46. Prolog - změna DB/programu za běhu (demonstrace na prohledávání stavového prostoru, práce se seznamy).

Práce se seznamy

second([X,Y|_],Y).

V Prologu lze použít:

- prázdný seznam **[ ]**
- neprázdný seznam **[H | T]**
  - proměnná **H** je unifikovaná s prvním prvkem seznamu
  - proměnná **T** je unifikovaná se zbytkem seznamu bez prvního prvku
- *v neprázdném seznamu lze vynutit minimální konkrétní počet elementů pomocí zápisu **[H1, H2, H3, ..., Hn | T]***
  - *seznam obsahující minimálně n elementů H1, H2, H3, ..., Hn*

3 typy seznamů: normální, diferenční a funkcionální

Definujme několik predikátů pro práci se seznamy:

- **nalezení prvního prvku**
  - **implementace**
    - $head([H|\_], H).$
  - **vysvětlení**
    - proměnná H se unifikuje s prvním elementem seznamu pomocí zápisu $[H|\_]$
    - unifikuje se i s výstupem H
    - pokud je vstupní seznam prázdný, predikát není splněn
  - prakticky je zbytečné definovat podobný predikát, neboť pokud máme k dispozici seznam unifikovaný s proměnnou $List$, stačí pouze provést unifikaci $List = [H|\_]$ a první prvek si takto vyčíst
- **nalezení posledního prvku**
  - **implementace**

$$last([Res], Res) :- !.$$
$$last([\_|T], Res) :- last(T, Res).$$

  - **vysvětlení**
    - pokud je v seznamu jediný prvek, je poslední
    - pokud je v seznamu více prvků, spouštíme predikát rekurzivně pro vstupní seznam bez prvního prvku, jehož podoba nás nezajímá
    - pokud je vstupní seznam prázdný, predikát není splněn
    - pokud bychom neuvedli operátor řezu v rámci klauzule $last([Res], Res) :- !.$, pak po nalezení posledního prvku by bylo možné pokračovat

append(L1,L2,L12) def jako.
append([],L2,L2).
append([X|T],L2,[X|Z]) :- append(T,L2,Z).

reverse(L1, L2) def jako
reverse([], []).
reverse([X|T], Z) :- reverse(T, Y), append(Y, [X], Z).

alternativní cestou pro jednoprvkový seznam skrze hlavičku $last([\_|T], Res)$, kdy unifikujeme $T = [\,]$ a voláme predikát $last([\,], Res)$, který vždy selže

- **test, zda se element nachází v seznamu** member(X, L)
  - **implementace**

$$member(X, [X|\_]) :- !.$$
$$member(X, [\_|T]) :- member(X, T).$$

  - **vysvětlení**
    - pokud je hledaný prvek první v seznamu, pak je nalezen
      - pokud je hledaný prvek nalezen, pomocí operátoru řezu zrušíme všechny ostatní neprozkoumané výpočetní větve, protože nemá smysl pátrat po alternativách
    - jinak je predikát spuštěn rekurzivně pro seznam bez prvního prvku
- **test, zda je seznam množinou**
  - testujeme, zda je každý prvek seznamu unikátní
  - **implementace**

$$isSet([\,]).$$
$$isSet([H|T]) :- not(member(H, T)), isSet(T).$$

  - **vysvětlení**
    - $isSet([\,])$. říká, že prázdná množina je množina
    - v rámci volání $isSet([H|T])$ ověříme, zda se prvek H nachází ve zbytku seznamu T
    - pokud ano, selžeme, jinak pokračujeme rekurzivním voláním predikátu pro zbytek seznamu T
    - zde není nutné použít operátor řezu, neboť hlavičky predikátů jsou vzájemně výlučné (žádný predikát není možné současně unifikovat s oběma z nich)
- **hledání podmnožin**
  - mějme predikát *mySubset(Set, Res)*, který pro vstupní množinu *Set* do proměnné *Res* postupně naunifikuje veškeré její podmnožiny
  - předpokládejme, že proměnná *Set* nikdy neobsahuje duplicity
  - nyní nás nezajímá, že seznam je uspořádaný a množina ne
  - **implementace**

$$mySubset([\,], [\,]).$$
$$mySubset([H|T], [H|X]) :- mySubset(T, X).$$
$$mySubset([H|T], X) :- mySubset(T, X).$$

  - **vysvětlení**
    - $mySubset([\,], [\,])$. je fakt, který říká, že prázdná množina je podmnožinou prázdné množiny

bagof(+Var, +Goal, -Bag)
- do Bag nám unifikuje seznam možných hodnot proměnné +Var v rámci lokálního podcíle +Goal

Př: bagof(C, A^B^foo(A,B,C), CS).

- predikát $mySubset([H|T],[H|X])$ do výsledné podmnožiny dává první prvek vstupního seznamu a spouští se rekurzivně bez prvního prvku
- predikát $mySubset([H|T],X)$ do výsledné podmnožiny nedává první prvek vstupního seznamu a spouští se rekurzivně bez prvního prvku
- při prohledávání skrze všechny možné cesty jsou nalezené veškeré možné podmnožiny

A, B ignoruj

Změna databáze za běhu

Prolog je jeden z mála dosud živých jazyků, jež umožňují za běhu měnit vlastní program. Program v prologu se skládá z databáze klauzulí, které se dále dělí na:

program se skládá z množiny klauzulí (Hornových klauzulí)

- **pravidla**
  - $H :- B_1, B_2, \cdots, B_n.$
    - podmíněný predikát
    - skládá se z hlavičky H a těla $B_1, B_2, \cdots, B_n$
    - hlavička je splněna, pokud je splněno tělo
- **fakty**
  - $H.$
    - platný fakt
    - skládá se pouze z hlavičky s prázdným tělem
    - prázdné tělo je triviálně splněno, tedy hlavička platí

Jednotlivé predikáty, jež jsou v rámci programu popsány pomocí klauzulí, se dle možné modifikovatelnosti dělí na:

- **statické**
  - předem pevně stanoveny
  - za běhu programu se neočekávají změny jejich definic
  - za běhu programu se neočekává vytváření nových alternativ pro tyto predikáty
- **dynamické**
  - modifikovatelné za běhu pomocí predikátů **assert, asserta, assertz, retract, retractall**
  - lze vytvořit novou klauzuli, která bude pro odpovídající dynamický predikát další alternativou při vyhodnocování
  - lze odstranit existující klauzuli, která pro odpovídající dynamický predikát již nově alternativou při vyhodnocování nebude
  - lze ovlivnit, zda se nově vytvořená klauzule bude nacházet před dosud existujícími klauzulemi, nebo za nimi
  - predikát je dynamický, pokud tuto skutečnost deklarujeme pomocí klíčového slova **dynamic**
  - **syntaxe**
    - $:- dynamic\ název/arita$
    - je třeba uvést
      - název predikátu

nejsem si jistej, zda to zmiňoval, je to i implementation dependant

        - aritu predikátu (očekávaný počet argumentů)
    - uvádíme na začátku programu
- **příklad**
    - $:-dynamic\ mazlicek/1$
        - predikát $mazlicek$ očekávající jeden argument je nyní dynamický
    - $:-dynamic\ edge/2$
        - predikát $edge$ očekávající dva argumenty je nyní dynamický

Mějme predikát P s aritou n, jenž je v programu deklarován jako dynamický. Potom lze použít následující predikáty:

- **asserta**
    - **syntaxe**
        - $asserta\big(P(a_1, a_2, \cdots, a_n)\big).$
        - $asserta\big(P(a_1, a_2, \cdots, a_n) :-(b_1, b_2, \cdots, b_k)\big).$
    - zavolání tohoto predikátu způsobí, že se **na začátek** prologovského programu (databáze) vloží fakt $P(a_1, a_2, \cdots, a_n).$, případně pravidlo $P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k.$
    - od tohoto okamžiku je součástí databáze a může ovlivnit další vyhodnocování
    - pokud některá z hodnot v hlavičce byla vyjádřena pomocí volné proměnné, pak za ni lze v dotazech úspěšně dosadit libovolný term
    - úspěšné provedení $asserta$ je vyhodnoceno jako **true**
    - do programu je možné přidat klauzuli identickou s již existující klauzulí, v takovém případě bude při vyhodnocování prolog potenciálně postupovat dvakrát touž cestou
    - **příklad**
        - mějme dynamický predikát definovaný jako $:-dynamic\ chova/2.$
        - mějme následující program:
            - $chova(teta, kote).$
            - $chova(teta, tele).$
        - proveďme nyní tuto sekvenci dotazů
            - $?-chova(teta, X).$
                - $X = kote;$
                - $X = tele.$
            - $?-asserta(chova(teta, sele)).$
                - $true.$
            - $?-chova(teta, X).$
                - $X = sele;$
                - $X = kote;$
                - $X = tele.$
            - $?-asserta(chova(deda, X)).$
                - $true.$
            - $?-chova(deda, dite).$
                - $true.$

- $?-chova(deda, 801).$
  - $true.$
- $?-chova(X, Y).$
  - $X = deda;$
  - $X = teta,$
    $Y = sele;$
  - $X = teta,$
    $Y = kote;$
  - $X = teta,$
    $Y = tele.$

- **assertz**
  - **syntaxe**
    - $assertz(P(a_1, a_2, \cdots, a_n)).$
    - $assertz(P(a_1, a_2, \cdots, a_n) :- (b_1, b_2, \cdots, b_k)).$
  - zavolání tohoto predikátu způsobí, že se **na konec** prologovského programu (databáze) vloží fakt $P(a_1, a_2, \cdots, a_n).$, případně pravidlo $P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k.$
  - od tohoto okamžiku je součástí databáze a může ovlivnit další vyhodnocování
  - pokud některá z hodnot v hlavičce byla vyjádřena pomocí volné proměnné, pak za ni lze v dotazech úspěšně dosadit libovolný term
  - úspěšné provedení $assertz$ je vyhodnoceno jako **true**
  - do programu je možné přidat klauzuli identickou s již existující klauzulí, v takovém případě bude při vyhodnocování prolog potenciálně postupovat dvakrát touž cestou
  - **příklad**
    - mějme dynamický predikát definovaný jako $:-dynamic\ chova/2.$
    - mějme následující program:
      - $chova(teta, kote).$
      - $chova(teta, tele).$
    - proveďme nyní tuto sekvenci dotazů
      - $?-chova(teta, X).$
        - $X = kote;$
        - $X = tele.$
      - $?-assertz(chova(teta, sele)).$
        - $true.$
      - $?-chova(teta, X).$
        - $X = kote;$
        - $X = tele;$
        - $X = sele.$

- **assert**
  - **syntaxe**
    - $assert(P(a_1, a_2, \cdots, a_n)).$
    - $assert(P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k).$

- zavolání tohoto predikátu způsobí, že se do prologovského programu (databáze) vloží fakt $P(a_1, a_2, \cdots, a_n)$., případně pravidlo $P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k$.
- od tohoto okamžiku je součástí databáze a může ovlivnit další vyhodnocování
- assert obecně **negarantuje** konkrétní místo vložení daného predikátu do databáze
  - v rámci SWI-Prolog funguje $assert$ stejně jako $assertz$
  - ISO standard Prologu toto všem negarantuje

- **retract**
  - **syntaxe**
    - $retract(P(a_1, a_2, \cdots, a_n))$.
    - $retract(P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k)$.
  - zavolání tohoto predikátu způsobí, že se z prologovského programu (databáze) **odstraní první výskyt** faktu $P(a_1, a_2, \cdots, a_n)$, případně pravidla $P(a_1, a_2, \cdots, a_n) :- b_1, b_2, \cdots, b_k$, pokud se v programu vyskytuje
  - pokud se daná klauzule v programu nenachází, vrací predikát **false.**
  - pokud se daná klauzule v programu nachází a při unifikaci cíle s odstraňovanou položkou v programu nedochází k substituci za žádnou proměnnou, vrací predikát **true.**
  - jinak vrací informaci o unifikovaných proměnných
  - pokud se cíl uvedený v predikátu $retract$ unifikuje s vyšším množstvím klauzulí v programu, postupně v rámci zpětného prohledávání může shora dolů odstranit všechny tyto výskyty
  - **příklad**
    - mějme dynamický predikát definovaný jako : $-dynamic\ chova/2$.
    - mějme následující program:
      - $chova(teta, kote)$.
      - $chova(teta, tele)$.
      - $chova(deda, kote)$.
      - $chova(deda, sele)$.
      - $chova(teta, jehne)$.
      - $chova(teta, jehne)$.
    - proveďme nyní tuto sekvenci dotazů
      - $?-chova(deda, X)$.
        - $X = kote$;
        - $X = sele$.
      - $?-retract(chova(deda, tele))$.
        - $false$.
      - $?-chova(deda, X)$.
        - $X = kote$;
        - $X = sele$.
      - $?-retract(chova(deda, sele))$.
        - $true$.
      - $?-chova(deda, X)$.
        - $X = kote$.
      - $?-retract(chova(X, kote))$.

        - $X = teta;$
        - $X = deda.$
    - $?-chova(X, kote).$
        - $false.$
    - $?-chova(teta, jehne).$
        - $true;$
        - $true.$
    - $?-retract(chova(teta, jehne)).$
        - $true;$
        - $true.$
    - $?-chova(teta, jehne).$
        - $false.$

- **retractall**
    - **syntaxe**
        - $retractall(P(a_1, a_2, \cdots, a_n)).$
    - zavolání tohoto predikátu způsobí, že se z prologovského programu (databáze) **odstraní** veškeré výskyty klauzulí, jejichž hlavičku lze unifikovat s $P(a_1, a_2, \cdots, a_n)$
    - **příklad**
        - mějme dynamický predikát definovaný jako : $-dynamic\ chova/2.$
        - mějme následující program:
            - $chova(teta, kote).$
            - $chova(teta, tele).$
            - $chova(deda, kote).$
            - $chova(deda, sele).$
            - $chova(teta, jehne).$
            - $chova(teta, jehne).$
        - proveďme nyní tuto sekvenci dotazů
            - $?-retractall(chova(teta, X)).$
                - $true.$
            - $?-chova(X, Y).$
                - $X = deda,$
                  $Y = kote;$
                - $X = deda,$
                  $Y = sele.$
            - $?-retractall(chova(X, Y)).$
                - $true.$
            - $?-chova(X, Y).$
                - $false.$

Pozn.: Použijeme-li některý z predikátů ***asserta, assertz, assert, retract, retractall*** *na statický predikát (predikát, který nebyl explicitně deklarován jako dynamický pomocí klíčového slova* ***dynamic****), jenž byl definován ve zdrojovém kódu, dojde k běhové chybě. Pokud v konzoli*

pracujeme s novými predikáty, jež se ve zdrojovém kódu neobjevily, chovají se jako dynamické i bez předchozí deklarace.

Pozn.: Účinky predikátů ***asserta, assertz, assert, retract, retractall se při zpětném navracení neanulují!*** *Pokud se přes ně budeme vracet zpět v rámci zpětného prohledávání při hledání jiné výpočetní větve, jejich efekt přidání prvku do databáze/odstranění prvku z databáze není zvrácen.*

Způsoby aktualizace databáze

Při změnách databáze v Prologu existují dvě základní strategie, jež popisují, v jakém okamžiku jsou přidané/odstraněné klauzule viditelné zbytku programu. Konkrétně jde o:

- **okamžitou aktualizaci pohledu**
  - pokud dochází po změně databáze ke zpětnému navracení, změna je okamžitě viditelná i předchozím predikátům volaným před změnou databáze
  - dnes téměř nepoužívané
  - **příklad**
    - mějme dynamický predikát vytvořený jako : $-dynamic\ mocniny/1$
    - proveďme nyní tuto sekvenci dotazů
      - $?-assertz(mocniny(1)).$
        - $true.$
      - $?-mocniny(A).$
        - $A = 1.$
      - $?-mocniny(A), B\ is\ A * 2, assertz(mocniny(B)).$
        - $A = 1,$ $B = 2;$
        - $A = 2,$ $B = 4;$
        - $A = 4,$ $B = 8;$
        - $A = 8,$ $B = 16;$
        - program nabízí alternativy donekonečna
    - v rámci dotazů jsme do databáze vložili fakt $mocniny(1)$
    - při volání cíle $?-mocniny(A), B\ is\ A * 2, assertz(mocniny(B))$ tedy dojde k následujícímu chování
      - vzhledem k existenci faktu $mocniny(1)$ dochází k unifikaci $A = 1$
      - vestavěný predikát is vyhodnotí výraz $A * 2$ jako 2 a provede unifikaci $B = 2$
      - $assertz(mocniny(B))$ do databáze vloží fakt $mocniny(2)$
      - při stiskutí středníku dochází ke zpětnému prohledávání
        - $assertz(mocniny(B))$ je ignorováno
        - $B\ is\ A * 2$ je ignorováno
        - $mocniny(A)$ **nachází další alternativu a provede unifikaci** $A = 2$

        - **vložení nové hodnoty do databáze se projevilo ihned a je možné danou hodnotu okamžitě využít při zpětném prohledávání**
    - podobným způsobem se pokračuje donekonečna, databáze je stále plněna mocninami čísla 2
- **logickou aktualizaci pohledu**
    - pokud dochází po změně databáze ke zpětnému navracení, změna databáze nemusí být okamžitě viditelná předchozím predikátům
    - při každé změně databáze dochází k inkrementaci hodnoty **generace dabatáze**
    - **každý podcíl** je označen identifikátorem generace databáze, v níž byl zahájen
    - **každá klauzule** v databázi je označena identifikátorem generace databáze, v níž byla vytvořena
    - pokud je klauzule odstraněna, je navíc označena identifikátorem generace databáze, v níž byla odstraněna
    - **klauzule v databázi je z podcíle viditelná pouze tehdy, pokud je podcíl identifikován generací databáze, jež se nachází v intervalu** $< created, erased >$ **dané klauzule**
    - logická aktualizace pohledu je součástí ISO standardu Prologu, v současnosti je používaná ve většině implementací
    - **příklad**
        - mějme dynamický predikát vytvořený jako : $-dynamic\ mocniny/1$
        - proveďme nyní tuto sekvenci dotazů
            - $?-assertz(mocniny(1)).$
                - $true.$
            - $?-mocniny(A).$
                - $A = 1.$
            - $?-mocniny(A), B\ is\ A * 2, assertz(mocniny(B)).$
                - $A = 1,$
                  $B = 2;$
            - $?-mocniny(A).$
                - $A = 1;$
                - $A = 2;$
            - $?-mocniny(A), B\ is\ A * 2, assertz(mocniny(B)).$
                - $A = 1,$
                  $B = 2;$
                - $A = 2,$
                  $B = 4;$
            - $?-mocniny(A).$
                - $A = 1;$
                - $A = 2;$
                - $A = 2;$
                - $A = 4;$
            - v rámci dotazů jsme do databáze vložili fakt $mocniny(1)$

- při volání cíle $?-mocniny(A), B\ is\ A*2, assertz(mocniny(B))$ tedy dojde k následujícímu chování
  - vzhledem k existenci faktu $mocniny(1)$ dochází k unifikaci $A=1$
  - vestavěný predikát is vyhodnotí výraz $A*2$ jako 2 a provede unifikaci $B=2$
  - $assertz(mocniny(B))$ do databáze vloží fakt $mocniny(2)$
  - při stiskutí středníku dochází ke zpětnému prohledávání
    - $assertz(mocniny(B))$ je ignorováno
    - $B\ is\ A*2$ je ignorováno
    - $mocniny(A)$ **nevede na žádnou unifikaci, neboť v tomto bodu výběru již neexistuje žádná alternativa**
    - **fakt** $mocniny(2)$ **byl do databáze vložen v pozdější generaci, než v níž byl volán cíl** $mocniny(A)$
  - při dalším volání $?-mocniny(A), B\ is\ A*2, assertz(mocniny(B))$. je již fakt $mocniny(2)$ viditelný
  - do databáze je podruhé vloženo $mocniny(2)$ a nově i $mocniny(4)$

Demonstrace změny databáze na prohledávání stavového prostoru

Mějme následující program v prologu:

- $:-dynamic\ edge/2.$
- $:-dynamic\ used/1.$
- $states(1,9).$
- $edge(1,2).$
- $edge(1,4).$
- $edge(2,5).$
- $edge(3,5).$
- $edge(4,5).$
- $edge(4,7).$
- $edge(4,8).$
- $edge(5,1).$
- $edge(6,8).$
- $edge(6,9).$
- $edge(7,1).$
- $edge(8,7).$
- $edge(9,6).$

Tento soubor faktů definuje graf, přičemž

- $states(1,9)$ značí, že používáme stavy očíslované hodnotami od 1 do 9
- $edge(A,B)$ značí, že ze stavu A vede hrana do stavu B
- $:-dynamic\ edge/2$, $:-dynamic\ used/1$ značí, že predikáty $edge, used$ s uvedenými aritami jsou dynamické

Definujme nyní predikát *path(Start, End, Path)*, který pro zadané uzly *Start*, *End* do proměnné *Path* unifikuje cestu z uzlu *Start* do uzlu *End*, pokud existuje.

- $path(Start, Start, [Start]) :- !.$
- $path(Start, End, [Start|Rest]) :-$
  - $assertz(used(Start))$,
  - $edge(Start, Next)$,
  - $not(used(Next))$,
  - $path(Next, End, Rest)$.
- $path(Start, \_, \_) :-$
  - $used(Start)$,
  - $retract(used(Start))$,
  - $fail$.

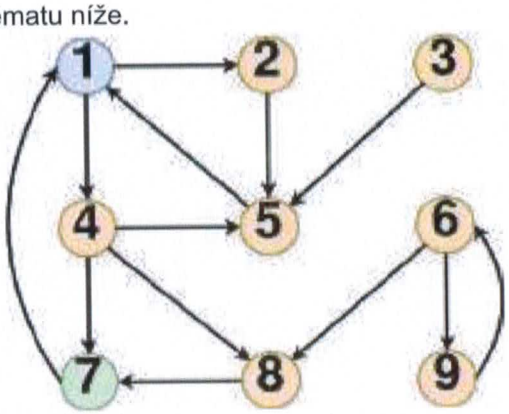
Vysvětlení:

- pokud se počáteční uzel $Start$ shoduje s cílovým uzlem, končíme a vracíme cestu $[Start]$
- jinak hledáme delší cestu
- do databáze si pomocí $assertz(used(Start))$ poznamenáme, že uzel $Start$ byl již navštíven, tedy nechceme přes něj vést zbytek cesty, neboť cesta nemůže obsahovat cyklus
- pomocí volání $edge(Start, Next)$ do proměnné $Next$ naunifikujeme takový uzel, do nějž lze jednokrokově přejít z uzlu $Start$
- ověříme, že uzel $Next$ není již zablokován, neboť jsme přes něj v minulosti již procházeli
  - pokud ano, zahájíme zpětné navracení a vybereme jiný uzel $Next$, do nějž lze jednokrokově přejít z uzlu $Start$
  - pokud žádný vhodný náhradní uzel $Next$ neexistuje, použijeme třetí alternativu predikátu s hlavičkou $path(Start, \_, \_)$, která z databáze smaže $used(Start)$, čímž vyjadřuje, že jsme se vydali špatnou cestou
  - pomocí predikátu $fail$ zaručíme, že bude vykonáno zpětné navracení zabezpečující jiné pokračování vyhodnocení
- pokud je uzel $Next$ dosud nepoužitý, pokračujeme v prohledávání z něj
- **tento predikát postupně najde veškeré cesty z uzlu $Start$ do uzlu End**
- **pokud vyhledáme veškeré cesty, máme garanci, že databáze použitých uzlů bude na konci zcela smazaná**
- **pokud nevyhledáme všechny cesty a ukončíme vyhodnocování dříve, zůstanou v databázi některé klauzule $used(A)$, kde A je uzel nalezené cesty**
- **pokud před dalším separátním spuštěním predikátu $path$ tuto databázi nesmažeme pomocí $retractall(used(X))$, bude hledání cest fungovat obecně nesprávně**

Příklad.

- Uvažujme o grafu uvedeném výše a o výše popsané implementaci predikátu $path$
- Zavoláme-li predikát $path(1, 7, Path)$, pak vyhodnocování proběhne následovně:
  - vydáme se cestou $1 \rightarrow 2 \rightarrow 5$, která bude odhalena jako nesprávná
  - vracíme se zpět do uzlu 1 a pokračujeme cestou $1 \rightarrow 4 \rightarrow 5$, která bude odhalena jako nesprávná
  - vracíme se zpět do uzlu 4 a pokračujeme cestou $4 \rightarrow 7$, jež vede do hledaného cíle
  - unifikujeme $Path = [1,4,7]$
  - při hledání alternativní cesty se vracíme do uzlu 4 a pokračujeme cestou $4 \rightarrow 8 \rightarrow 7$, která vede do hledaného cíle
  - unifikujeme $Path = [1,4,8,7]$
  - při hledání alternativní cesty už není žádná další cesta nalezena
    - při neúspěšném hledání dalších cest vymažeme veškeré hodnoty $used(X)$ z databáze

Graf je vyobrazený na schématu níže.



Uveďme nyní ještě několik predikátů pracujících s uvedenou grafovou reprezentací, jež zahrnují manipulaci s dynamickými predikáty.

- **odstranění smyček z grafu**
  - **implementace**
    - $removeLoops :-$
      - $edge(X, X)$,
      - $retract(edge(X, X))$,
      - $fail$.
    - $removeLoops$.
  - **vysvětlení**
    - pomocí predikátu $edge(X, X)$ naunifikujeme do proměnné X takový uzel, jenž má v grafu smyčku
    - pomocí $retract(edge(X, X))$ tuto smyčku odstraníme z databáze
    - pomocí $fail$ zahájíme zpětné prohledávání, tedy naunifikujeme do proměnné X další uzel se smyčkou a pokračujeme odstraněním smyčky

- **při zpětném prohledávání přes predikát** $retract(edge(X,X))$ **není předchozí účinek tohoto predikátu anulován**
- jakmile v grafu neexistuje další smyčka, nepovede se unifikace $edge(X,X)$ a v rámci zpětného prohledávání se přesuneme k faktu $removeLoops.$, čímž odstraňování smyček úspěšně skončí

- **symetrický uzávěr grafu**
  - pokud v grafu existuje hrana $edge(A,B)$, bude v symetrickém uzávěru přítomna i hrana $edge(B,A)$
  - **implementace**
    - $symmetricClosure :-$
      - $edge(A,B),$
      - $not(edge(B,A)),$
      - $assertz(edge(B,A)),$
      - $fail.$
    - $symmetricClosure.$
  - **vysvětlení**
    - pomocí predikátu $edge(A,B)$ naunifikujeme do proměnných A,B uzly tak, že existuje hrana z uzlu A do uzlu B
    - pomocí predikátu $not(edge(B,A))$ zkontrolujeme, zda se opačná hrana již v grafu nenachází
      - pokud ano, nebudeme ji do grafu přidávat a vracíme se k unifikaci další hrany
    - pomocí $assertz(edge(B,A))$ přidáme zpětnou hranu do databáze
    - pomocí $fail$ zahájíme zpětné prohledávání, tedy naunifikujeme do proměnných A, B další incidentní uzly
      - **při zpětném prohledávání přes predikát** $assertz(edge(B,A))$ **není předchozí účinek tohoto predikátu anulován**
    - jakmile v grafu neexistuje další neprozkoumaná hrana, nepovede se unifikace $edge(A,B)$ a v rámci zpětného prohledávání se přesuneme k faktu$symmetricClosure.$, čímž přidávání hran do grafu skončí
    - **jelikož pracujeme s logickou aktualizací pohledu, pomocí** $edge(A,B)$ **při zpětném prohledávání nenaunifikujeme žádnou hranu, kterou jsme v rámci výpočtu symterického uzávěru do grafu přidali**
    - **první volání** $edge(A,B)$ **proběhlo v rámci první generace databáze a další volání REDO pro tento predikát stále probíhá v rámci téže generace databáze**
    - **přidávání nových hran do grafu ovšem generace databáze zvyšuje, tyto hrany tedy nejsou v první generaci viditelné**

- **tranzitivní uzávěr grafu**
  - pokud v grafu existuje cesta z uzlu A do uzlu B o alespoň jedné hraně, pak v tranzitivním uzávěru grafu bude přítomna i hrana $edge(A,B)$
  - **implementace**

- $transitiveClosure :-$
  - $states(Min, Max)$,
  - $between(Min, Max, \_)$,
  - $edge(A, B)$,
  - $edge(B, C)$,
  - $not(edge(A, C))$,
  - $assertz(edge(A, C))$,
  - $fail$.
- $transitiveClosure$.

- **vysvětlení**
  - pomocí predikátu $states(Min, Max)$ najdeme identifikátory stavů s nejnižším a nejvyšším identifikátorem
    - zde předpokládáme, že jde o konečnou, nepřerušovanou posloupnost přirozených čísel
  - **pomocí predikátu** $between(Min, Max, \_)$ **zajistíme, že v rámci zpětného prohledávání se následující úsek těla predikátu** $transitiveClosure$ **provede tolikrát, kolik je v grafu uzlů**
    - $between(A, B, C)$ postupně unifikuje s proměnnou C veškerá přirozená čísla z intervalu $< A, B >$
    - my zde konkrétní hodnotu C zahodíme, máme ale jistotu, že $between(Min, Max, \_)$ bude splněno právě tolikrát, kolik stavů existuje
  - pomocí predikátů $edge(A, B)$, $edge(B, C)$ naunifikujeme do proměnných A, B, C takové uzly, aby platilo, že existuje hrana z A do B a hrana z B do C
  - pomocí predikátu $not(edge(A, C))$ ověříme, zda se hrana z uzlu A do uzlu C v grafu již nenachází
    - pokud ano, vracíme se pomocí zpětného prohledávání zpět a hledáme další vhodné hrany
  - jinak pomocí predikátu $assertz(edge(A, C))$ přidáme hranu z uzlu A do uzlu C do databáze
  - pomocí $fail$ zahájíme zpětné prohledávání, tedy naunifikujeme do proměnných A, B, C další vhodné uzly
  - po provedení celého procesu bude predikát úspěšně vyhodnocen díky alternativě $transitiveClosure$., jež je faktem
  - **pokud bychom na začátek těla** $transitiveClosure$ **neuvedli predikáty**$states(Min, Max)$, $between(Min, Max, \_)$, **nebyl by výsledkem skutečný tranzitivní uzávěr grafu**
  - **používáme logickou aktualizaci pohledu, tedy opakované znovusplňování predikátu** $edge(A, B)$ **by probíhalo stále ve stejné generaci databáze jako při prvním volání tohoto predikátu, nebralo by tudíž zřetel na nově přidané hrany do grafu**
  - **jakmile** $edge(A, B)$ **poprvé neuspěje, vracíme se skrze zpětné prohledávání k predikátu** $between(Min, Max, \_)$, **který najde další přirozené číslo na intervalu** $< Min, Max >$

- **pokud uspěje, je predikát $edge(A, B)$ volán znovu od začátku, tentokrát již v aktuální generaci databáze, v níž jsou viditelné dříve přidané hrany do grafu**
- výsledek je správný tranzitivní uzávěr grafu, neboť celý proces stačí spustit právě tolikrát, kolik v grafu existuje uzlů *(viz Floyd-Warshallův algoritmus)*

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

FLP - Prolog - DB

- assert, asserta, assertz
- retract

Příklady:

```
is_integer(0).
is_integer(X) :- is_integer(Y), X is Y+1.

range(Start, End, _) :- Start > End, !, fail.
range(Start, End, X) :- X = Start.
range(Start, End, X) :- NStart is Start+1, range(NStart, End, X).
```

Příklad např. pohyb koněm ze start(X,Y) na end(EX,EY)

```
solve(start(X,Y), end(X,Y)) :- !, assert(pos(X,Y)).
solve(start(X,Y), end(EX,EY)) :-
assert(pos(X,Y)),
move(X,Y,XX,YY),
not(pos(XX,YY)),
solve(start(XX,YY), end(EX,EY)), !.
```

```
solve(start(X,Y), _) :-
    pos(X,Y),
    retract(pos(X,Y)),
    fail.
```

Případně print výsledku:

```
print_result :-
    pos(X,Y),
    write(X), write(','), write(Y),
    nl,
    fail.
print_result.
```