, X) :-
    N > 2, N1 is N - 1, N2 is N - 2, fib(N1, Y), fib(N2, Z), X is Y + Z.
```

10.2 Práce se seznamy

Příklad 10.2.1 Které z následujících zápisů představují korektní zápis seznamu? Pokud je zápis korektní, určete počet prvků v daném seznamu.

a) `[1|[2,3,4]]`
b) `[1,2,3|[]]`
c) `[1|2,3,4]`
d) `[1|[2|[3|[4]]]]`
e) `[[]|[]]`
f) `[[1,2]|[4]]`
g) `[[1,2],[3,4]|[5,6,7]]`

Příklad 10.2.2 Implementujte vlastní verze predikátů, které pro seznamy počítají standardní seznamové funkce, které znáte z Haskellu. Konkrétně imitujte funkce `head`, `tail`, `last` a `init`.

Příklad 10.2.3 Napište následující predikáty pro práci se seznamy za pomoci vestavěného predikátu `append/3`.

a) `prefix/2` uspěje, jestliže je první argument prefixem seznamu ve druhém argumentu.
b) `suffix/2` uspěje, jestliže je první argument sufixem seznamu ve druhém argumentu.
c) `element/2` uspěje, jestliže je první argument členem seznamu ve druhém argumentu.
d) `adjacent/3` uspěje, jestliže jsou první dva prvky členy seznamu ve třetím argumentu a jsou vedle sebe (v daném pořadí).
e) `sublist/2` uspěje, jestliže je seznam v prvním argumentu podseznamem seznamu v druhém argumentu.

Příklad 10.2.4 Předpokládejme existenci predikátu `mf/2`, použitelného v módu (+,-). Vytvořte predikát `map/2`, který bude imitovat chování Haskellové funkce `map`, tj. bude možné pomocí něj „aplikovat funkci `mf`“ na každý prvek zadaného seznamu.

Příklad 10.2.5 Určete, co počítá (jaký význam má) následující program:

```
something([], []).
something([H|T], [H|S]) :- delete(T, H, X), something(X, S).
```

Příklad 10.2.6 S využitím knihovních funkcí pro seznamy napište následující predikáty tak, aby je bylo možno používat v módu (+,?). Ve všech případech můžete předpokládat, že vstupní seznam neobsahuje duplicitní prvky.

a) `variation3/2`, který uspěje, pokud je druhý argument tříprvkovou variací bez opakování ze seznamu v prvním argumentu.
b) `combination3/2`, který uspěje, pokud je druhý argument tříprvkovou kombinací bez opakování ze seznamu v prvním argumentu. Trojice prvků v kombinaci ať jsou uspořádány (lze k tomu využít binární operátor `@<`, který umožňuje porovnávat termy[1]).

Příklad 10.2.7 Napište predikát `doubles/2`, který uspěje, jestli prvky ve druhém seznamu jsou dvojnásobky prvků v prvním seznamu.

Například dotaz `?- doubles([5,3,2], [10,6,4]).` uspěje, zatímco dotaz `?- doubles([1,2,3], [2,4,5]).` neuspěje.

Příklad 10.2.8 Napište následující predikáty:

a) `listLength/2`, který vypočítá délku zadaného seznamu.
b) `listSum/2`, který vypočítá součet čísel v zadaném seznamu.
c) `fact/2`, který vypočítá faktoriál zadaného čísla.
d) `scalar/3`, který vypočítá skalární součin zadaných dvou seznamů (můžete předpokládat, že jsou stejné délky).

Zkuste tyto predikáty přepsat za pomocí akumulátoru, aby se při výpočtu mohla použít optimalizace posledního volání.

Příklad 10.2.9 Napište predikát `filter/2`, který ze seznamu odstraní všechny nečíselné hodnoty. Například dotaz `?- filter([5,s(0),3,a,2,A,7], X).` uspěje se substitucí `X = [5,3,2,7]`.

Příklad 10.2.10 Napište predikát `digits/2`, který ze seznamu cifer vytvoří číslo. Například dotaz `?- digits([2,8,0,7], X).` uspěje se substitucí `X = 2807`.

Příklad 10.2.11 Napište predikát `nth/3`, který vrátí n-tý prvek seznamu. Například dotaz `?- nth(4, [5,2,7,8,0], X).` uspěje se substitucí `X = 8`.

Příklad 10.2.12 Definujte následující predikáty s pomocí akumulátoru:

a) `listSum/2`, který sečte seznam čísel a na nečíselném seznamu neuspěje (použijte `number/1`),
b) `fact/2`, který spočítá faktoriál zadaného čísla,
c) `fib/2`, který (efektivně) spočítá n-tý člen Fibonacciho posloupnosti,
d) `reverseAcc/2`, který obrátí pořadí prvků v seznamu.

[1] http://www.swi-prolog.org/pldoc/doc_for?object=section%282,%274.7%27,swi%28%27/doc/Manual/compare.html%27%29%29

Příklad 10.2.13 Napište predikát `mean/2`, který spočítá aritmetický průměr zadaného seznamu. Například dotaz `?- mean([1,2,3,4,5,6], X).` uspěje se substitucí `X = 3.5`.

Příklad 10.2.14 Napište predikát `zip/3`, který ze dvou seznamů vytvoří nový spojením příslušných prvků do dvojic. Například dotaz `zip([1,2,3], [4,5,6], S).` uspěje se substitucí `S = [1-4,2-5,3-6]`. V případě rozdílných délek seznamů se přebytečné prvky ignorují.

Řešení

Řešení 10.1.1

a) Unifikace neuspěje, term `2` není unifikovatelný s termem `+(1, 1)`.
b) Unifikace uspěje.
c) Uspěje, aritmeticky se vyhodnotí pravý argument operátoru `is`, a poté unifikuje s levou stranou (tedy unifikujeme `2 = 2`).
d) Neuspěje, pravá strana se vyhodnotí na `2`, levá strana je ale term `1 + 1`, který není unifikovatelný s `2`.
e) Uspěje s přiřazením `X = 1 + 1`, unifikace nevyhodnocuje aritmetiku.
f) Uspěje s přiřazením `X = 2` díky aritmetickému vyhodnocení přes `is`.
g) Neuspěje, protože pravý argument `is` obsahuje volnou (nepřiřazenou) proměnnou.
h) Uspěje, do `X` je přiřazeno díky předchozí unifikaci.

Řešení 10.1.2

a) Uspěje, dojde k substituci.
b) Neuspěje, `X` není identické s `2`.
c) Uspěje, v okamžiku testování identity je `X` již nastaveno na `2`.
d) Neuspěje, `=:=` aritmeticky vyhodnocuje obě strany, a proto ani jedna nesmí obsahovat neinstanciovanou proměnnou.
e) Neuspěje, termy nejsou identické.
f) Uspěje, protože se nejprve aritmeticky vyhodnotí obě strany na `2`.
g) Uspěje, protože se nejprve aritmeticky vyhodnotí obě strany na `2`.

Řešení 10.1.3

a) Uspěje, obě strany se nejprve aritmeticky vyhodnotí a nerovnost platí.
b) Uspěje.
c) Uspěje.
d) Uspěje.
e) Uspěje, termy nejsou identické.
f) Neuspěje, obě strany se nejprve aritmeticky vyhodnotí, výsledné termy jsou ale identické.
g) Uspěje, obě strany se nejprve aritmeticky vyhodnotí a výsledné termy nejsou identické.

Řešení 10.1.4

a) Unifikace, uspěje se substitucí [`X = Y + 1`].
b) Nekorektní výraz, pravá strana operátoru `is` obsahuje proměnnou.
c) Unifikace, uspěje se substitucí [`X = Y`].
d) Porovnání na identitu, neuspěje (proměnné jsou různé).
e) Unifikace, neuspěje (různé termy).
f) Unifikace, neuspěje (různé termy).
g) Unifikace, uspěje s prázdnou substitucí.
h) Aritmetické vyhodnocení + unifikace, uspěje (vhodnější by však pravděpodobně bylo použít operátor `=:=`).

i) Aritmetické vyhodnocení + unifikace, unifikace neuspěje (ekvivalentní `1 + 1 = 2`, kde se liší term na nejvyšší úrovni – `+/2` vs. `2/0`).
j) Aritmetické porovnání, uspěje.
k) Porovnání na (identickou) různost, uspěje.
l) Nekorektní aritmetické porovnání, výrazy na obou stranách nejsou instanciovány.
m) Aritmetická nerovnost, uspěje.
n) Nekorektní výraz, operátor `<=` neexistuje, správně je `=<`.
o) Aritmetické porovnání, uspěje.
p) V daném kontextu funguje podobně jako aritmetické porovnání, tedy neuspěje (`sin(2)` se vyhodnotí, ale `sin(X)` zůstává jako aplikace termu na proměnnou).
q) Unifikace, uspěje se substitucí [`X = 2 + Y`].
r) Nekorektní aritmetické porovnání, výrazy na obou stranách nejsou plně instanciovány.

Řešení 10.1.5

a) Uspěje (na porovnání na aritmetickou rovnost by se však měl použít operátor `=:=`).
b) Uspěje.
c) Unifikace neuspěje, termy jsou různé.
d) Neuspěje, termy nejsou identické.
e) Neuspěje, aritmetická nerovnost není splněna.
f) Uspěje, aritmetická rovnost platí.
g) Uspěje, termy jsou identické.
h) Neuspěje, termy nejsou identické.
i) Uspěje, aritmetická rovnost je splněna.
j) Neuspěje, aritmetická nerovnost není splněna.

Řešení 10.1.6

```
convert(0, 0).
convert(X, s(Y)) :- X1 is X - 1, convert(X1, Y).
```

Řešení 10.1.7

Jednodušší řešení:

```
firstnums(0, 0).
firstnums(N, S) :- N1 is N - 1, firstnums(N1, S1), S is S1 + N.
```

Řešení využívající akumulátor (a optimalizaci posledního volání):

```
firstnums(N, S) :- firstnums(N, 0, S).
firstnums(0, S, S).
firstnums(N, X, S) :- N1 is N - 1, X1 is X + N, firstnums(N1, X1, S).
```

Toto řešení má nevýhodu v tom, že po prvně nalezeném řešení se Prolog pokusí nalézt další (v místě, kde bylo použito faktu, se pokusí použít pravidla), avšak žádné další nenalezne a výpočet neskončí. To lze napravit pomocí tzv. *řezů*, které budou probírány později. Fakt by se upravil na `firstnums(0, S, S) :- !.` Řez (`!`) zakáže hledání dalšího řešení, pokud uspěje tento (původně) fakt.

Připomeňme také, že predikát v Prologu je definován nejen identifikátorem, ale také aritou, takže dvou- a tříargumentové `firstnums` jsou rozdílné a nezávislé predikáty.

Výše uvedené řešení sice funguje, avšak níže uvedené je efektivnější (využívá vzorec pro součet aritmetické posloupnosti)

```
firstnums2(N, S) :- S is (N * (N + 1)) // 2.
```

Použitý operátor `//` realizuje celočíselné dělení.

Řešení 10.1.8

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

Řešení 10.1.9

```
powertwo(1).
powertwo(X) :- X mod 2 =:= 0, X1 is X div 2, powertwo(X1).
```

Řešení 10.1.10

```
prime(Number) :- testPrime(2, Number).

testPrime(P, P) :- !.
testPrime(D, P) :- D < P, P mod D =\= 0, D1 is D + 1, testPrime(D1, P).
```

Opět, v řešení používáme řez, abychom zamezili hledání dalšího řešení, které by neexistovalo.

Řešení 10.1.11

```
gcd(X, X, X).
gcd(A, B, X) :- A > B, A1 is A - B, gcd(A1, B, X).
gcd(A, B, X) :- B > A, B1 is B - A, gcd(A, B1, X).
```

Dodejme, že existuje i efektivnější verze využívající zbytky po dělení.

Řešení 10.1.12

```
dsum(N, Sum) :- dsum(N, 0, Sum).
dsum(0, Sum, R) :- !, R = Sum.
dsum(N, X, Sum) :- X1 is X + N mod 10, N1 is N // 10, dsum(N1, X1, Sum).
```

Opět, v řešení používáme řez, abychom zamezili hledání dalšího řešení, které by neexistovalo.

Řešení 10.1.13 Program je velice neefektivní (má exponenciální složitost). Hodnoty předchozích členů posloupnosti se počítají opakovaně. Nejlépe to uvidíte, když si nakreslíte strom výpočtu pro nějaké malé `N`, například `6`.

Takto zapsaný program kromě toho neumožňuje optimalizaci koncového volání (koncovou rekurzi nebo tail recursion).

Řešení 10.2.1

a) Korektní seznam s délkou 4.

b) Korektní seznam s délkou 3.
c) Nekorektní výraz (zbytek seznamu za znakem `|` není zapsaný správně).
d) Korektní seznam s délkou 4.
e) Korektní seznam s délkou 1.
f) Korektní seznam s délkou 2 (není homogenní!).
g) Korektní seznam s délkou 5 (není homogenní!).

Řešení 10.2.2

```
head([H|_], H).

tail([_|T], T).

last([Last], Last).
last([_|T], Last) :- last(T, Last).

init([_], []).
init([H|Tail], [H|Init]) :- init(Tail, Init).
```

Řešení 10.2.3

a)
```
prefix(Prefix, List) :- append(Prefix, _, List).
```
b)
```
suffix(Suffix, List) :- append(_, Suffix, List).
```
c)
```
element(X, List) :- append(_Prefix, [X|_Suffix], List).
```
d)
```
adjacent(X, Y, List) :- append(_Prefix, [X,Y|_Suffix], List).
```
e)
```
sublist(Sub, List) :- append(_Prefix, SubSuffix, List),
    append(Sub, _Suffix, SubSuffix).
```

Řešení 10.2.4

```
map([], []).
map([H1|T1], [H2|T2]) :- mf(H1, H2), map(T1, T2).
```

Řešení 10.2.5 Predikát `something/2` odstraní ze seznamu v prvním argumentu duplicitní prvky (ponechá jenom první výskyt) a výsledný seznam unifikuje do druhého argumentu.

Řešení 10.2.6 V řešení využíváme knihovní funkci `member/2`, která uspěje, pokud je první argument prvkem seznamu v druhém argumentu.

a)

```
variation3(List, Variation3) :-
    member(A, List), member(B, List), A \= B,
    member(C, List), A \= C, B \= C, Variation3 = [A,B,C].
```

b)

```
combination3(List, Combination3) :-
    variation3(List, [X,Y,Z]), X @< Y, Y @< Z,
    Combination3 = [X,Y,Z].
```

Řešení 10.2.7

```
doubles([], []).
doubles([X1|T1], [X2|T2]) :- X2 is 2 * X1, doubles(T1, T2).
```

Řešení 10.2.8

a)

```
listLength([], 0).
listLength([_|T], Length) :-
    listLength(T, LengthTail), Length is LengthTail + 1.
```

b)

```
listSum([], 0).
listSum([H|T], Sum) :- listSum(T, SumTail), Sum is SumTail + H.
```

c)

```
fact(0, 1) :- !.
fact(N, Fact) :- NN is N - 1, fact(NN, SubFact), Fact is SubFact * N.
```

d)

```
scalar([], [], 0).
scalar([H1|T1], [H2|T2], Scalar) :-
    scalar(T1, T2, ScalarTail), Scalar is ScalarTail + H1 * H2.
```

Všechny predikáty se však dají naprogramovat i efektivněji: tak, aby se při výpočtu mohla použít optimalizace posledního volání. Myšlenkou je použít pomocnou proměnnou (akumulátor), ve kterém si budeme „střádat“ mezivýsledek. Následují všechna řešení přepsaná tímto způsobem. Pro pomocný predikát využívající akumulátor používáme stejné jméno, protože se liší v aritě.

a)

```
listLength2(List, Length) :- listLength2(List, Length, 0).
listLength2([], Length, Length).
listLength2([_|T], Length, Acc) :- Acc2 is Acc + 1,
                                   listLength2(T, Length, Acc2).
```

b)

```
listSum2(List, Sum) :- listSum2(List, Sum, 0).
listSum2([], Sum, Sum).
listSum2([H|T], Sum, Acc) :- Acc2 is Acc + H, listSum2(T, Sum, Acc2).
```

c)

```
fact2(N, Fact) :- fact2(N, Fact, 1).
fact2(0, Fact, Fact) :- !.
fact2(N, Fact, Acc) :- Acc2 is Acc * N, NN is N - 1,
    fact2(NN, Fact, Acc2).
```

d)

```
scalar2(List1, List2, Scalar) :- scalar2(List1, List2, Scalar, 0).
scalar2([], [], Scalar, Scalar).
scalar2([H1|T1], [H2|T2], Scalar, Acc) :- Acc2 is Acc + H1 * H2,
    scalar2(T1, T2, Scalar, Acc2).
```

Řešení 10.2.9

```
filter([], []).
filter([X|T1], [X|T2]) :- number(X), !, filter(T1, T2).
filter([_X|T1], T2) :- filter(T1, T2).
```

Řešení 10.2.10

```
digits(S, X) :- digits(S, 0, X).
digits([], X, X).
digits([A|T], P, X) :- P1 is 10 * P + A, digits(T, P1, X).
```

Řešení 10.2.11

```
nth(1, [X|_], X).
nth(N, [_|XS], X) :- N > 1, N1 is N - 1, nth(N1, XS, X).
```

Řešení 10.2.12

```
listSum(List, Sum) :- listSum2(List, 0, Sum).
listSum2([], Sum, Sum).
listSum2([X|XS], Acc, Sum) :-
    number(X), Acc1 is Acc + X, listSum2(XS, Acc1, Sum).

fact(N, F) :- fact2(N, 1, F).
fact2(0, F, F).
fact2(N, Acc, F) :-
    N > 0, Acc1 is N * Acc, N1 is N - 1, fact2(N1, Acc1, F).

fib(N, Fib) :- fib2(N, 0, 1, Fib).
fib2(0, Fib, _, Fib).
fib2(N, A, B, Fib) :-
    N > 0, B1 is A + B, N1 is N - 1, fib2(N1, B, B1, Fib).

reverseAcc(List, Reverse) :- reverseAcc2(List, [], Reverse).
reverseAcc2([], List, List).
reverseAcc2([X | XS], List, R) :- reverseAcc2( XS, [X | List], R).
```

Řešení 10.2.13

```
mean(S, X) :- mean(S, 0, 0, X).
mean([], Sum, Count, X) :- X is Sum / Count.
mean([H|T], Sum, Count, X) :-
    Sum1 is Sum + H, Count1 is Count + 1, mean(T, Sum1, Count1, X).
```

Řešení 10.2.14

```
zip([H1|T1], [H2|T2], [H1-H2|T]) :- zip(T1, T2, T).
zip([], [_|_], []).
zip(_, [], []).
```