

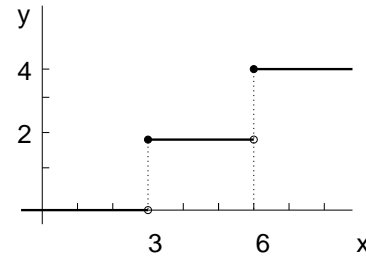
Řez a upnutí

$f(X,0) :- X < 3, !.$

přidání **operátoru řezu** `,!,'`

$f(X,2) :- 3 \leq X, X < 6, !.$

$f(X,4) :- 6 \leq X.$



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

$f(X,0) :- X < 3, !. \quad \% (1)$

$f(X,2) :- X < 6, !. \quad \% (2)$

$f(X,4).$

?- $f(1,Y).$

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

Řez, negace

Řez a ořezání

$f(X,Y) :- s(X,Y).$

$f(X,Y) :- s(X,Y), !.$

$s(X,Y) :- Y \text{ is } X + 1.$

$s(X,Y) :- Y \text{ is } X + 1.$

$s(X,Y) :- Y \text{ is } X + 2.$

$s(X,Y) :- Y \text{ is } X + 2.$

?- $f(1,Z).$

?- $f(1,Z).$

$Z = 2 ? ;$

$Z = 2 ? ;$

$Z = 3 ? ;$

no

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

?- $h(X,Y).$

$h(X,Y)$

$h(1,Y) :- t1(Y), !.$

$X=1 / \quad \backslash \quad X=2$

$h(2,Y) :- a.$

$t1(Y) \quad a \quad (\text{vynechej: upnutí})$

$t1(1) :- b.$

$Y=1 / \quad \backslash \quad Y=2$

$b \quad c \quad (\text{vynechej: ořezání})$

$t1(2) :- c.$

$/$

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: cvičení

- 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).
```

- Napište predikát pro výpočet maxima max(X, Y, Max)

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  

```

Typy řezu

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

```
dobre(X),rozumne(X)
    |
    dle (1), X/citroen
    |
rozumne(citroen)
    |
    dle (4)
    |
\+ drahe(citroen)  dle (II)
    |
    dle (I)
    |
drahe(citroen),!,fail  □
                        yes
    |
    no
```

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

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

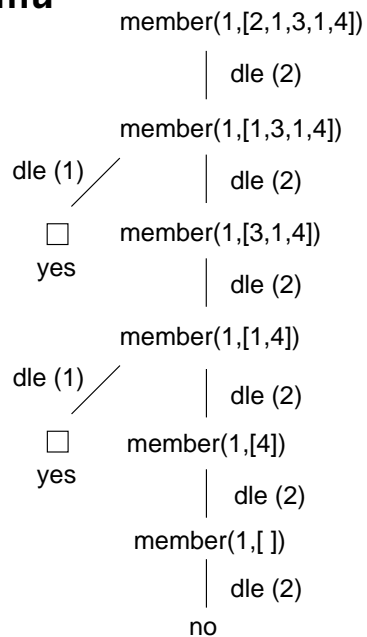
Predikáty na řízení běhu programu II.

- `call(P)`: zavolá cíl P a uspěje, pokud uspěje P
- nekonečná posloupnost backtrackovacích voleb: `repeat`
`repeat.`
`repeat :- repeat.`
- klasické použití: **generuj akci X, proved' ji a otestuj, zda neskončit**
`Hlava :- ...`
`uloz_stav(StaryStav),`
`repeat,`
`generuj(X), % deterministické: generuj, provadej, testuj`
`provadej(X),`
`testuj(X),`
`!,`
`obnov_stav(StaryStav),`
`...`

Seznamy

Prvek seznamu

- `member(X, S)`
- platí: `member(b, [a,b,c])`.
- neplatí: `member(b, [[a,b]| [c]])`.
- X je prvek seznamu S, když
 - X je hlava seznamu S nebo
 $\text{member}(X, [X | _])$. % (1)
 - X je prvek těla seznamu S
 $\text{member}(X, [_ | \text{Te}lo]) :- \text{member}(X, \text{Te}lo)$. % (2)
- Další příklady použití:
 - `member(X, [1,2,3])`.
 - `member(1, [2,1,3,1])`.



Reprezentace seznamu

- **Seznam:** `[a, b, c]`, prázdný seznam `[]`
- **Hlava (libovolný objekt), tělo (seznam):** `.(Hlava, TeLo)`
 - všechny strukturované objekty stromy - i seznamy
 - funktor ".", dva argumenty
 - `.(a, .(b, .(c, []))) = [a, b, c]`
 - notace: `[Hlava | TeLo] = [a|TeLo]`
 TeLo je v `[a|TeLo]` seznam, tedy píšeme `[a, b, c] = [a | [b, c]]`
- Lze psát i: `[a,b|TeLo]`
 - před "|" je libovolný počet prvků seznamu, za "|" je seznam zbývajících prvků
 - `[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]`
 - pozor: `[[a,b] | [c]] ≠ [a,b | [c]]`
- Seznam jako **neúplná datová struktura:** `[a,b,c|T]`
 - Seznam = `[a,b,c|T]`, `T = [d,e|S]`, Seznam = `[a,b,c,d,e|S]`

Spojení seznamů

- `append(L1, L2, L3)`
- Platí: `append([a,b], [c,d], [a,b,c,d])`
- Neplatí: `append([b,a], [c,d], [a,b,c,d])`,
`append([a,[b]], [c,d], [a,b,c,d])`
- Definice:
 - pokud je 1. argument prázdný seznam, pak 2. a 3. argument jsou stejné seznamy:
`append([], S, S)`.
 - pokud je 1. argument neprázdný seznam, pak má 3. argument stejnou hlavu jako 1.:
`append([X|S1], S2, [X|S3]) :- append(S1, S2, S3)`.



Příklady použití append

- `append([], S, S)`.
`append([X|S1], S2, [X|S3]) :- append(S1, S2, S3)`.
- **Spojení seznamů:** `append([a,b,c], [1,2,3], S)`.
`S = [a,b,c,1,2,3]`
`append([a, [b,c], d], [a, [], b], S)`.
`S = [a, [b,c], d, a, [], b]`
- **Dekompozice seznamu na dva seznamy:** `append(S1, S2, [a, b])`.
`S1 = [], S2 = [a,b] ;`
`S1 = [a], S2 = [b] ? ;`
`S1 = [a,b], S2 = []`
- **Vyhledávání v seznamu:** `append(Pred, [c | Za], [a,b,c,d,e])`.
`Pred = [a,b], Za = [d,e]`
- **Předchůdce a následník:** `append(_, [Pred,c,Za|_], [a,b,c,d,e])`.
`Pred = b, Za = d`

Smazání prvku seznamu `delete(X, S, S1)`

- Seznam `S1` odpovídá seznamu `S`, ve kterém je smazán prvek `X`
 - jestliže `X` je hlava seznamu `S`, pak výsledkem je tělo `S`
`delete(X, [X|Telo], Telo)`.
 - jestliže `X` je v těle seznamu, pak `X` je smazán až v těle
`delete(X, [Y|Telo], [Y|Telo1]) :- delete(X, Telo, Telo1)`.
- `delete` smaže libovolný výskyt prvku pomocí backtrackingu
`?- delete(a, [a,b,a,a], S)`.
`S = [b,a,a];`
`S = [a,b,a];`
`S = [a,b,a]`
- `delete`, který smaže pouze první výskyt prvku `X`
 - `delete(X, [X|Telo], Telo) :- !`.
`delete(X, [Y|Telo], [Y|Telo1]) :- delete(X, Telo, Telo1)`.