Operátory, aritmetika

Operátory

- Infixová notace: `2*a + b*c`
- Prefixová notace: `+( *(2,a), *(b,c) )` priorita +: 500, priorita *: 400
  - prefixovou notaci lze získat predikátem display/1

    `:- display((a:-s(0),b,c)).` `:-(a, ,(s(0), ,(b,c)))`
- **Priorita operátorů**: operátor s **nejvyšší** prioritou je hlavní funktor
- Uživatelsky definované operátory: `zna`

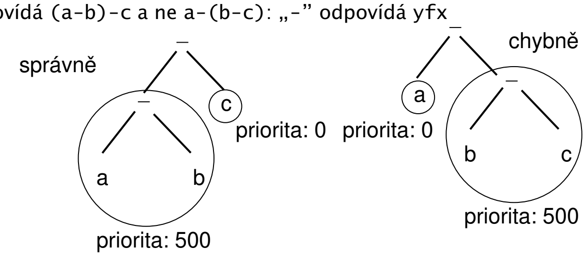
  `petr zna alese.` `zna( petr, alese).`
- Definice operátoru: `:- op( 600, xfx, zna ).` priorita: 1..1200
  - `:- op( 1100, xfy, ; ).` nestrukturované objekty: 0

    `:- op( 1000, xfy, , ).`

    `p :- q,r; s,t.` `p :- (q,r) ; (s,t).` ; má vyšší prioritu než ,
  - `:- op( 1200, xfx, :- ).` `:-` má nejvyšší prioritu
- Definice operátoru není spojena s datovými manipulacemi (kromě spec. případů)

Typy operátorů

- Typy operátorů
  - infixové operátory: `xfx, xfy, yfx` př. xfx = yfx -
  - prefixové operátory: `fx, fy` př. fx ?- fy -
  - postfixové operátory: `xf, yf`
- x a y určují **prioritu argumentu**
  - x reprezentuje argument, jehož priorita musí být **striktně menší** než u operátoru
  - y reprezentuje argument, jehož priorita je **menší nebo rovna** operátoru
  - `a-b-c` odpovídá `(a-b)-c` a ne `a-(b-c)`: „-” odpovídá `yfx`



Aritmetika

- Předdefinované operátory

  `+, -, *, /, **` mocnina, `//` celočíselné dělení, `mod` zbytek po dělení
- `?- X = 1 + 2.` `X = 1 + 2` = odpovídá unifikaci
- `?- X is 1 + 2.`

  `X = 3` „is” **je speciální předdefinovaný operátor**, který vynutí evaluaci
  - porovnej: `N = (1+1+1+1+1)` `N is (1+1+1+1+1)`
  - pravá strana musí být vyhodnotitelný výraz (bez proměnné)
  - výraz na pravé straně je nejdříve aritmeticky vyhodnocen a pak unifikován s levou stranou

    volání `?- X is Y + 1.` způsobí chybu
- Další speciální předdefinované operátory

  `>, <, >=, =<, =:=` **aritmetická rovnost,** `=\=` **aritmetická nerovnost**
  - porovnej: `1+2 =:= 2+1` `1+2 = 2+1`
  - obě strany musí být vyhodnotitelný výraz: volání `?- 1 < A + 2.` způsobí chybu

Různé typy rovností a porovnání

`X = Y` X a Y jsou unifikovatelné

`X \= Y` X a Y nejsou unifikovatelné, (také `\+ X = Y`)

`X == Y` X a Y jsou identické

porovnej: ?- A == B. . . . no ?- A=B, A==B. . . . B = A yes

`X \== Y` X a Y nejsou identické

porovnej: ?- A \== B. . . . yes ?- A=B, A \== B. . . . A no

`X is Y` Y je aritmeticky vyhodnoceno a výsledek je přiřazen X

`X =:= Y` X a Y jsou si aritmeticky rovny

`X =\= Y` X a Y si aritmeticky nejsou rovny

`X < Y` aritmetická hodnota X je menší než Y (`=<`, `>`, `>=`)

`X @< Y` term X předchází term Y (`@=<`, `@>`, `@>=`)

1. porovnání termů: podle alfabetického n. aritmetického uspořádání
2. porovnání struktur: podle arity, pak hlavního funktoru a pak zleva podle argumentů

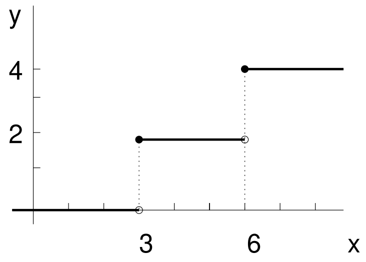
?- f(pavel, g(b)) @< f(pavel, h(a)). . . . yes

Řez, negace

Řez a upnutí

```
f(X,0) :- X < 3, !.
f(X,2) :- 3 =< X, X < 6, !.
f(X,4) :- 6 =< X.
```

přidání **operátoru řezu** „!''




```
?- f(1,Y), Y>2.

f(X,0) :- X < 3, !.  %(1)
f(X,2) :- X < 6, !.  %(2)
f(X,4).

?- f(1,Y).
```

- Smazání řezu v (1) a (2) změní chování programu
- **Upnutí:** po splnění podcílů před řezem se už další klauzule neuvažují

Řez a ořezání

```
f(X,Y) :- s(X,Y).
s(X,Y) :- Y is X + 1.
s(X,Y) :- Y is X + 2.

?- f(1,Z).
Z = 2 ? ;
Z = 3 ? ;
no
```

```
f(X,Y) :- s(X,Y), !.
s(X,Y) :- Y is X + 1.
s(X,Y) :- Y is X + 2.

?-  f(1,Z).
Z = 2 ? ;
no
```

- **Ořezání:** po splnění podcílů před řezem se už neuvažuje další možné splnění těchto podcílů
- Smazání řezu změní chování programu

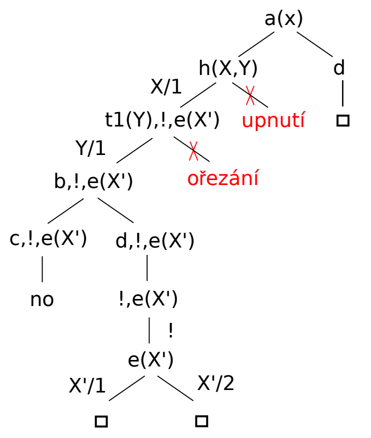
Chování operátoru řezu

- Předpokládejme, že klauzule `H :- T1, T2, ..., Tm, !, ...Tn.` je aktivována voláním cíle G, který je unifikovatelný s H. G=h(X,Y)
- V momentě, kdy je nalezen řez, existuje řešení cílů T1, ..., Tm X=1,Y=1
- **Ořezání:** při provádění řezu se už další možné splnění cílů T1, ..., Tm nehledá a všechny ostatní alternativy jsou odstraněny Y=2
- **Upnutí:** dále už nevyvolávám další klauzule, jejichž hlava je také unifikovatelná s G X=2

```
?- h(X,Y).                          h(X,Y)
                              X=1  /    \  X=2
h(1,Y) :- t1(Y), !.              t1(Y)     a    (vynechej: upnutí)
h(2,Y) :- a.                  Y=1 /    \ Y=2
                                b        c   (vynechej: ořezání)
t1(1) :- b.                    /
t1(2) :- c.
```


Řez: návrat na rodiče

```
?- a(X).

(1) a(X) :- h(X,Y).
(2) a(X) :- d.

(3) h(1,Y) :- t1(Y), !, e(X).
(4) h(2,Y) :- a.

(5) t1(1) :- b.
(6) t1(2) :- c.

(7) b :- c.
(8) b :- d.

(9) d.

(10) e(1) .
(11) e(2).
```



- Po zpracování klauzule s řezem se vracím až na rodiče této klauzule, tj. a(X)

Řez: příklad

```
c(X) :- p(X).                c1(X) :- p(X), !.
c(X) :- v(X).                c1(X) :- v(X).

         p(1).  p(2).           v(2).

?- c(2).                     ?- c1(2).
true ? ;  %p(2)              true ?  ; %p(2)
true ? ;  %v(2)              no
no

?- c(X).                     ?- c1(X).
X = 1 ? ; %p(1)              X = 1 ?  ; %p(1)
X = 2 ? ; %p(2)              no
X = 2 ? ; %v(2)
no
```


Řez: cvičení

1. Porovnejte chování uvedených programů pro zadané dotazy.

```
a(X,X) :- b(X).         a(X,X) :- b(X),!.       a(X,X) :- b(X),c.
a(X,Y) :- Y is X+1.     a(X,Y) :- Y is X+1.     a(X,Y) :- Y is X+1.
b(X) :- X > 10.         b(X) :- X > 10.         b(X) :- X > 10.
                                                c :- !.

?- a(X,Y).
?- a(1,Y).
?- a(11,Y).
```

2. Napište predikát pro výpočet maxima `max( X, Y, Max )`

Typy řezu

- Zlepšení efektivity programu: určíme, které alternativy nemá smysl zkoušet

  Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození
  - `f(X,1) :- X >= 0, !.  f(X,-1) :- X < 0.`

    bez řezu zkouším pro nezáporná čísla 2. klauzuli
- **Modrý řez:** odstraní redundantní řešení
  - `f(X,1) :- X >= 0, !.  f(0,1). f(X,-1) :- X < 0.` bez řezu vrací f(0,1) 2x
- **Červený řez:** odstraní úspěšná řešení
  - `f(X,1) :- X >= 0, !.  f(_X,-1).` bez řezu uspěje 2. klauzule pro nezáporná čísla

Negace jako neúspěch

- **Speciální cíl pro nepravdu (neúspěch)** `fail` **a pravdu** `true`
- X a Y nejsou unifikovatelné: `different(X, Y)`
-
```
different( X, Y ) :- X = Y, !, fail.
different( _X, _Y ).
```
- X je muž: `muz(X)`

```
muz( X ) :- zena( X ), !, fail.
muz( _X ).
```


Negace jako neúspěch: operátor \+

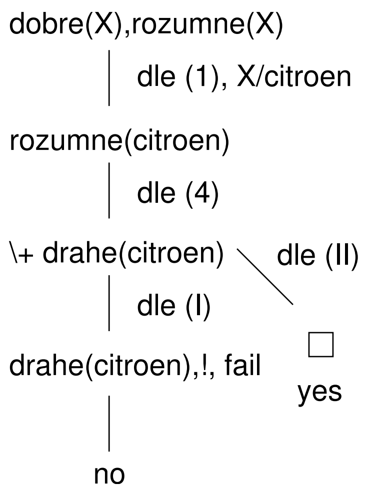
-
```
different(X,Y) :- X = Y, !, fail.     muz(X) :- zena(X), !, fail.
different(_X,_Y).                     muz(_X).
```
- Unární operátor \+ P
  - jestliže P uspěje, potom \+ P neuspěje

    `\+(P) :- P, !, fail.`
  - v opačném případě \+ P uspěje

    `\+(_).`
- `different( X, Y ) :- \+ X=Y.`
- `muz( X ) :- \+ zena( X ).`
- Pozor: takto definovaná negace \+P vyžaduje **konečné odvození** P

Negace a proměnné

```
\+(P) :- P, !, fail. % (I)
\+(_).               % (II)

dobre( citroen ).          % (1)
dobre( bmw ).              % (2)
drahe( bmw ).              % (3)
rozumne( Auto ) :-         % (4)
   \+ drahe( Auto ).

?- dobre( X ), rozumne( X ).
```



Negace a proměnné

```
\+(P) :- P, !, fail. % (I)
\+(_).               % (II)

dobre( citroen ).        % (1)
dobre( bmw ).            % (2)

drahe( bmw ).            % (3)

rozumne( Auto ) :-       % (4)
   \+ drahe( Auto ).

?- rozumne( X ), dobre( X ).
```

rozumne(X), dobre(X)

dle (4)

\+ drahe(X), dobre(X)

dle (I)

drahe(X),!,fail,dobre(X)

dle (3), X/bmw

!, fail, dobre(bmw)

fail,dobre(bmw)

no

Bezpečný cíl

- `?- \+ drahe( citroen ).` yes

  `?- \+ drahe( X ).` no
- `?- rozumne( citroen ).` yes

  `?- rozumne( X ).` no
- `\+ P` **je bezpečný: proměnné** P **jsou v okamžiku volání** P **instanciovány**
  - negaci používáme pouze pro bezpečný cíl P

Chování negace

- `?- \+ drahe( citroen ).` yes

  `?- \+ drahe( X ).` no
- Negace jako neúspěch používá **předpoklad uzavřeného světa**

  pravdivé je pouze to, co je dokazatelné
- `?- \+ drahe( X ).` \+ drahe(X) :- drahe(X),!,fail. \+ drahe(X).

  z definice \+ plyne: není dokazatelné, že existuje X takové, že drahe(X) platí

  tj. **pro všechna** X platí `\+ drahe( X )`
- `?- drahe( X ).`

  PTÁME SE: existuje X takové, že `drahe( X )` platí?
- ALE: pro cíle s negací neplatí **existuje** X takové, že `\+ drahe( X )`

⇒ **negace jako neúspěch není ekvivalentní negaci v matematické logice**

Predikáty na řízení běhu programu I.

- **řez „!"**
- `fail`: cíl, který vždy neuspěje `true`: cíl, který vždy uspěje
- `\+ P`: negace jako neúspěch

  `\+ P :- P, !, fail; true.`
- `once(P)`: vrátí pouze jedno řešení cíle P

  `once(P) :- P, !.`
- Vyjádření **podmínky:** `P -> Q ; R`
  - jestliže platí P tak Q `(P -> Q ; R) :- P, !, Q.`
  - v opačném případě R `(P -> Q ; R) :- R.`
  - příklad: `min(X,Y,Z) :- X =< Y -> Z = X ; Z = Y.`
- `P -> Q`
  - odpovídá: `(P -> Q; fail)`
  - příklad: `zaporne(X) :- number(X) -> X < 0.`

Predikáty na řízení běhu programu II.

- `call(P)`: zavolá cíl P a uspěje, pokud uspěje P
- nekonečná posloupnost backtrackovacích voleb: `repeat`

```
repeat.
repeat :- repeat.
```

klasické použití: **generuj akci** X**, proveď ji a otestuj, zda neskončit**

```
Hlava :- ...
         uloz_stav( StaryStav ),
         repeat,
           generuj( X ),       % deterministické: generuj, provadej, testuj
           provadej( X ),
           testuj( X ),
         !,
         obnov_stav( StaryStav ),
         ...
```