45. Prolog - způsob vyhodnocení (základní princip, unifikace, chování vestavěných predikátů, operátor řezu - vhodné a nevhodné užití).

Úvod k Prologu

Prolog je programovací jazyk, který je

- **deklarativní**
  - programátor popisuje pouze cíl výpočtu
  - přesný postup výpočtu si volí samotný systém
  - pouze abstraktní vyjádření faktů a vztahů mezi nimi
- **logický**
  - formální bází prologu je predikátová logika
  - Prolog omezuje predikátovou logiku na Hornovy klauzule
    - Disjunkce literárů, z nichž nejvíce jeden z nich je v pozitivní formě:
      - $b \vee \neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n$
    - což lze přepsat do typické formule implikačního tvaru jako:
      - $b \Leftarrow (a_1 \wedge a_2 \wedge \cdots \wedge a_n)$
  - při běhu využívá
    - **rekurzi**
    - **unifikaci**
    - **zpětné prohledávání** (backtracking)
- **vhodný pro úlohy založené na prohledávání stavového prostoru**

program - klauzule
dotazy - cíle

V prologu rozlišujeme

- **termy**
  - **konstanty**
    - *801*
    - *kote*
      - *Nesmí začínat velkým písmenem.*
  - **proměnné**
    - *Y, Xs, Promenna*
      - Musí začínat velkým písmenem.
    - _
      - anonymní proměnná
      - Zástupný znak vložený namísto proměnné, když nechceme s danou proměnnou pracovat (váže se na libovolnou proměnnou).

- **složené termy**
  - $tree(10, 20, leaf, leaf)$
    - funktor aplikovaný na argumenty
    - začíná malým písmenem

- **klauzule**
  - **pravidla**
    - $H :- B_1, B_2, \cdots, B_n.$
      - prologovský zápis pro Hornovu klauzuli $h \Leftarrow (b_1 \wedge b_2 \wedge \cdots \wedge b_n)$
      - zápis :– značí zpětnou implikaci
      - zápis , značí konjunkci
      - H je hlava
      - $B_1, B_2, \cdots, B_n$ jsou cíle
      - pravidlo musí být ukončeno tečkou
      - podmíněný predikát
        - pokud lze současně splnit veškeré predikáty v těle (splnit všechny cíle), lze splnit i hlavičku
  - **fakty**
    - $H.$
      - atom, fakt
      - pravidlo s prázdným tělem
      - ukončeno tečkou
      - odpovídá klauzuli $h \Leftarrow \top$
      - hlavička H vyplývá z tautologie
        - je tedy platným faktem

- **programy**
  - množiny predikátů
- **dotazy**
  - pravidla s prázdnou hlavou
  - tělo obsahuje jeden nebo více cílů

Vyhodnocování a unifikace

Dva termy v prologu jsou **unifikovatelné**, pokud je splněno alespoň jedno z následujícího:

- jsou identické
- obsahují proměnné, za něž je možné substituovat jiné termy tak, aby tyto termy nově identické byly

Unifikace se v prologu provádí

- explicitně v predikátu =
- při volání procedur zadáním cíle

Pro unifikaci platí:

- každá konstanta se unifikuje sama na sebe
  - *kote* = *kote*

        - **true.**
    - *801 = 801*
        - **true.**
- dvě různé konstanty se na sebe nikdy neunifikují
    - *kote = stene*
        - **false.**
    - *801 = 611*
        - **false.**
- každý funktor aplikovaný na konstanty se unifikuje sám se sebou
    - *mazlicek(kote) = mazlicek(kote)*
        - **true.**
- proměnná může být unifikovaná s konstantou
    - *X = kote*
        - **X = kote.**
        - termy *X*, *kote* je možné unifikovat, pokud se konstanta *kote* substituuje za *X*
- proměnná v těle funktoru může být unifikovná s konstantou v těle téhož funktoru s touž aritou na téže pozici
    - *mazlicek(X) = mazlicek(kote)*
        - **X = kote.**
    - *chova(X, kote) = chova(tomas, kote)*
        - **X = tomas.**
- proměnná může být unifikovaná s jinou proměnnou, pokud se obě unifikují se stejným konstantním termem
    - *X = kote, Y = kote*
        - **X = Y, Y = kote.**
- proměnná nemůže být unifikovaná současně na dva různé konstantní termy
    - *X = kote, X = stene*
        - **false.**
- funktory s různou aritou, s různým jménem, s různými konstantami na stejných pozicích na sebe nejsou unifikovatelné
    - *mazlicek(kote) = mazlicek(kote, stene)*
        - **false.**
    - *mazlicek(kote) = zviratko(kote)*
        - **false.**
    - *chova(tomas, X) = chova(kote, kote)*
        - **false.**

Unifikaci můžeme například použít:

- pro explicitní přiřazení do proměnné
    - *X = 801*
        - proměnná X se unifikuje s konstantou 801
- pro test na rovnost proměnných
    - *X = Y*

- pro práci se seznamy
  - *L = [H|_]*
    - první element seznamu L se unifikuje s proměnnou H
  - *L = [_|T]*
    - s proměnnou T se unifikuje seznam L bez prvního prvku
- předávání parametrů
  - máme např. definovaný predikát *elem(List, Index, Result)*, který vrátí element v zadaném seznamu *List* na zadaném indexu *Index* a výsledek unifikuje do proměnné *Result*
  - použitím *elem(L, 2, R)*
    - předáme parametr indexu konkrétní hodnotou 2
    - předáme parametr seznamu odkazem skrze proměnnou L

Mějme program v prologu, který obsahuje pravidla a dále fakty jako databázi informací. Sémantické vyhodnocování dotazů provádíme dle následujícího schématu:

- jednotlivé podcíle prohledáváme do hloubky
  - pokud máme například predikáty

```
positiveValues([]).
positiveValues([H|T]) :-
        positiveValues(T),
        H > 0.
```

dochází postupně k rekurzivnímu zanořování do predikátu *positiveValues(T)*, podmínka *H > 0* se vyhodnocuje až po vynoření se z rekurze

- výběr podcílů v klauzulích provádíme zleva doprava
  - pokud je predikát splněn, voláme následující v řadě
  - pokud je nesplněn, vracíme se zpět pomocí zpětného prohledávání
  - predikát může být vyhodnocován znovu s jiným přiřazením hodnot do proměnných
  - ve výše uvedeném příkladu je jako první prováděn podcíl *positiveValues(T)* uvedený jako první, následně podcíl *H > 0*
- databázi informací prohledáváme shora dolů
- pokud v dotazu ve formě predikátu nejsou žádné proměnné (případně pouze anonymní proměnné), snažíme se odvodit, zda lze predikát vyhodnotit jako pravdivý
  - vracíme pouze informaci *true*, nebo *false*
- pokud se v dotazu vyskytují proměnné (neanonymní), snažíme se najít takové termy, které by mohly tyto proměnné substituovat, aby s tímto přiřazením byl predikát vyhodnocen jako pravdivý
  - vracíme konkrétní zvolené přiřazení proměnných, nebo *false*
- v průběhu vyhodnocování dochází ke čtyřem různým operacím
  - **call**
    - první zavolání vyhodnocení predikátu
    - v takovém okamžiku je vytvořen takzvaný bod výběru (**choice point**), který je uložen do paměti
    - pro volaný predikát jsou do paměti poznamenána veškerá pravidla z programu, která současně

        - mají stejný název jako volaný predikát
        - mají stejný počet argumentů jako volaný predikát
    - tyto vzory jsou kandidáti, jejichž těla mohou být novým podcílem při následujícím vyhodnocování volaného predikátu
    - mezi těmito vzory je následně nalezen takový, který může být s aktuálním cílem unifikovaný a který je v kódu uvedený nejdříve
        - pokud takový podcíl existuje, je provedena
            - operace **exit**, pokud je tento podcíl faktem (pravidlem bez těla)
            - operace **call**, pokud je podcíl pravidlem s tělem obsahujícím další predikáty
        - pokud takový podcíl neexistuje, je provedena operace **fail**
    - ostatní kandidáti zaznamenáni v paměti v rámci bodu výběru mohou být použiti později v případě zpětného prohledávání (**backtracking**), pokud bude aktuální výběr pravidla vyhodnocen jako neúspěšný
- **exit**
    - úspěšné ukončení vyhodnocení predikátu
    - predikát byl vyhodnocený jako **true**
    - pokud je úspěšně vyhodnocený predikát součástí těla jiného dříve volaného predikátu, pak
        - je volán následující predikát v těle pomocí **call**, pokud takový existuje
        - vynoříme se z rekurze pomocí další operace **exit**, pokud je aktuálně vyhodnocený predikát poslední v těle predikátu na vyšší úrovni
- **fail**
    - neúspěšné ukončení vyhodnocení predikátu
    - predikát byl vyhodnocený jako **false**
    - při operaci **fail** je smazán veškerý pokrok provedený od posledního uložení bodu výběru a stav posledního bodu výběru je obnoven, existuje-li
    - vracíme se v čase zpět a hledáme poslední provedenou operaci **call** nebo **redo**, přičemž
        - pokračujeme dále v čase zpět, pokud při nalezeném volání **call** nebo **redo** již v rámci bodu výběru neexistují žádné další alternativy poznamenané v paměti
        - provedeme operaci **redo**, pokud v rámci nalezeného bodu výběru ještě existují nevyužité alternativy poznamenané v paměti
        - provedeme operaci **fail**, pokud se vynoříme o úroveň výše z volání nějakého predikátu
    - pokud žádný dříve vytvořený bod výběru neexistuje, původní volaný predikát je vyhodnocen jako **false**
- **redo**
    - volba jiné cesty při prohledávání stavového prostoru po předchozím neúspěchu

- jsme v situaci, kdy došlo k operaci **fail** a byl nalezen bod výběru, v němž zbývaly ještě nějaké alternativy poznamenané v paměti
  - tyto alternativy nemusí být nutně vůbec unifikovatelné s aktuálním podcílem, stačí pouze, aby nějaké alternativy stále existovaly
- začne opětovné vyhodnocování tohoto predikátu, tentokrát však odlišnou cestou

- *lze použít **operátor řezu** pro řízení běhu celého výpočtu jiným způsobem*

Příklad. Mějme predikát *ruzniSousede(List)*, který je vyhodnocený jako pravdivý tehdy, pokud je *List* seznam, kde žádné dva sousední elementy nejsou stejné. Definován je následovně:

```
ruzniSousede([ ]).
ruzniSousede([ _ ]).
ruzniSousede([A,B|T]) :-
        ruzniSousede([B|T]),
        A \= B.
```

Tento program

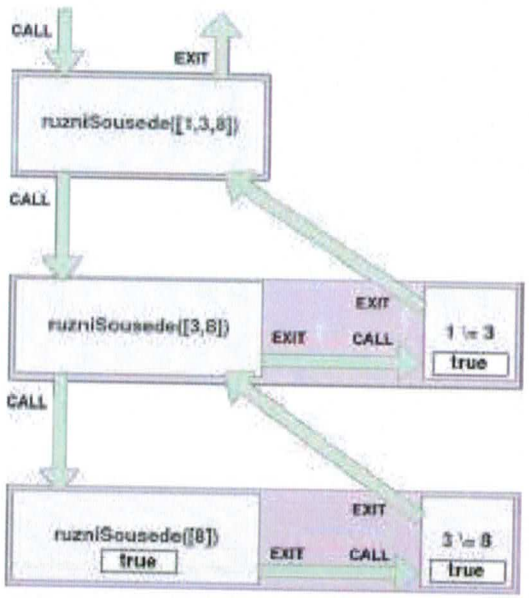
- obsahuje dva fakty
  - *ruzniSousede([ ]).*
    - pro prázdný seznam platí, že v něm nejsou dva stejné sousední prvky
  - *ruzniSousede([_]).*
    - pro jednoprvkový seznam platí, že v něm nejsou dva stejné sousední prvky
  - jde o pravdy uvedené v databázi
- obsahuje pravidlo, které není atomem
  - *ruzniSousede([A,B|T]) :-*
    ruzniSousede([B|T]),
    A \= B.
  - pokud jsou splněny klauzule *ruzniSousede([B|T])* a *A \= B*, pak je splněno *ruzniSousede([A,B|T])*
- používá unifikaci pro práci se seznamy
  - *ruzniSousede([B|T])*
    - s proměnnou *B* se unifikuje první element seznamu
    - s proměnnou *T* se unifikuje zbytek seznamu
  - *ruzniSousede([A,B|T])*
    - s proměnnou *A* se unifikuje první element seznamu
    - s proměnnou *B* se unifikuje druhý element seznamu
    - s proměnnou *T* se unifikuje zbytek seznamu

Jako cíl zvolme *ruzniSousede([1, 3, 8])*. Vyhodnocení probíhá následovně:

- **(call): ruzniSousede([1, 3, 8])**

- *ruzniSousede([1, 3, 8])* se unifikuje s hlavičkou *ruzniSousede([A,B|T])*, konkrétně: A = 1, B = 3, T = [8]
- v cíli je predikát *ruzniSousede([1, 3, 8])* rozdělen na podcíle *ruzniSousede([3|[8]])*, *1 \= 3* dle těla tohoto predikátu
- jelikož prohledáváme do hloubky a vyhodnocujeme podcíle zleva doprava, volá se *ruzniSousede([3|[8]])*
- **(call): ruzniSousede([3, 8])**
  - *ruzniSousede([3, 8])* se unifikuje s hlavičkou *ruzniSousede([A,B|T])*, konkrétně:
    A = 3, B = 8, T = [ ]
  - v cíli je predikát *ruzniSousede([3, 8])* rozdělen na podcíle *ruzniSousede([8|[ ]])*,
    3 \= 8 dle těla tohoto predikátu
  - jelikož prohledáváme do hloubky a vyhodnocujeme podcíle zleva doprava, volá se *ruzniSousede([8|[ ]])*
  - **(call): ruzniSousede([8])**
    - *ruzniSousede([8|[ ]])* se unifikuje s hlavičkou *ruzniSousede([_])*, což je fakt z naší databáze, tedy aktuální podcíl je pravdivý
    - ukončujeme vyhodnocování *ruzniSousede([8|[ ]])* jako pravdivé
    - **(exit): ruzniSousede([8])**
  - predikát *ruzniSousede([8])* byl vyhodnocen úspěšně jako **true**, pokračujeme s dalším podcílem na této úrovni, konkrétně *3 \= 8*
  - **(call): 3 \= 8**
    - tento podcíl je vyhodnocen jako pravdivý, ukončujeme jeho vyhodnocení
    - **(exit): 3 \= 8**
  - všechny predikáty v těle predikátu *ruzniSousede([3, 8])* byly vyhodnoceny jako pravdivé, takže i tento predikát vyhodnocujeme jako pravdivý
  - **(exit): ruzniSousede([3, 8])**
- predikát *ruzniSousede([3, 8])* byl vyhodnocen úspěšně jako **true**, pokračujeme s dalším podcílem na této úrovni, konkrétně *1 \= 3*
- **(call): 1 \= 3**
  - tento podcíl je vyhodnocen jako pravdivý, ukončujeme jeho vyhodnocení
  - **(exit): 1 \= 3**
- všechny predikáty v těle predikátu *ruzniSousede([1, 3, 8])* byly vyhodnoceny jako pravdivé, takže i tento predikát vyhodnocujeme jako pravdivý
- **(exit): ruzniSousede([1, 3, 8])**

- vrací se příznak **true**

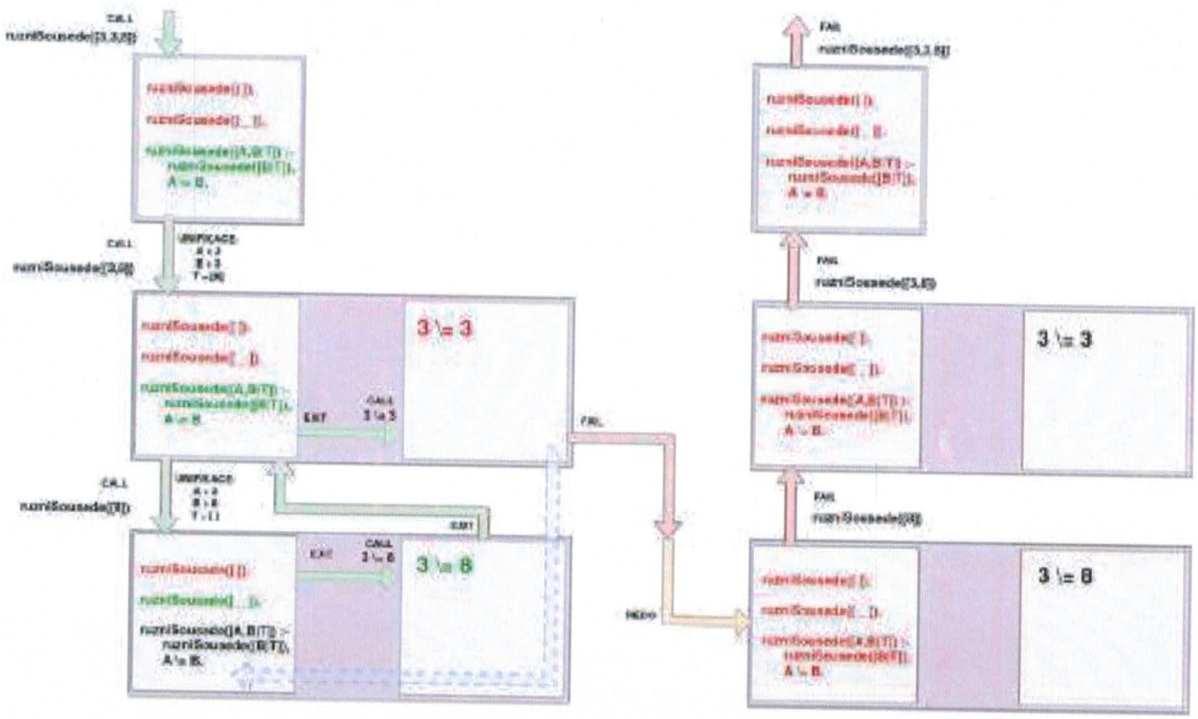


Pokud ale jako cíl zvolíme *ruzniSousede([3, 3, 8])*, vyhodnocení probíhá následovně:

- **(call)**: **ruzniSousede([3, 3, 8])**
  - *ruzniSousede([3, 3, 8])* se unifikuje s hlavičkou *ruzniSousede([A,B|T])*, konkrétně: A = 3, B = 3, T = [8]
  - v cíli je predikát *ruzniSousede([3, 3, 8])* rozdělen na podcíle *ruzniSousede([3|[8]])*, *3 \= 3* dle těla tohoto predikátu
  - jelikož prohledáváme do hloubky a vyhodnocujeme podcíle zleva doprava, volá se *ruzniSousede([3|[8]])*
  - **(call)**: **ruzniSousede([3, 8])**
    - *ruzniSousede([3, 8])* se unifikuje s hlavičkou *ruzniSousede([A,B|T])*, konkrétně:
      A = 3, B = 8, T = [ ]
    - v cíli je predikát *ruzniSousede([3, 8])* rozdělen na podcíle *ruzniSousede([8|[ ]])*,
      3 \= 8 dle těla tohoto predikátu
    - jelikož prohledáváme do hloubky a vyhodnocujeme podcíle zleva doprava, volá se *ruzniSousede([8|[ ]])*
    - **(call)**: **ruzniSousede([8])**
      - *ruzniSousede([8|[ ]])* se unifikuje s hlavičkou *ruzniSousede([_])*, což je fakt z naší databáze, tedy aktuální podcíl je pravdivý
      - ukončujeme vyhodnocování *ruzniSousede([8|[ ]])* jako pravdivé
      - **(exit)**: **ruzniSousede([8])**
    - predikát *ruzniSousede([8])* byl vyhodnocen úspěšně jako **true**, pokračujeme s dalším podcílem na této úrovni, konkrétně *3 \= 8*
    - **(call)**: **3 \= 8**
      - tento podcíl je vyhodnocen jako pravdivý, ukončujeme jeho vyhodnocení

- **(exit)**: **3 \= 8**
    - všechny predikáty v těle predikátu *ruzniSousede([3, 8])* byly vyhodnoceny jako pravdivé, takže i tento predikát vyhodnocujeme jako pravdivý
    - **(exit)**: **ruzniSousede([3, 8])**
- predikát *ruzniSousede([3, 8])* byl vyhodnocen úspěšně jako **true**, pokračujeme s dalším podcílem na této úrovni, konkrétně *3 \= 3*
- **(call)**: **3 \= 3**
    - tento podcíl je vyhodnocen jako nepravdivý, ukončujeme jeho vyhodnocení
    - **(fail)**: **3 \= 3**
- **vracíme se zpět,** hledáme poslední takové použití call nebo redo, pro které platí, že je možné pro tento predikát využít některou další dosud nepoužitou cestu
    - V bodě výběru volání (call): ruzniSousede([8]) existuje nepoužitá alternativa *ruzniSousede([A,B|T])* s tělem *ruzniSousede([B|T]), A \= B*
    - provádíme tedy (redo): ruzniSousede([8])
- **(redo): ruzniSousede([8])**
    - unifikace s jedinou dosud nepoužitou alternativou *ruzniSousede([A,B|T])* není možná, ukončujeme vyhodnocení
    - **(fail)**: **ruzniSousede([8])**
- **vracíme se zpět,** hledáme poslední takové použití call nebo redo, pro které platí, že je možné pro tento predikát využít některou další dosud nepoužitou cestu
    - u volání (call): ruzniSousede([3, 8]) na vyšší úrovni už neexistuje další nepoužitá alternativa
    - ukončujeme neúspěšně vyhodnocení ruzniSousede([3, 8])
    - **(fail)**: **ruzniSousede([3, 8])**

- **vracíme se zpět,** hledáme poslední takové použití call nebo redo, pro které platí, že je možné pro tento predikát využít některou další dosud nepoužitou cestu
    - u volání (call): ruzniSousede([3, 3, 8]) na vyšší úrovni už neexistuje další nepoužitá alternativa
    - ukončujeme neúspěšně vyhodnocení ruzniSousede([3, 3, 8])
    - **(fail)**: **ruzniSousede([3, 3, 8])**
- vrací se příznak **false**



Chování vestavěných predikátů

Prolog pro praktické účely zahrnuje řadu vestavěných predikátů, které typicky

- mohou být mimologické
  - nejsou založeny na predikátové logice
- slouží k provedení vedlejších účinků
  - vstupně-výstupní operace
  - práce se souborem
- nemusí být znovu splnitelné
  - nelze je použít v rámci zpětného prohledávání
    - např. provádění zpětného prohledávání při práci se souborem nedává smysl

Typicky využíváme vestavěné predikáty

- **is**
  - vestavěný predikát sloužící k vyhodnocování aritmetických výrazů
  - syntaxe
    - *proměnná* **is** *výraz*
    - *(číselná konstanta* **is** *výraz)*
  - výrazy v prologu jsou obyčejné nevyhodnocené termy bez významu (neinterpretované termy)
  - vyhodnoceny mohou být teprve tehdy, objeví-li se na pravé straně operátoru *is*
    - *is* vynucuje vyhodnocení
  - použití *X* **is** *výraz*

- Prolog vyhodnotí výraz a následně se pokusí výsledek unifikovat s proměnnou vlevo (X)

Y is X + 1

- nejprve je rozhodnuto, zda *výraz* je skutečně term odpovídající validnímu aritmetickému výrazu
  - pokud není, operátor *is* selže
- následně se prolog pokusí *výraz* vyhodnotit
  - pokud se *výraz* nepovede vyhodnotit, operátor *is* selže
  - validní výraz se nemusí povést vyhodnotit například v případě, jakmile obsahuje volnou proměnnou
- dále se prolog pokusí vyhodnocený výsledek unifikovat s proměnnou *X*
  - unifikace může selhat
- *na levé straně se může vyskytovat i číselná konstanta, ovšem v takovém případě je vhodnější použití operátoru =:= na test rovnosti výrazů*
  - *2 is 1 + 1* uspěje
  - *2 =:= 1 + 1* uspěje
  - *2.0 is 1 + 1* selže
  - *2.0 =:= 1 + 1* uspěje

- *is* není totéž jako přiřazení do proměnné v imperativních jazycích
  - *X = X + 1* v imperativních jazycích
    - zvýší hodnotu proměnné *X* o *1*, pokud tato proměnná existuje
  - *X is X + 1* v prologu
    - selže, pokud je *X* volná proměnná, neboť výraz *X + 1* nelze vyhodnotit
    - selže, pokud je *X* vázaná proměnná, neboť se nepovede unifikovat *X* s *X + 1*
- *is* funguje **pouze jednosměrně**, není možné dopočítat hodnotu proměnné na pravé straně operátoru *is* na základě konstanty na levé straně
  - *10 is 8 + X*
    - selže, pokud je *X* volná proměnná, neboť výraz *8 + X* nelze vyhodnotit
- **příklad**
  - *X = 201 + 17, Y = X - 4, Z is Y + 1, W = Z * 6*
  - po vyhodnocení získáme
    - *X = 201 + 17*
      - *201 + 17* je pouze nevyhodnocený term
    - *Y = 201 + 17 - 4*
      - *201 + 17 - 4* je pouze nevyhodnocený term
      - vznikl přidáním členu *-4* za term unifikovaný s *X*
    - *Z = 215*
      - teprve nyní došlo díky operátoru *is* k vyhodnocení výrazu
      - *Y + 1* je validní vyhodnotitelný výraz
      - *201 + 17 - 4 + 1 = 215*, pročež *Z* je unifikováno s *215*
    - *W = 215 * 6*
      - *215 * 6* je pouze nevyhodnocený term
      - vycházíme ovšem z hodnoty proměnné *Z*, s níž je už unifikovaný výsledek dříve uvedeného výrazu

- **var**
  - vestavěný predikát, který je splněn, pokud má na vstupu volnou proměnnou
  - **příklad**
    - var(X)
      - **true**
    - X = 81, var(X)
      - **false**

- **nonvar**
  - vestavěný predikát, který je splněn, pokud má na vstupu vázanou proměnnou
  - **příklad**
    - nonvar(X)
      - **false**
    - X = 81, nonvar(X)
      - **X = 81**

- **=..**
  - vestavěný predikát, který rozbaluje term do seznamu a naopak
  - **syntaxe**
    - *term* **=..** *seznam*
  - pokud je senzam jednoprvkový, je odpovídající term tato jediná položka seznamu
    - musí jít o validní term
  - pokud je seznam víceprvkový, je odpovídající term složený term (funktor aplikovaný na argumenty)
    - první položka seznamu je daný funktor nebo proměnná unifikovaná s daným funktorem
    - zbylé položky seznamu jsou argumenty funktoru
  - obdobným způsobem se termy uvedené na levé straně operátoru rozbalují do seznamu
  - **příklad**
    - *X =.. [10]*
      - **X = 10**
    - *X =.. [append]*
      - **X = append**
    - *X =.. [node, 10, 20, leaf, leaf]*
      - **X = node(10, 20, leaf, leaf)**
    - *X =.. [odd, 4]*
      - **X = odd(4)**
    - *Y = odd, X =.. [Y, 4]*
      - **Y = odd**
      - **X = odd(4)**
    - *X =.. [4, odd]*
      - **chyba,** 4 není funktor ani proměnná unifikovaná s funktorem

        - *X =.. [is, Y, 4]*
            - **X = (Y is 4)**
        - *801 =.. X*
            - **X = [801]**
        - *leaf =.. X*
            - **X = [leaf]**
        - *positive(10) =.. X*
            - **X = [positive, 10]**
        - *complexNum(1, 1) =.. X*
            - **X = [complexNum, 1, 1]**
        - *(X =.. odd(8)) =.. Y*
            - **Y = [=.., X, odd(8)]**

- **call**
    - vestavěný predikát, který **zavolá** predikát, na nějž je aplikován
    - **call(P)** má tentýž význam, jako kdyby byl predikát **P** na místě zapsán
    - lze kombinovat s operátorem **=..** a na místě volat predikáty vytvořené z položek seznamu
    - **příklad**
        - Y =.. [is, X, 801 - 10], call(Y)
            - **Y = (X is 801 - 10)**
            - **X = 791**
        - call(var(X))
            - **true**
        - Y =.. [append, [801, 802], [804], Res], call(Y)
            - **Y = append([801, 802], [804], [801, 802, 804])**
            - **Res = [801, 802, 804]**

- **read**
    - čtení z aktuálně nastaveného vstupního proudu
- **write**
    - zápis do aktuálně nastaveného výstupního proudu
- **fail**
    - predikát, který vždy selže
- **true**
    - predikát, který vždy uspěje
- **repeat**
    - predikát, který je vždy možné znovu splnit
- **not**

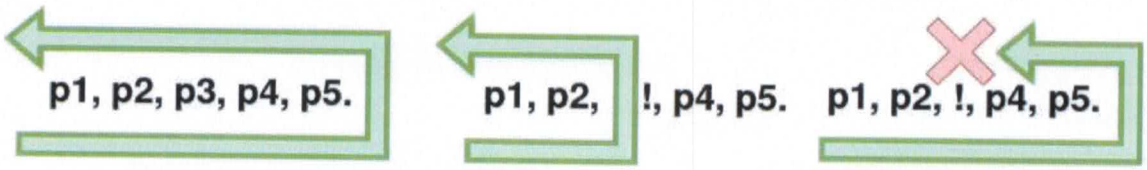
- predikát, který mění úspěch na neúspěch a naopak
- **definice**
  - *not(P) :- call(P), !, fail.*
  - *not(P).*

Operátor řezu

Operátor řezu **!** v prologu nám umožňuje řídit metodu vyhodnocování predikátů tím způsobem, že od místa svého použití zabraňuje zpětnému prohledávání.

Mějme predikát p s tělem $p_1, p_2, \ldots, p_k, !, p_{(k+1)}, \ldots, p_n$, přičemž v těle tohoto predikátu se nachází jediný výskyt operátoru řezu *!*. Potom:

- v případě **selhání** některého z predikátů $p_1, p_2, \ldots, p_k$ dochází ke zpětnému navracení obvyklým způsobem
- v případě **splnění** predikátu p_k přejdeme přes operátor řezu *!* vždy automaticky zleva doprava, jako kdyby šlo o splněný predikát
- v případě **selhání** některého z predikátů $p_{(k+1)}, \ldots, p_n$ dochází ke zpětnému navracení, ovšem přes operátor řezu *!* se není možné vrátit zpět doleva
  - všechny body navracení vytvořené predikáty $p_1, p_2, \ldots, p_k$ jsou přechodem přes operátor řezu *!* zrušeny
  - pokud při zpětném navracení narazíme na operátor řezu, ukončíme zpětné navracení na této úrovni a vynoříme se v prohledávání o úroveň výše



Operátor řezu je typicky **vhodné** použít v následujících případech:

- chceme zamezit, aby byla použita jiná než nalezená varianta pravidla
  - v rámci prohledávání nalezneme variantu predikátu, s níž se podcíl unifikuje
  - pokud bychom hledali jiné řešení, mohla by být použita jiná varianta, což je v některých případech nežádoucí
  - **příklad**
    - *factorial(0, 1) :- !.*
      factorial(N, F) :-
      N1 is N-1,
      factorial(N1, F1),
      *F is N * F1.*
    - predikát pro výpočet faktoriálu
    - jakmile jednou použijeme variantu *factorial(0, 1)*, není možné provádět zpětné navracení na této úrovni
    - nelze tedy nikdy unifikovat *0* s *N* v druhé variantě predikátu a zacyklit se tak, neboť zpětné prohledávání nám nebude umožněno – jakmile jednou

projdeme přes *factorial(0, 1) :- !.* v dané úrovni, už se nikdy pro tuto úroveň nepokusíme o unifikaci s *factorial(N, F).*
        - k zacyklení by mohlo dojít v případě, kdy bychom se snažili najít alternativní řešení k vypočtené hodnotě faktoriálu a hodnota *0* by byla unifikována s *N*
- chceme vynutit okamžité selhání: *term1, …, termn, !, fail.*
- chceme ukončit generování alternativních řešení
    - v rámci prohledávání může prolog nabídnout více různých řešení
    - může se stát, že smysl dává pouze první řešení a zbylá jsou absurdní, případně že nám stačí jediné řešení
    - pomocí operátoru řezu lze zařídit, aby se alternativní řešení nenabízela
    - **příklad**
        - *myInt(0).*
          myInt(X) :- myInt(Y), X is Y + 1.
        - *lcm(A, B, Res) :-*
          myInt(Res),
          Res >= A,
          Res >= B,
          0 is (Res mod A),
          0 is (Res mod B),
          !.
        - predikát *myInt(X)* je generátorem přirozených čísel od nuly, přičemž při zpětném navracení generuje vždy následující přirozené číslo
        - predikát *lcm(A, B, Res)* nalézá nejmenší společný násobek čísel *A, B* a unifikuje jej s proměnnou *Res*
        - pokud bychom nepoužili operátor řezu *!*, nabízel by predikát *lcm(A, B, Res)* další alternativy po nalezení nejmenšího společného násobku
            - generoval by veškeré další společné násobky *A, B*
        - jakmile je nejmenší společný násobek nalezen, přejdeme přes operátor řezu zleva doprava, čímž zamezíme zpětnému navracení a žádná další řešení nebudou nabídnuta

Ukázka **nevhodného** použití operátoru řezu:

- při použití konstant/proměnných v predikátu jiným než očekávaným způsobem
    - definujeme predikát, v němž na jednom místě očekáváme proměnnou, do níž se unifikuje výsledek
    - v predikátu používáme operátor řezu, který zamezuje dalšímu prohledávání a nabízení nesmyslných výsledků
    - při volání cíle ovšem v predikátu na místě výsledku použijeme konstantu (nikoliv proměnnou), která může být prohlášena za validní výsledek, ačkoliv jde o výsledek nesmyslný
    - **příklad**
        - *myInt(0).*
          myInt(X) :- myInt(Y), X is Y + 1.
        - *lcm(A, B, Res) :-*

myInt(Res),
Res >= A,
Res >= B,
0 is (Res mod A),
0 is (Res mod B),
!.

- predikát *lcm(A, B, Res)* pro výpočet nejmenšího společného násobku očekává jako třetí argument proměnnou, do níž se unifikuje nalezený nejmenší společný násobek
- pokud ovšem místo *Res* použijeme konstantu, která je libovolným společným násobkem *A*, *B*, bude predikát vyhodnocen jako **true**, ačkoliv nemusí jít o nejmenší společný násobek
  - *lcm(24, 9, 72)*
    - **true**
    - jde o nejmenší společný násobek
  - *lcm(24, 9, 216)*
    - **true**
    - **nejde** o nejmenší společný násobek, ovšem jde o společný násobek
    - pro vstup *24, 9, 216* je zjištěno, že
      - *216* je přirozené číslo
      - *216 >= 24, 216 >= 9*
      - *216* je dělitelné *24* i *9*
    - všechny podmínky jsou splněny, proto je cíl prohlášen za pravdivý
- operátor řezu zde nezabránil prohlášení nesprávného výsledku za správný

- nevhodné použití operátoru řezu v případě, kdy reálně chceme najít všechna možná řešení
- potlačení chybových hlášení
  - pokud použijeme operátor řezu k potlačení chybových hlášení/výjimek, je možné, že zbytek programu poběží nesprávným způsobem
  - **příklad**
    - *divide(X, 0, Res) :- !, fail.*
    - *divide(X, Y, Res) :-*
      Y > 0,
      Res is X / Y.
    - dělení nulou nezpůsobí výjimku, ale vynoření o úroveň výše, což může vést na neočekávané chování

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele kocotom.

FLP - Operátor řezu

Vhodné použití

```
remove(A, [A|T], T) :- !.
remove(A, [B|T1], [B|T2]) :- remove(A, T1, T2).
remove(A, [], []).
```

Nevhodné použití

```
parents(adam, 0) :- !.
parents(eva, 0) :- !.
parents(X, 2).
```

ale §parents(eva, 2) bude vyhodnoceno na true.