

**Řez, negace**

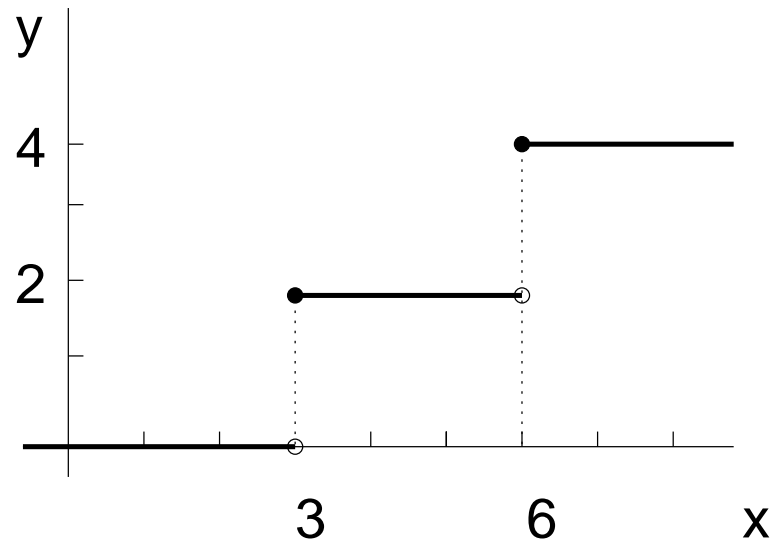
# Řez a upnutí

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

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

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

přidání **operátoru řezu** `!, !'`



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

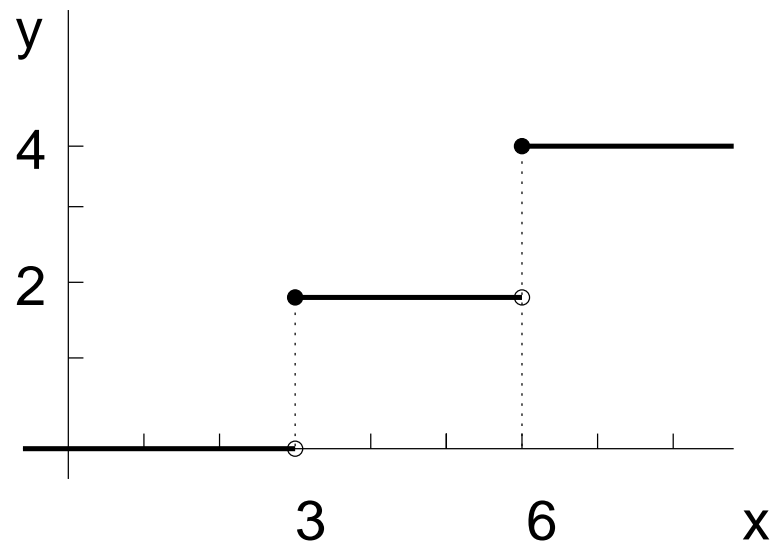
# Řez a upnutí

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

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

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

přidání **operátoru řezu** `,,!’’`



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

**Upnutí:** po splnění podcílů před řezem se už další klauzule neuvažují

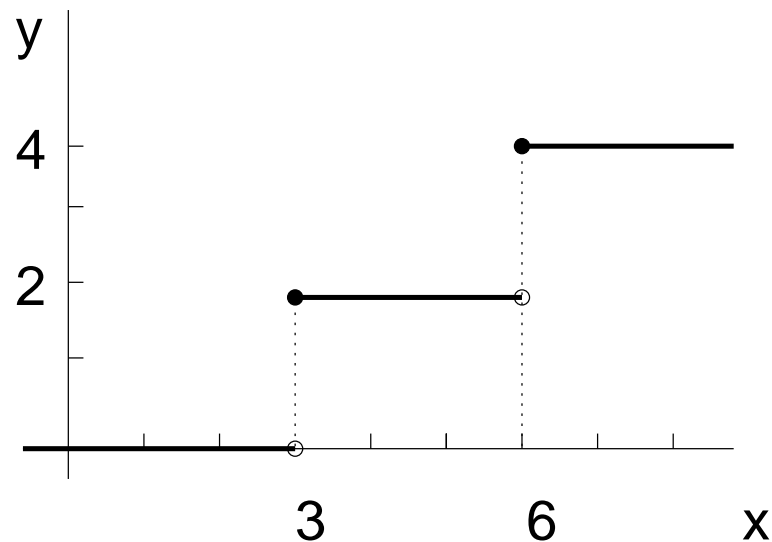
# Řez a upnutí

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

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

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

přidání **operátoru řezu** `,,!'`



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

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

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

$f(X,4).$

**Upnutí:** po splnění podcílů před řezem se už další klauzule neuvažují

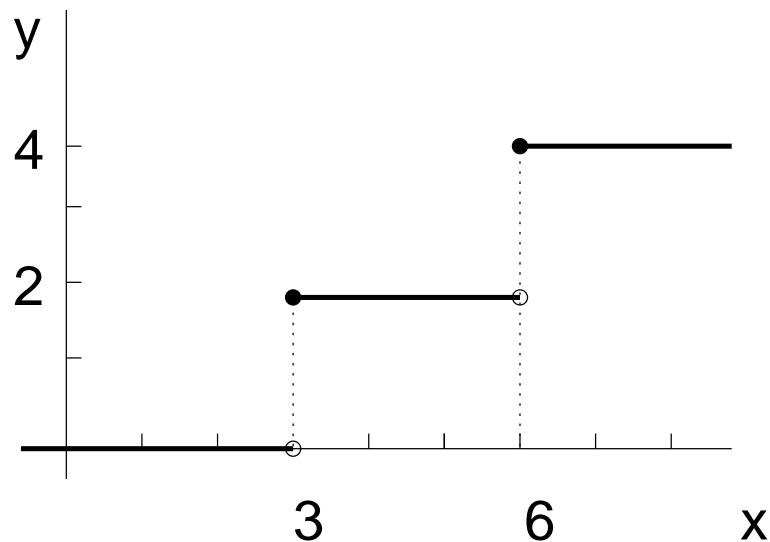
# Řez a upnutí

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

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

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

přidání **operátoru řezu** `,,!'`



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

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

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

$f(X,4).$

$?- f(1,Y).$

● Smazání řezu v (1) a (2) změní chování programu

● **Upnutí:** po splnění podcílů před řezem se už další klauzule neuvažují

# Řez a ořezání

$f(X,Y) \text{ :- } s(X,Y).$

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

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

$?- f(1,Z).$

# Řez a ořezání

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

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

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

?-  $f(1,Z).$

$Z = 2 ? ;$

$Z = 3 ? ;$

no

# Řez a ořezání

$f(X,Y) \text{ :- } s(X,Y).$

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

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

$?- f(1,Z).$

$Z = 2 ? ;$

$Z = 3 ? ;$

no

$f(X,Y) \text{ :- } s(X,Y), !.$

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

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

$?- f(1,Z).$

- **Ořezání:** po splnění podcílů před řezem se už neuvažuje další možné splnění těchto podcílů



# Řez a ořezání

$f(X,Y) \text{ :- } s(X,Y).$

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

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

$?- f(1,Z).$

$Z = 2 ? ;$

$Z = 3 ? ;$

no

$f(X,Y) \text{ :- } s(X,Y), !.$

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

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

$?- f(1,Z).$

$Z = 2 ? ;$

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  $H :- T_1, T_2, \dots, T_m, !, \dots, T_n.$  je aktivována voláním cíle  $G$ , který je unifikovatelný s  $H$ .  $G=h(X,Y)$
- V momentě, kdy je nalezen řez, existuje řešení cílů  $T_1, \dots, T_m$   $X=1, Y=1$
- **Ořezání:** při provádění řezu se už další možné splnění cílů  $T_1, \dots, T_m$  nehledá a všechny ostatní alternativy jsou odstraněny  $Y=2$
- **Upnutí:** dále už nevyvolávám další klauzule, jejichž hlava je také unifikovatelná s  $G$   $X=2$

$?- h(X,Y).$ $h(1,Y) :- t1(Y), !.$ $h(2,Y) :- a.$ $t1(1) :- b.$ $t1(2) :- c.$	<table style="border: none; margin-left: auto; margin-right: auto;"> <tr> <td colspan="2" style="text-align: center;"><math>h(X,Y)</math></td> </tr> <tr> <td style="text-align: center;"><math>X=1</math></td> <td style="text-align: center;"><math>X=2</math></td> </tr> <tr> <td style="text-align: center;">/</td> <td style="text-align: center;">\</td> </tr> <tr> <td style="text-align: center;"><math>t1(Y)</math></td> <td style="text-align: center;"><math>a</math></td> </tr> <tr> <td colspan="2" style="text-align: center;">(vynechej: upnutí)</td> </tr> <tr> <td style="text-align: center;"><math>Y=1</math></td> <td style="text-align: center;"><math>Y=2</math></td> </tr> <tr> <td style="text-align: center;">/</td> <td style="text-align: center;">\</td> </tr> <tr> <td style="text-align: center;"><math>b</math></td> <td style="text-align: center;"><math>c</math></td> </tr> <tr> <td colspan="2" style="text-align: center;">(vynechej: ořezání)</td> </tr> <tr> <td style="text-align: center;">/</td> <td></td> </tr> </table>	$h(X,Y)$		$X=1$	$X=2$	/	\	$t1(Y)$	$a$	(vynechej: upnutí)		$Y=1$	$Y=2$	/	\	$b$	$c$	(vynechej: ořezání)		/	
$h(X,Y)$																					
$X=1$	$X=2$																				
/	\																				
$t1(Y)$	$a$																				
(vynechej: upnutí)																					
$Y=1$	$Y=2$																				
/	\																				
$b$	$c$																				
(vynechej: ořezání)																					
/																					

# 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) .

# Řez: návrat na rodiče

?- a(X).

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) .

# 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.

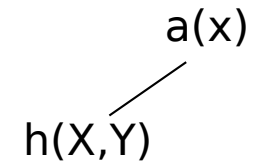
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

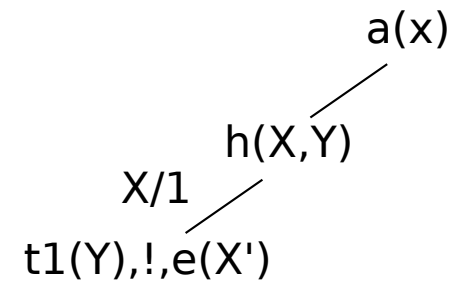
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

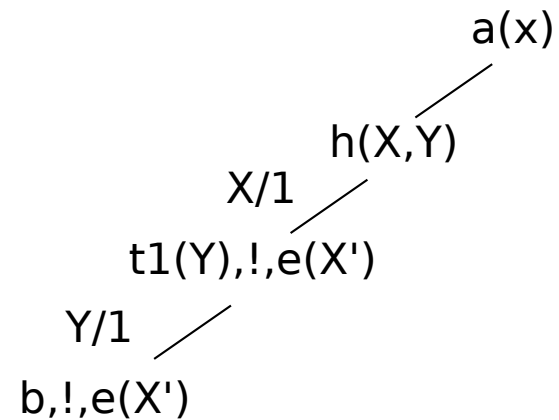
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

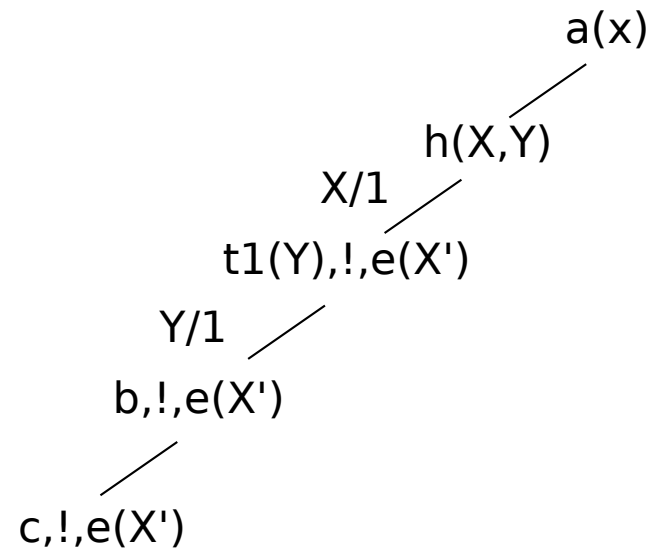
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .





# 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.

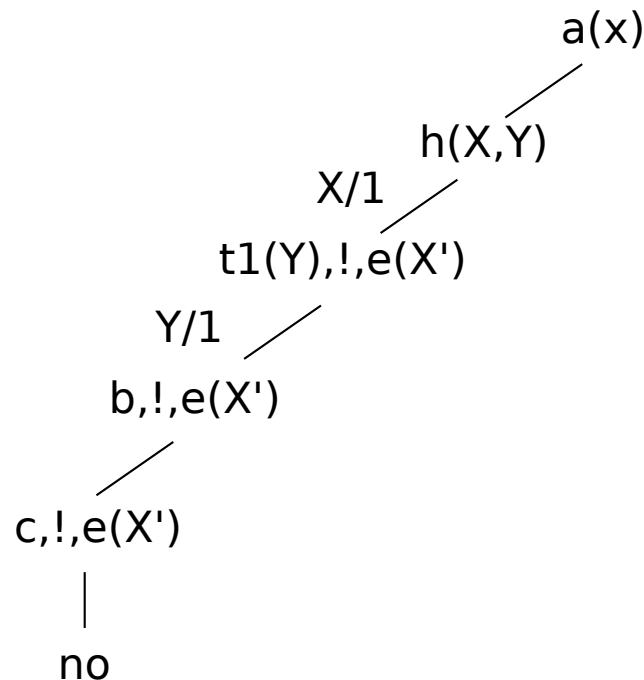
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

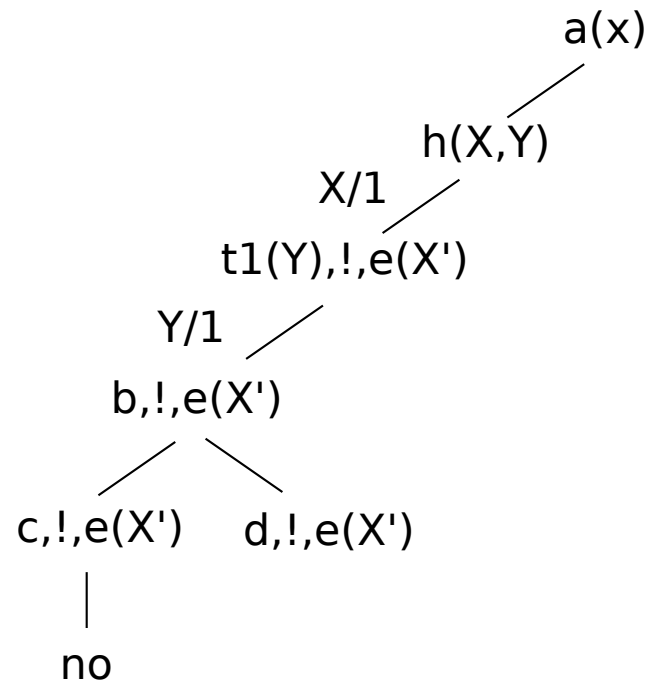
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

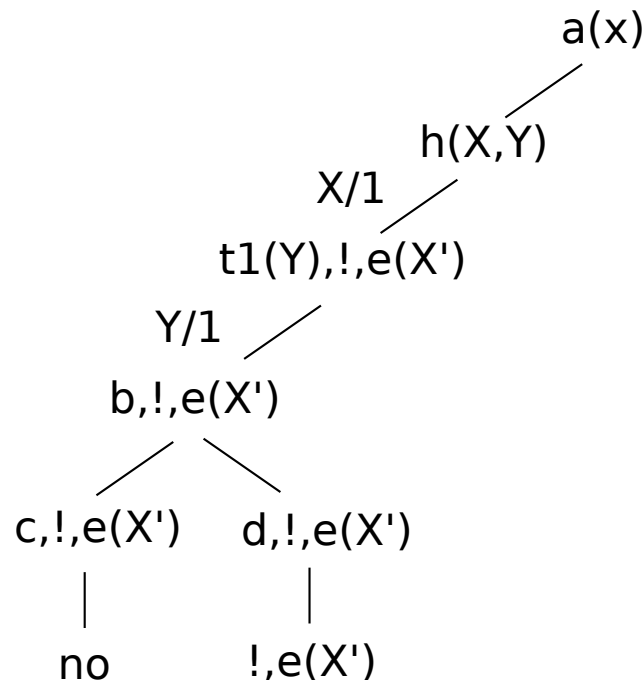
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

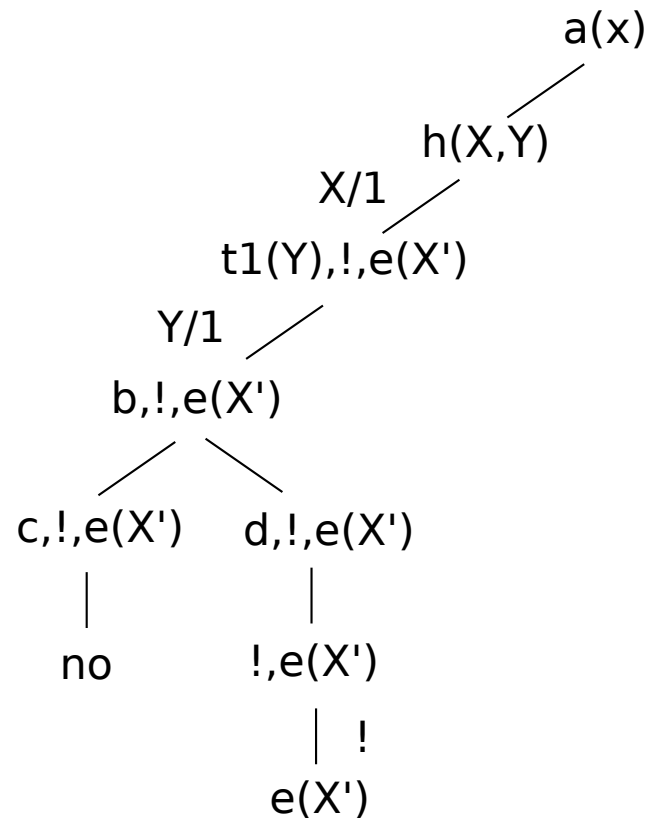
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

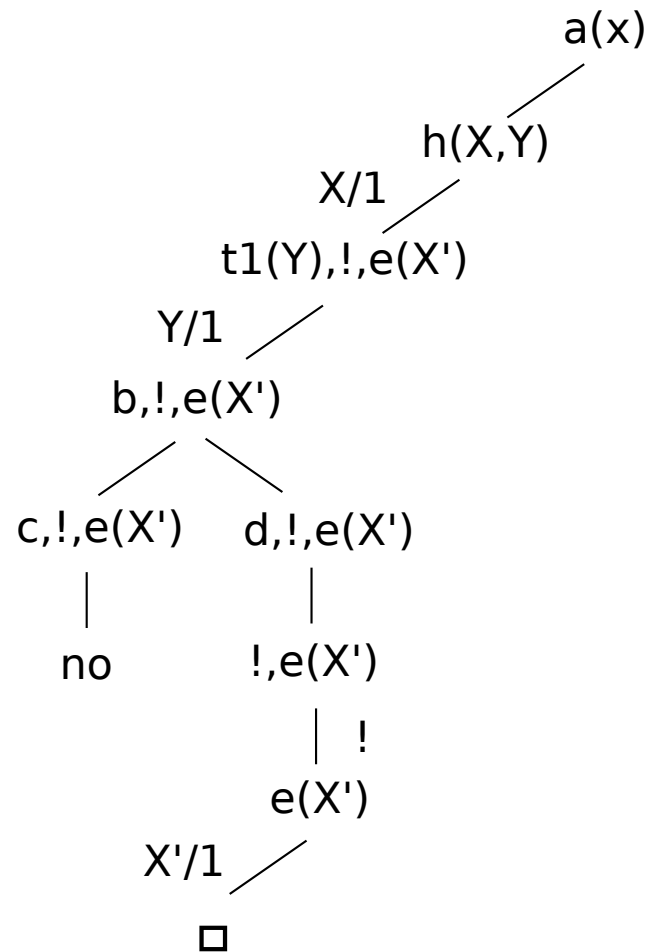
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

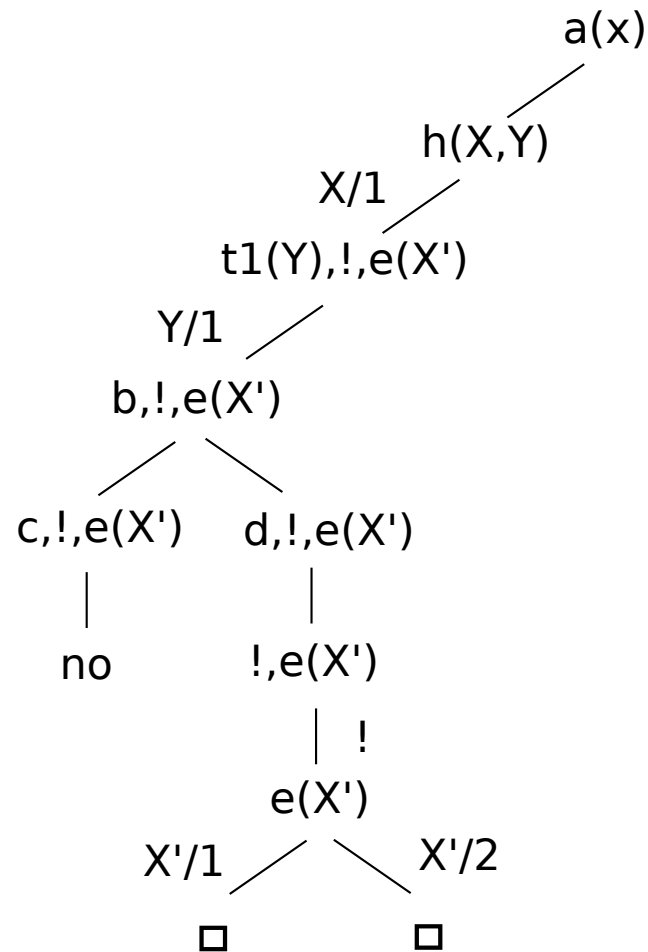
(7) b :- c.

(8) b :- d.

(9) d.

(10) e(1) .

(11) e(2) .



# 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.

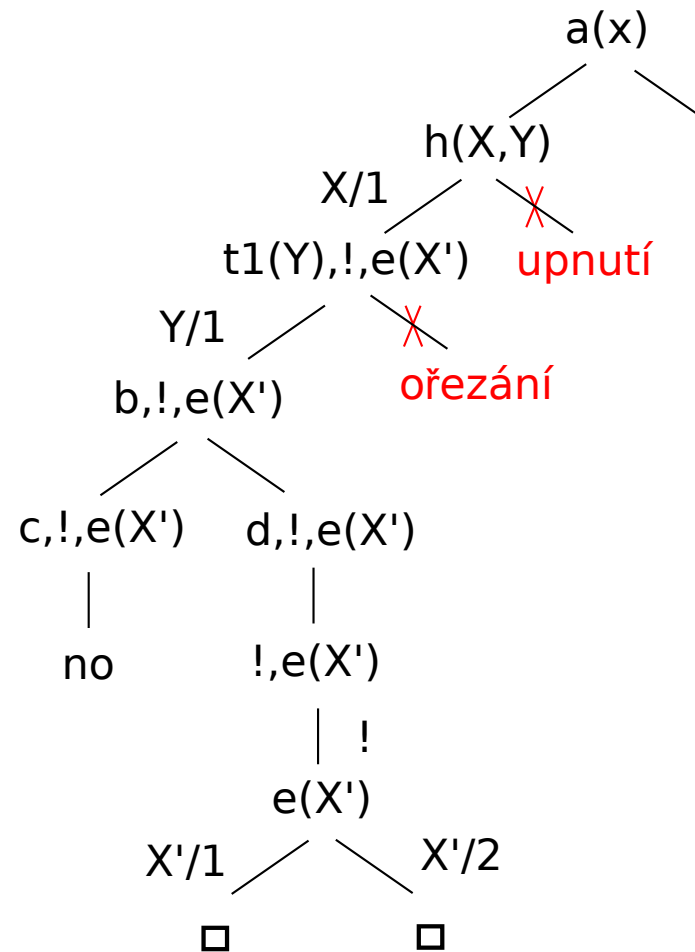
(7) b :- c.

(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: 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.

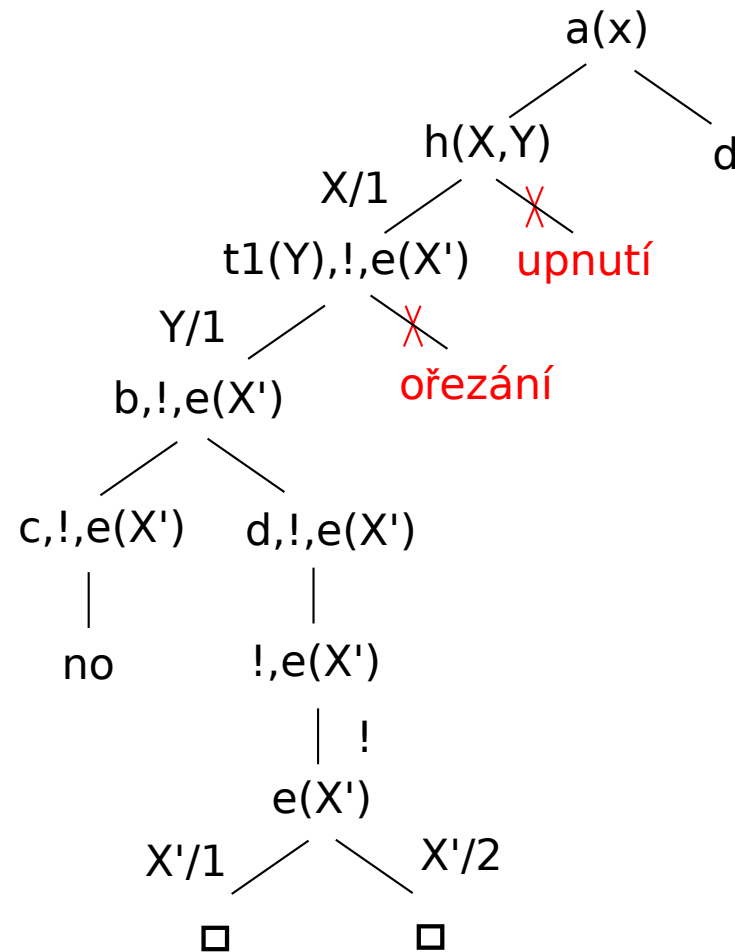
(7) b :- c.

(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: 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.

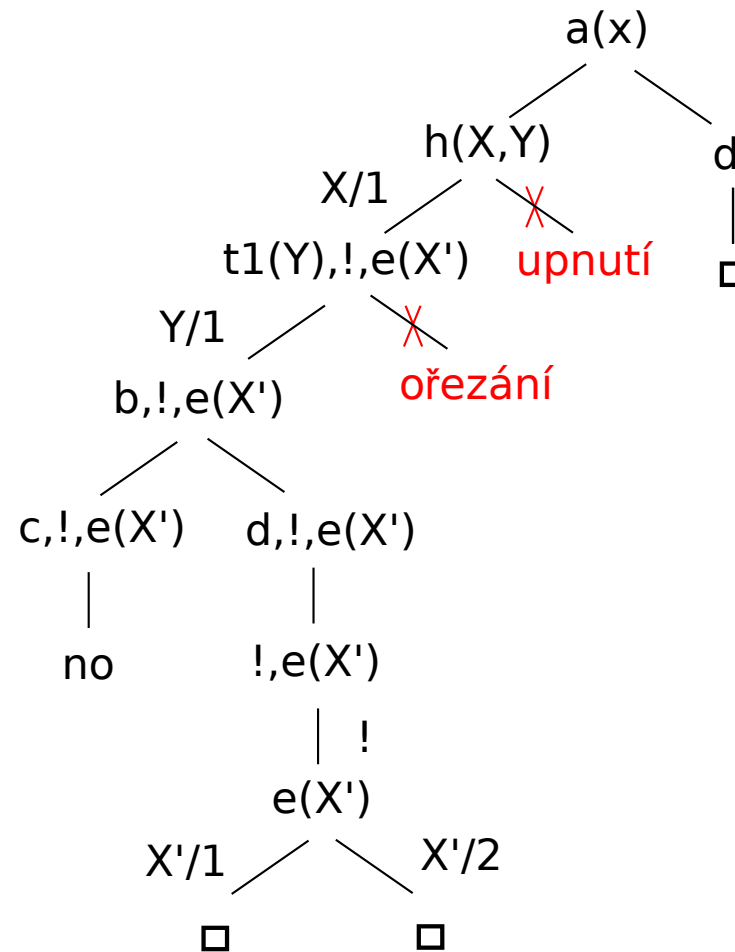
(7) b :- c.

(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

$c(X) :- p(X).$

$c(X) :- v(X).$

$p(1).$     $p(2).$     $v(2).$

$?- c(2).$

# Řez: příklad

`c(X) :- p(X).`

`c(X) :- v(X).`

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

`?- c(2).`

`true ? ; %p(2)`

`true ? ; %v(2)`

`no`

`?- c(X).`

# Řez: příklad

```
c(X) :- p(X).
```

```
c(X) :- v(X).
```

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

```
?- c(2).
```

```
true ? ; %p(2)
```

```
true ? ; %v(2)
```

```
no
```

```
?- c(X).
```

```
X = 1 ? ; %p(1)
```

```
X = 2 ? ; %p(2)
```

```
X = 2 ? ; %v(2)
```

```
no
```

## Řez: příklad

```
c(X) :- p(X).
```

```
c(X) :- v(X).
```

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

```
c1(X) :- v(X).
```

```
p(1). p(2).
```

```
v(2).
```

```
?- c(2).
```

```
true ? ; %p(2)
```

```
true ? ; %v(2)
```

```
no
```

```
?- c1(2).
```

```
?- c(X).
```

```
X = 1 ? ; %p(1)
```

```
X = 2 ? ; %p(2)
```

```
X = 2 ? ; %v(2)
```

```
no
```

## Řez: příklad

```
c(X) :- p(X).
```

```
c(X) :- v(X).
```

```
p(1). p(2).
```

```
v(2).
```

```
?- c(2).
```

```
true ? ; %p(2)
```

```
true ? ; %v(2)
```

```
no
```

```
?- c(X).
```

```
X = 1 ? ; %p(1)
```

```
X = 2 ? ; %p(2)
```

```
X = 2 ? ; %v(2)
```

```
no
```

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

```
c1(X) :- v(X).
```

```
?- c1(2).
```

```
true ? ; %p(2)
```

```
no
```

```
?- c1(X).
```

## Řez: příklad

```
c(X) :- p(X).
```

```
c(X) :- v(X).
```

```
p(1). p(2).
```

```
v(2).
```

```
?- c(2).
```

```
true ? ; %p(2)
```

```
true ? ; %v(2)
```

```
no
```

```
?- c(X).
```

```
X = 1 ? ; %p(1)
```

```
X = 2 ? ; %p(2)
```

```
X = 2 ? ; %v(2)
```

```
no
```

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

```
c1(X) :- v(X).
```

```
?- c1(2).
```

```
true ? ; %p(2)
```

```
no
```

```
?- c1(X).
```

```
X = 1 ? ; %p(1)
```

```
no
```

# Řez: cvičení

1. Porovnejte chování uvedených programů pro zadané dotazy.

<code>a(X,X) :- b(X).</code>	<code>a(X,X) :- b(X), !.</code>	<code>a(X,X) :- b(X), c.</code>
<code>a(X,Y) :- Y is X+1.</code>	<code>a(X,Y) :- Y is X+1.</code>	<code>a(X,Y) :- Y is X+1.</code>
<code>b(X) :- X &gt; 10.</code>	<code>b(X) :- X &gt; 10.</code>	<code>b(X) :- X &gt; 10.</code>
		<code>c :- !.</code>

?- 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

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

# Typy řezu

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

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

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

bez řezu zkouším pro nezáporná čísla 2. klauzuli

# Typy řezu

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

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

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

bez řezu zkouším pro nezáporná čísla 2. klauzuli

- **Modrý řez:** odstraní redundantní řešení

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

# Typy řezu

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

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

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

bez řezu zkouším pro nezáporná čísla 2. klauzuli

- **Modrý řez:** odstraní redundantní řešení

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

bez řezu vrátí  $f(0,1)$  2x

# Typy řezu

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

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

- $f(X,1) :- X \geq 0, !. f(X,-1) :- X < 0.$

bez řezu zkouším pro nezáporná čísla 2. klauzuli

- **Modrý řez:** odstraní redundantní řešení

- $f(X,1) :- X \geq 0, !. f(0,1). f(X,-1) :- X < 0.$

bez řezu vrací  $f(0,1)$  2x

- **Červený řez:** odstraní úspěšná řešení

- $f(X,1) :- X \geq 0, !. f(_X,-1).$

# Typy řezu

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

Poznámka: na vstupu pro X očekávám číslo

- **Zelený řez:** odstraní pouze neúspěšná odvození

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

bez řezu zkouším pro nezáporná čísla 2. klauzuli

- **Modrý řez:** odstraní redundantní řešení

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

bez řezu vrací  $f(0,1)$  2x

- **Červený řez:** odstraní úspěšná řešení

- $f(X,1) :- X \geq 0, !. \quad f(_X,-1).$

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

`\+(_).`



# 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 ).
```

# 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(X),rozumne(X)

```
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 ).
```

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

dobre(X),rozumne(X)

| dle (1), X/citroen

rozumne(citroen)

# 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)
```

# 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 (I)
```

```
drahe(citroen),!, fail
```

# 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 (I)
```

```
drahe(citroen),!, fail
```

```
|  
no
```



# 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 (I)

|\ dle (II)

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 ).
```

# 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 ).
```

# Negace a proměnné

rozumne(X), dobre(X)

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

```
\+(_). % (II)
```

```
dobre( citroen ). % (1)
```

```
dobre( bmw ). % (2)
```

```
drahe( bmw ). % (3)
```

```
rozumne( Auto ) :- % (4)
```

```
    \+ drahe( Auto ).
```

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

# 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)

# 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 (1)

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

| dle (3), X/bmw

!, fail, dobre(bmw)

# 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 (1)

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

| dle (3), X/bmw

!, fail, dobre(bmw)

| fail,dobre(bmw)



# 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

● `?- drahe( X ).`

PTÁME SE: existuje X takové, že `drahe( X )` platí?

● `?- \+ 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 )`

# Chování negace

● ?- \+ drahe( citroen ). yes

?- \+ drahe( X ). no

● ?- drahe( X ).

PTÁME SE: existuje X takové, že drahe( X ) platí?

● ?- \+ 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 )

● TEDY: pro cíle s negací neplatí **existuje X** takové, že \+ drahe( X )

# Chování negace

● ?- \+ drahe( citroen ). yes

?- \+ drahe( X ). no

● ?- drahe( X ).

PTÁME SE: existuje X takové, že drahe( X ) platí?

● ?- \+ 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 )

● TEDY: pro cíle s negací neplatí **existuje X** takové, že \+ drahe( X )

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

● Negace jako neúspěch používá **předpoklad uzavřeného světa**

pravdivé je pouze to, co je dokazatelné

# 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.`

# 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, !.`

# 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.`



# 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`

# 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.`

# 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

# Reprezentace seznamu

- **Seznam**: [a, b, c], prázdný seznam []
- **Hlava (libovolný objekt), tělo (seznam)**: .(Hlava, TeĽo)
  - všechny strukturované objekty stromy – i seznamy
  - funktor ".", dva argumenty
  - .(a, .(b, .(c, []))) = [a, b, c]
  - notace: [ Hlava | TeĽo ] = [a|TeĽo]

# Reprezentace seznamu

- **Seznam**: [a, b, c], prázdný seznam []
- **Hlava (libovolný objekt), tělo (seznam)**: .(Hlava, TeĽo)
  - všechny strukturované objekty stromy – i seznamy
  - funktor ".", dva argumenty
  - .(a, .(b, .(c, []))) = [a, b, c]
  - notace: [ Hlava | TeĽo ] = [a|TeĽo]  
TeĽo je v [a|TeĽo] seznam, tedy píšeme [ a, b, c ] = [ a | [ b, c ] ]

# Reprezentace seznamu

- **Seznam**: [a, b, c], prázdný seznam []
- **Hlava (libovolný objekt), tělo (seznam)**: .(Hlava, TeĽo)
  - všechny strukturované objekty stromy – i seznamy
  - funktor ".", dva argumenty
  - $.(a, .(b, .(c, []))) = [a, b, c]$
  - notace: [ Hlava | TeĽo ] = [a|TeĽo]  
TeĽo je v [a|TeĽo] seznam, tedy píšeme [ a, b, c ] = [ a | [ b, c ] ]
- Lze psát i: [a,b|TeĽo]
  - před "|" je libovolný počet prvků seznamu , za "|" je seznam zbývajících prvků
  - [a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]



# Reprezentace seznamu

- **Seznam:**  $[a, b, c]$ , prázdný seznam  $[]$
- **Hlava (libovolný objekt), tělo (seznam):**  $.(Hlava, TeĽo)$ 
  - všechny strukturované objekty stromy – i seznamy
  - funktor ".", dva argumenty
  - $.(a, .(b, .(c, []))) = [a, b, c]$
  - notace:  $[Hlava | TeĽo] = [a|TeĽo]$ 

TeĽo je v  $[a|TeĽo]$  seznam, tedy píšeme  $[a, b, c] = [a|[b, c]]$
- Lze psát i:  $[a,b|TeĽo]$ 
  - před "|" je libovolný počet prvků seznamu, za "|" je seznam zbývajících prvků
  - $[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]$
  - pozor:  $[ [a,b] | [c] ] \neq [ a,b | [c] ]$

# Reprezentace seznamu

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

# 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  
`member( X, [ X | _ ] )`.   %(1)
  - X je prvek těla seznamu S  
`member( X, [ _ | Telo ] ) :-  
    member( X, Telo )`.   %(2)

# Prvek seznamu

`member(1,[2,1,3,1,4])`

- `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  
`member( X, [ X | _ ] )`.   %(1)
  - X je prvek těla seznamu S  
`member( X, [ _ | Telo ] ) :-  
    member( X, Telo )`.   %(2)

# 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

`member( X, [ X | _ ] )`.   %(1)

- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-  
    member( X, Telo )`.   %(2)

`member(1,[2,1,3,1,4])`

|  
dle (2)

`member(1,[1,3,1,4])`

# Prvek seznamu

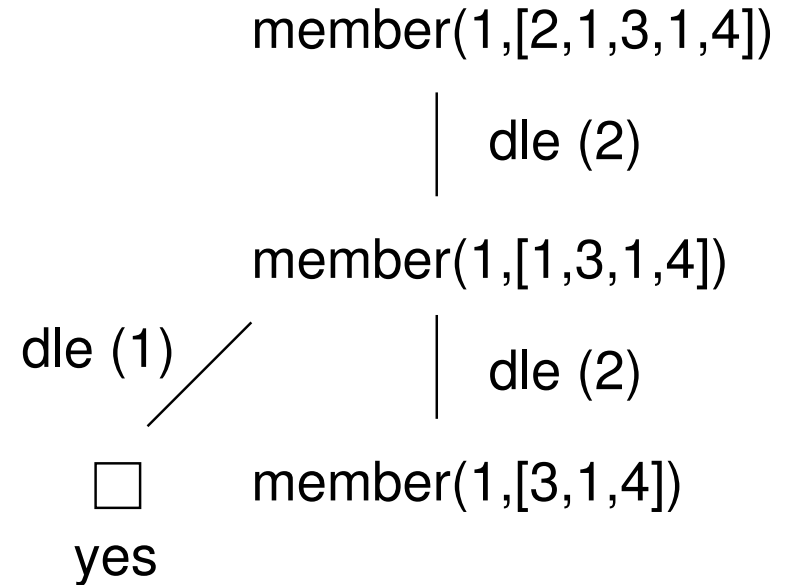
- `member( X, S )`
- platí: `member( b, [a,b,c] )`.
- neplatí: `member( b, [[a,b]| [c]] )`.
- X je prvek seznamu S, když

- X je hlava seznamu S nebo

`member( X, [ X | _ ] ).` %(1)

- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-  
member( X, Telo ).` %(2)



# Prvek seznamu

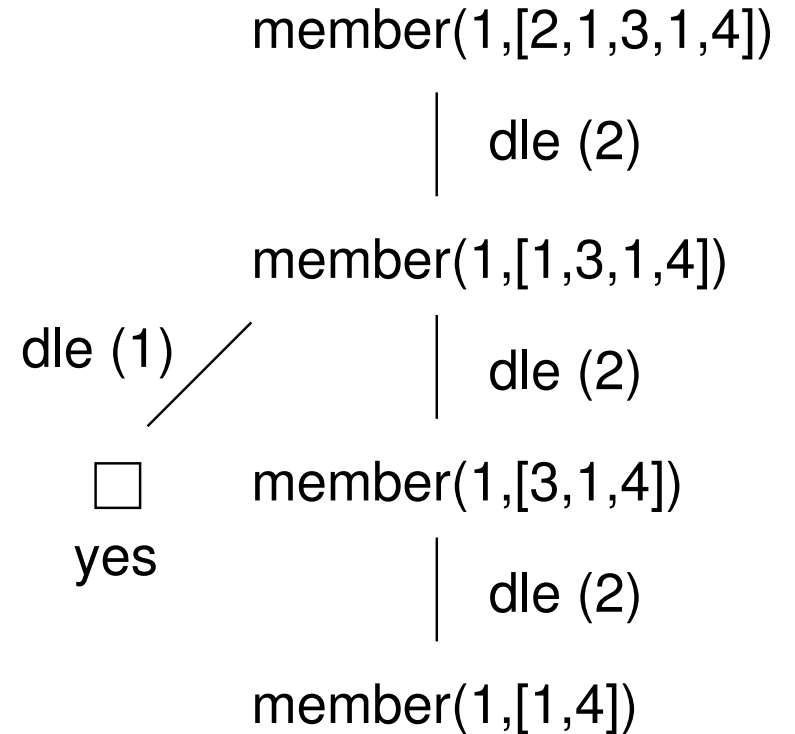
- `member( X, S )`
- platí: `member( b, [a,b,c] )`.
- neplatí: `member( b, [[a,b]|[c]] )`.
- X je prvek seznamu S, když

- X je hlava seznamu S nebo

`member( X, [ X | _ ] )`.   %(1)

- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-`  
`member( X, Telo )`.   %(2)



# Prvek seznamu

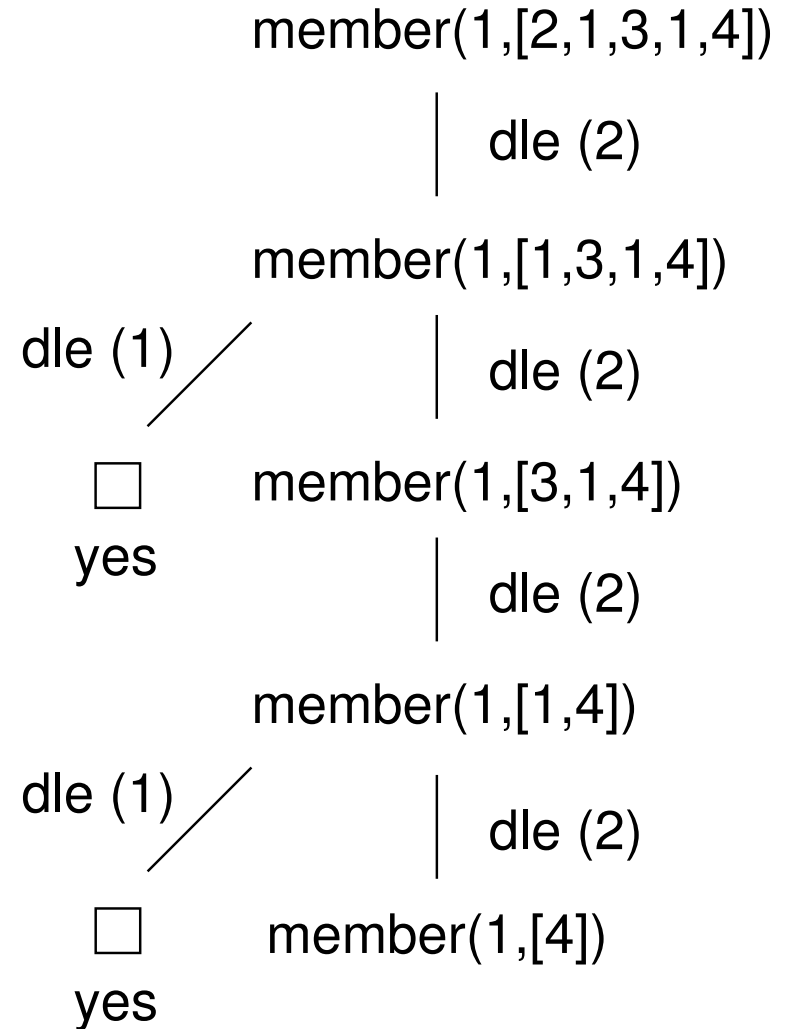
- `member( X, S )`
- platí: `member( b, [a,b,c] )`.
- neplatí: `member( b, [[a,b]| [c]] )`.
- X je prvek seznamu S, když

- X je hlava seznamu S nebo

`member( X, [ X | _ ] )`.   %(1)

- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-  
member( X, Telo )`.   %(2)





# Prvek seznamu

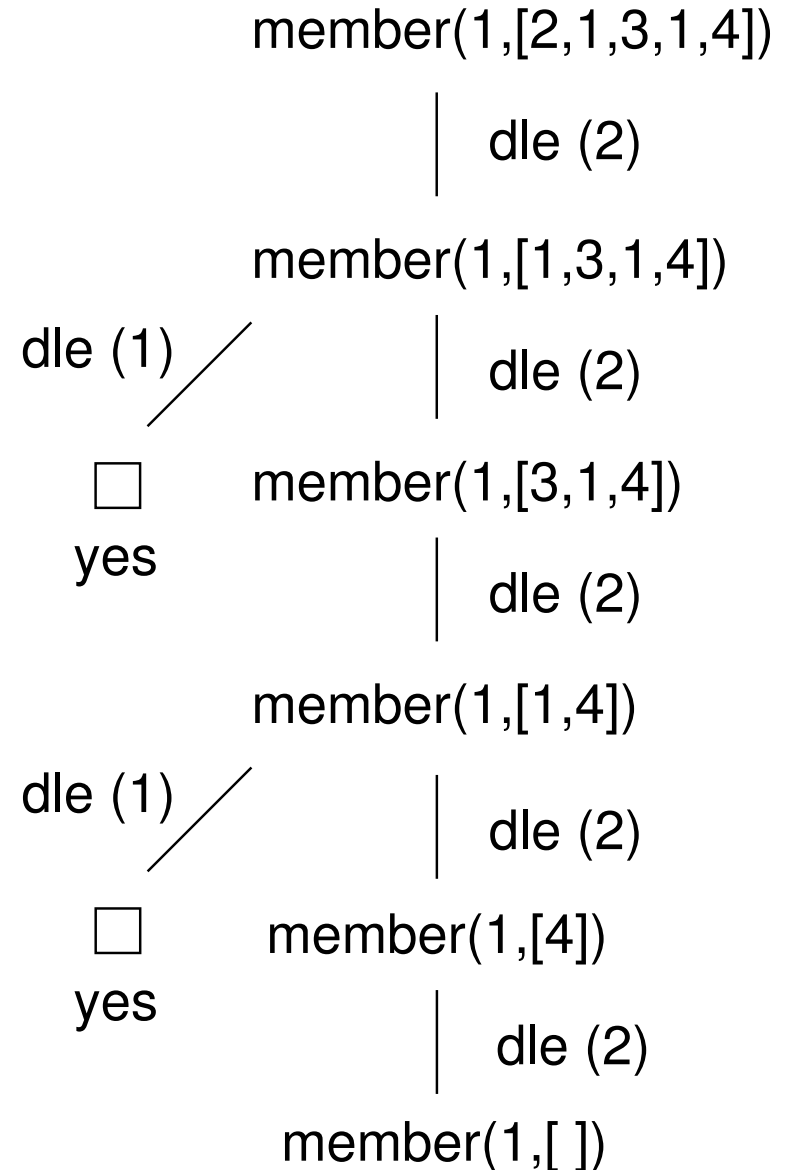
- `member( X, S )`
- platí: `member( b, [a,b,c] )`.
- neplatí: `member( b, [[a,b]|[c]] )`.
- X je prvek seznamu S, když

- X je hlava seznamu S nebo

`member( X, [ X | _ ] ).` %(1)

- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-  
member( X, Telo ).` %(2)



# Prvek seznamu

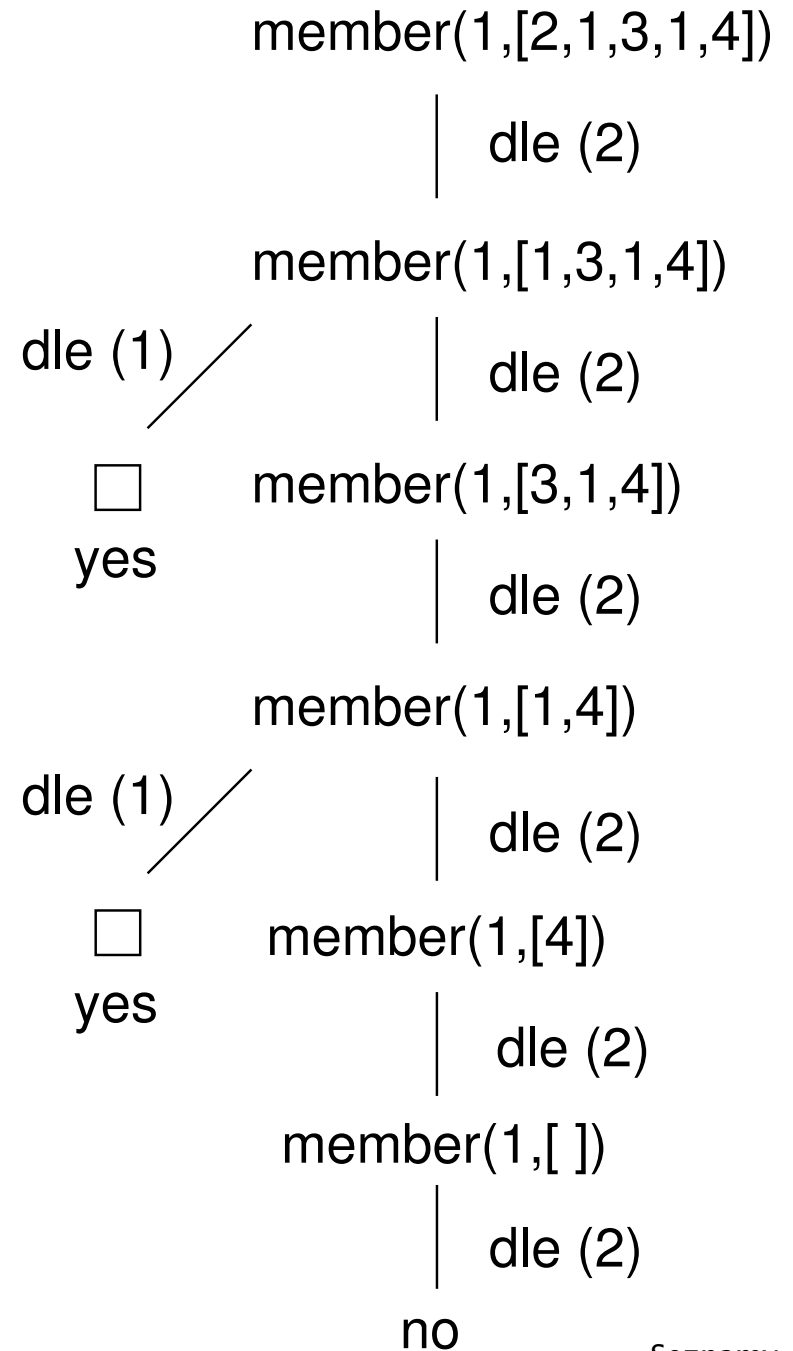
- `member( X, S )`
- platí: `member( b, [a,b,c] )`.
- neplatí: `member( b, [[a,b]|[c]] )`.
- X je prvek seznamu S, když

- X je hlava seznamu S nebo

`member( X, [ X | _ ] ).` %(1)

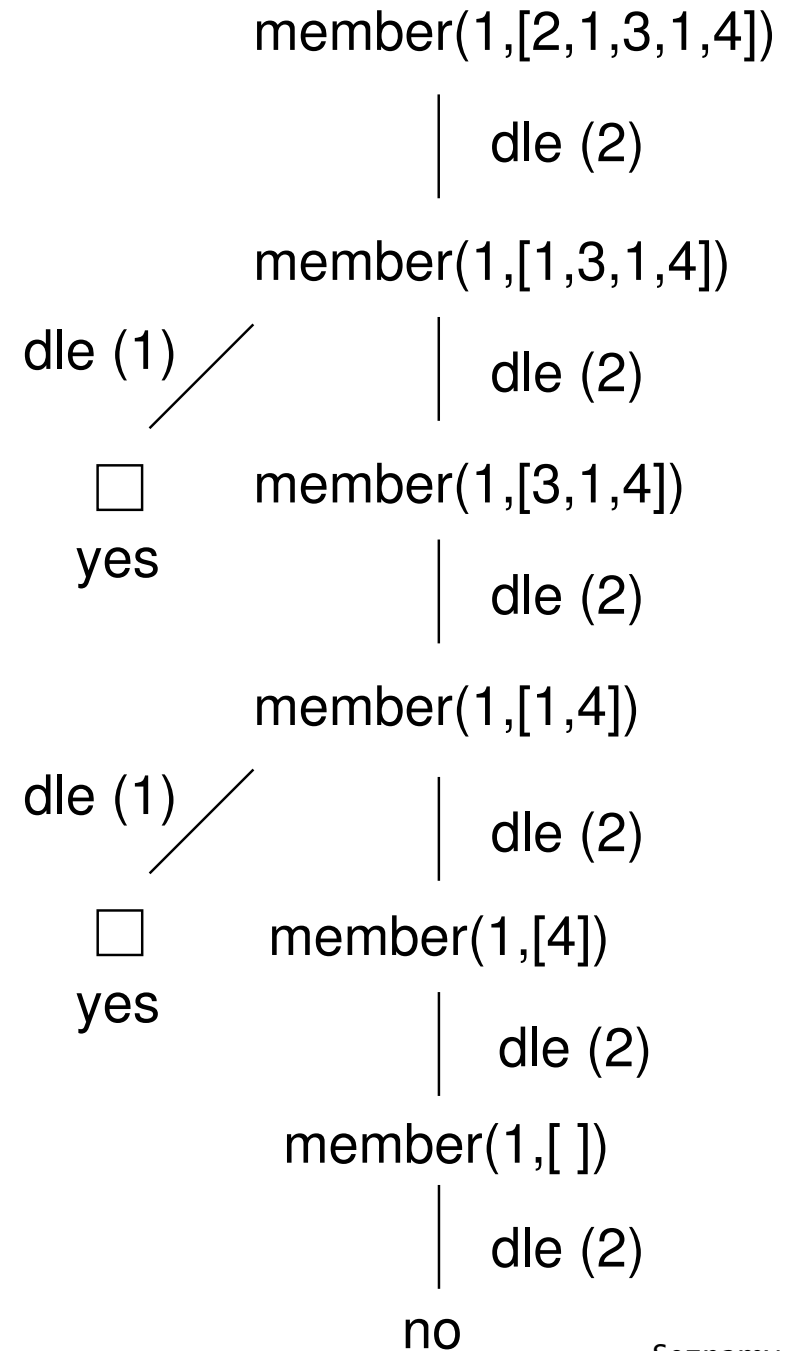
- X je prvek těla seznamu S

`member( X, [ _ | Telo ] ) :-  
member( X, Telo ).` %(2)



# 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  
`member( X, [ X | _ ] )`.   %(1)
  - X je prvek těla seznamu S  
`member( X, [ _ | Telo ] ) :-  
 member( X, Telo )`.   %(2)
- Příklady použití:
  - `member(1, [2,1,3])`.
  - `member(X, [1,2,3])`.



# 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] )`

# 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 )`.

# 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] )`

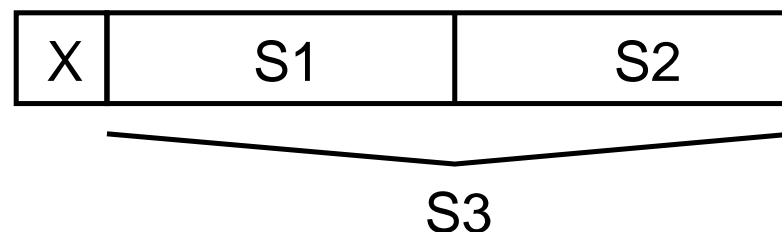
- Definice:

- 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 )`.



# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
  | (2)
```

```
  | A=[1|B]
```



# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
    | (2)
```

```
    | A=[1|B]
```

```
    |
```

```
:- append([2],[3,4],B).
```

# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
  | (2)
```

```
  | A=[1|B]
```

```
  |
```

```
:- append([2],[3,4],B).
```

```
  | (2)
```

```
  | B=[2|C] => A=[1,2|C]
```

# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
    | (2)
```

```
    | A=[1|B]
```

```
    |
```

```
:- append([2],[3,4],B).
```

```
    | (2)
```

```
    | B=[2|C] => A=[1,2|C]
```

```
    |
```

```
:- append([], [3,4], C).
```

# Cvičení: append/2

```
append( [], S, S ).      % (1)
```

```
append( [X|S1], S2, [X|S3] ) :- append( S1, S2, S3).  % (2)
```

```
:- append([1,2],[3,4],A).
```

```
  | (2)
```

```
  | A=[1|B]
```

```
  |
```

```
:- append([2],[3,4],B).
```

```
  | (2)
```

```
  | B=[2|C] => A=[1,2|C]
```

```
  |
```

```
:- append([], [3,4], C).
```

```
  | (1)
```

```
  | C=[3,4] => A=[1,2,3,4],
```

```
  |
```

```
yes
```