

# Implementace Prologu

## Literatura:

- Matyska L., Toman D.: Implementační techniky Prologu, Informační systémy, (1990), 21–59.  
<http://www.ics.muni.cz/people/matyska/vyuka/1p/1p.html>

# Opakování: základní pojmy

- Konečná množina klauzulí  $H$ lava :- Tělo tvoří **program P**.
- **Hlava** je literál
- **Tělo** je (eventuálně prázdná) konjunkce literálů  $T_1, \dots, T_a, a \geq 0$
- **Literál**  
je tvořen  $m$ -árním predikátovým symbolem  $(m/p)$  a  $m$  termy (argumenty)
- **Term** je konstanta, proměnná nebo složený term.
- **Složený term**  
s  $n$  termy na místě argumentů
- **Dotaz (cíl)** je neprázdná množina literálů.

# Interpretace

## Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

# Interpretace

## Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

## Procedurální (imperativní) sémantika:

Entry: Hlava::

```
{  
    call  $T_1$   
    :  
    call  $T_a$   
}
```

Volání procedury s názvem Hlava uspěje, pokud uspěje volání všech procedur (literálů) v těle.

# Interpretace

## Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

## Procedurální (imperativní) sémantika:

Entry: Hlava::

```
{  
    call  $T_1$   
    :  
    call  $T_a$   
}
```

Volání procedury s názvem Hlava uspěje, pokud uspěje volání všech procedur (literálů) v těle.

## Procedurální sémantika = podklad pro implementaci

# Abstraktní interpret

Vstup: Logický program  $P$  a dotaz  $G$ .

1. Inicializuj množinu cílů  $S$  literály z dotazu  $G$ ;  $S := G$
2. `while ( S != empty ) do`
3. Vyber  $A \in S$  a dále vyber klauzuli  $A' : -B_1, \dots, B_n$  ( $n \geq 0$ ) z programu  $P$  takovou, že  $\exists \sigma : A\sigma = A'\sigma$ ;  $\sigma$  je nejobecnější unifikátor.  
Pokud neexistuje  $A'$  nebo  $\sigma$ , ukonči cyklus.

# Abstraktní interpret

Vstup: Logický program  $P$  a dotaz  $G$ .

1. Inicializuj množinu cílů  $S$  literály z dotazu  $G$ ;  $S := G$
2. `while ( S != empty )` do
3. Vyber  $A \in S$  a dále vyber klauzuli  $A' : -B_1, \dots, B_n$  ( $n \geq 0$ ) z programu  $P$  takovou, že  $\exists \sigma : A\sigma = A'\sigma$ ;  $\sigma$  je nejobecnější unifikátor.  
Pokud neexistuje  $A'$  nebo  $\sigma$ , ukonči cyklus.
4. Nahrad'  $A$  v  $S$  cíli  $B_1$  až  $B_n$ .
5. Aplikuj  $\sigma$  na  $G$  a  $S$ .
6. `end while`
7. Pokud  $S == \text{empty}$ , pak výpočet úspěšně skončil a výstupem je  $G$  se všemi aplikovanými substitucemi.  
Pokud  $S \neq \text{empty}$ , výpočet končí neúspěchem.

# Abstraktní interpret – pokračování

Kroky (3) až (5) představují **redukci** (logickou inferenci) cíle A.

Počet redukcí za sekundu (LIPS) == indikátor výkonu implementace



# Abstraktní interpret – pokračování

Kroky (3) až (5) představují **redukci** (logickou inferenci) cíle A.

Počet redukcí za sekundu (LIPS) == indikátor výkonu implementace

## Věta

Existuje-li instance  $G'$  dotazu  $G$ , odvoditelná z programu  $P$  v konečném počtu kroků, pak bude tímto interpretem nalezena.

# Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle  $A$  z množiny cílů  $S$

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule  $A'$  z programu  $P$

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

# Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle  $A$  z množiny cílů  $S$

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule  $A'$  z programu  $P$

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

## Vztah k úplnosti:

1. Selekční pravidlo neovlivňuje úplnost

● možno zvolit libovolné v rámci SLD rezoluce

2. Prohledávání stromu výpočtu do šířky nebo do hloubky

# Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle  $A$  z množiny cílů  $S$

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule  $A'$  z programu  $P$

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

## Vztah k úplnosti:

1. Selekční pravidlo neovlivňuje úplnost

● možno zvolit libovolné v rámci SLD rezoluce

2. Prohledávání stromu výpočtu do šířky nebo do hloubky

„Prozření” – automatický výběr správné klauzule

● vlastnost abstraktního interpretu, kterou ale reálné interprety nemají

# Prohledávání do šířky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ 
  - necht' je těchto klauzulí  $q$
2. Vytvoříme  $q$  kopií množiny  $S$
3. V každé kopii redukuje  $A$  jednou z klauzulí  $A'_i$ .
  - aplikujeme příslušný nejobecnější unifikátor

# Prohledávání do šířky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ 
  - necht' je těchto klauzulí  $q$
2. Vytvoříme  $q$  kopií množiny  $S$
3. V každé kopii redukuje  $A$  jednou z klauzulí  $A'_i$ .
  - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukuje všechny množiny  $S_i$  současně.

# Prohledávání do šířky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ 
  - necht' je těchto klauzulí  $q$
2. Vytvoříme  $q$  kopií množiny  $S$
3. V každé kopii redukuje  $A$  jednou z klauzulí  $A'_i$ .
  - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukuje všechny množiny  $S_i$  současně.
5. Výpočet ukončíme úspěchem, pokud se alespoň jedna z množin  $S_i$  stane prázdnou.

# Prohledávání do šířky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$

- necht' je těchto klauzulí  $q$

2. Vytvoříme  $q$  kopií množiny  $S$

3. V každé kopii redukuje  $A$  jednou z klauzulí  $A'_i$ .

- aplikujeme příslušný nejobecnější unifikátor

4. V následujících krocích redukuje všechny množiny  $S_i$  současně.

5. Výpočet ukončíme úspěchem, pokud se alespoň jedna z množin  $S_i$  stane prázdnou.

- Ekvivalence s abstraktnímu interpretem

- pokud jeden interpret neuspěje, pak neuspěje i druhý

- pokud jeden interpret uspěje, pak uspěje i druhý



# Prohledávání do hloubky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem A.
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.

# Prohledávání do hloubky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli  $A'$ , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru  $\sigma$ ) a vybereme ze zásobníku další klauzuli.

# Prohledávání do hloubky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli  $A'$ , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru  $\sigma$ ) a vybereme ze zásobníku další klauzuli.
5. Pokud je zásobník prázdný, končí výpočet neúspěchem.
6. Pokud naopak zredukujeme všechny literály v  $S$ , výpočet končí úspěchem.

# Prohledávání do hloubky

1. Vybereme všechny klauzule  $A'_i$ , které je možno unifikovat s literálem  $A$ .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli  $A'$ , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru  $\sigma$ ) a vybereme ze zásobníku další klauzuli.
5. Pokud je zásobník prázdný, končí výpočet neúspěchem.
6. Pokud naopak zredukujeme všechny literály v  $S$ , výpočet končí úspěchem.

- Není úplné, tj. nemusí najít všechna řešení
- Nižší paměťová náročnost než prohledávání do šířky
- Používá se v Prologu

# Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

# Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

## Typy objektů

- **Primitivní objekty:**
  - konstanta
  - číslo
  - volná proměnná
  - odkaz (reference)

# Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

## Typy objektů

### ● Primitivní objekty:

- konstanta
- číslo
- volná proměnná
- odkaz (reference)

### ● Složené (strukturované) objekty:

- struktura
- seznam

# Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT



# Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

# Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Primitivní objekty uloženy přímo ve slově

# Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Primitivní objekty uloženy přímo ve slově

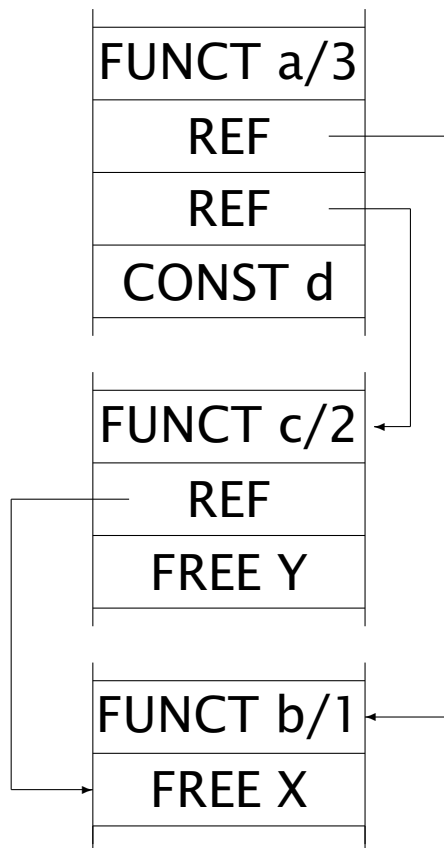
Složené objekty

- jsou instance termu ve zdrojovém textu, tzv. zdrojového termu
- zdrojový term bez proměnných  $\Rightarrow$  každá instancie ekvivalentní zdrojovému termu
- zdrojový term s proměnnými  $\Rightarrow$  dvě instance se mohou lišit aktuálními hodnotami proměnných, jedinečnost zajišťuje kopírování struktur nebo sdílení struktur

# Kopírování struktur

Příklad:

$a(b(X), c(X, Y), d)$ ,

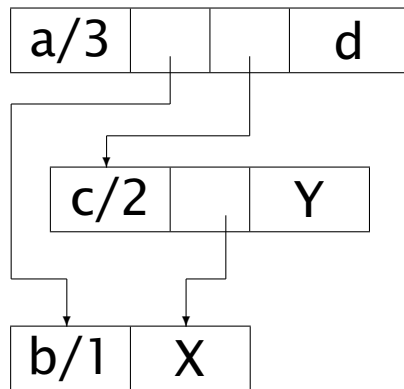


# Kopírování struktur II

● Term F s aritou A reprezentován A+1 slovy:

- funktor a arita v prvním slově
- 2. slovo nese první argument (resp. odkaz na jeho hodnotu) :
- A+1 slovo nese hodnotu A-tého argumentu

● Reprezentace vychází z orientovaných acyklických grafů:



● Vykopírována každá instance ⇒ **kopírování struktur**

● Termy ukládány na **globální zásobník**

# Sdílení struktur

- Vychází z myšlenky, že při reprezentaci je třeba řešit přítomnost proměnných
- Instance termu

< kostra\_termu; rámeč >

- kostra\_termu je zdrojový term s očíslovanými proměnnými
- rámeč je vektor aktuálních hodnot těchto proměnných
  - $i$ -tá položka nese hodnotu  $i$ -té proměnné v původním termu

# Sdílení struktur II

Příklad:

$a(b(X), c(X, Y), d)$

reprezentuje

$\langle a(b(\$1), c(\$1, \$2), d) ; [\text{FREE}, \text{FREE}] \rangle$

kde symbolem  $\$i$  označujeme  $i$ -tou proměnnou.

# Sdílení struktur II

Příklad:

`a(b(X), c(X, Y), d)`

reprezentuje

`< a(b($1), c($1, $2), d) ; [FREE, FREE] >`

kde symbolem `$i` označujeme  $i$ -tou proměnnou.

## Implementace:

`< &kostra_termu; &rámec >` (& vrací adresu objektu)

Všechny instance sdílí společnou `kostru_termu` ⇒ **sdílení struktur**



# Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné

# Srovnání: příklad

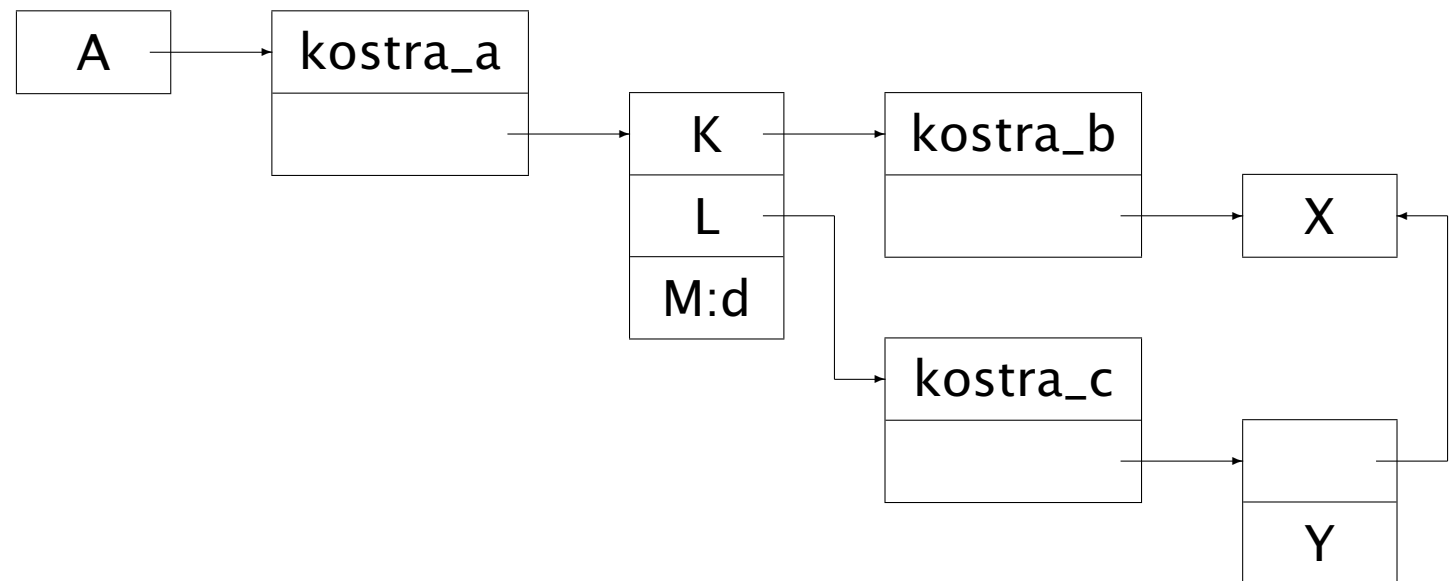
- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu

# Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu
- Postupná tvorba termů:

$A = a(K, L, M)$ ,  $K = b(X)$ ,  $L = c(X, Y)$ ,  $M = d$

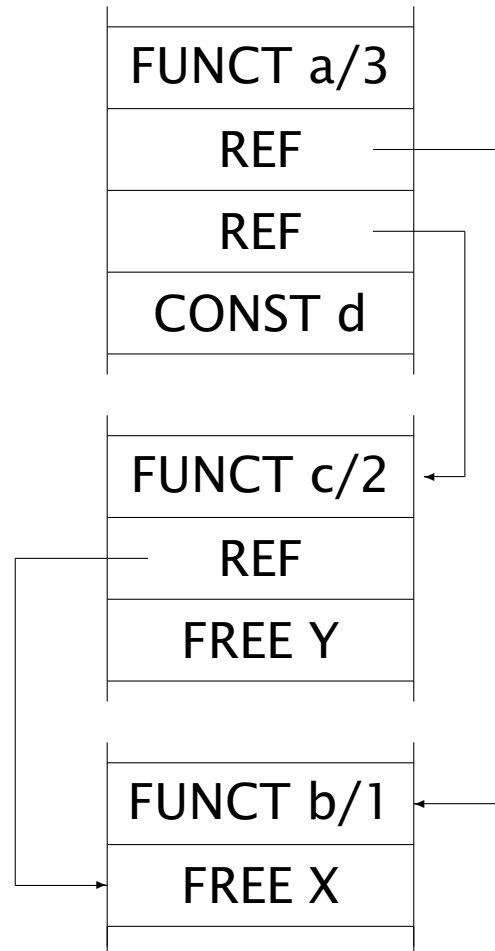
- Sdílení termů:



# Srovnání: příklad – pokračování

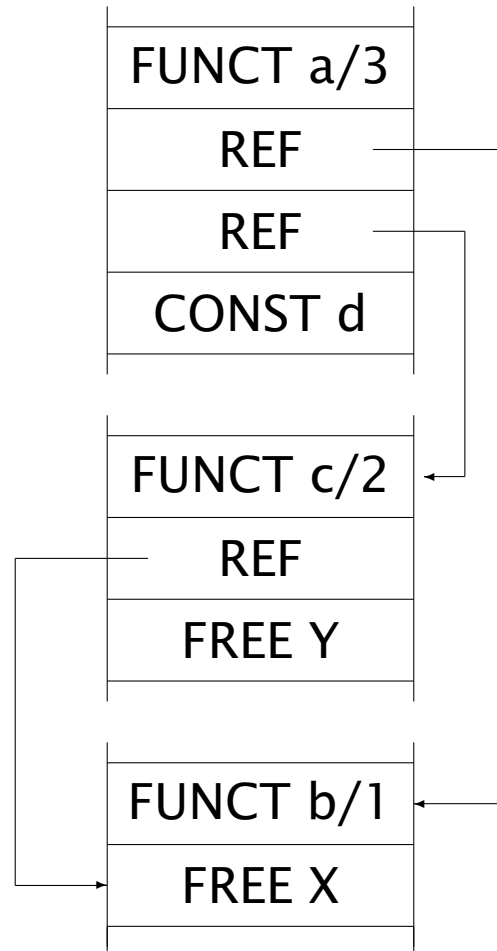
● Kopírování struktur:

$A = a(K, L, M)$ ,  $K = b(X)$ ,  $L = c(X, Y)$ ,  $M = d$



# Srovnání: příklad – pokračování

● Kopírování struktur:  $A = a(K, L, M), K = b(X), L = c(X, Y), M = d$



tj. identické jako přímé vytvoření termu  $a(b(X), c(X, Y), d)$

# Srovnání II

- **Složitost algoritmů pro přístup k jednotlivým argumentům**
  - sdílení struktur: nutná víceúrovňová nepřímá adresace
  - kopírování struktur: bez problémů
  - jednodušší algoritmy usnadňují i optimalizace

# Srovnání II

## ● Složitost algoritmů pro přístup k jednotlivým argumentům

- sdílení struktur: nutná víceúrovňová nepřímá adresace
- kopírování struktur: bez problémů
- jednodušší algoritmy usnadňují i optimalizace

## ● Lokalita přístupů do paměti

- sdílení struktur: přístupy rozptýleny po paměti
- kopírování struktur: lokalizované přístupy
- při stránkování paměti – rozptýlení vyžaduje přístup k více stránkám

# Srovnání II

- **Složitost algoritmů pro přístup k jednotlivým argumentům**
  - sdílení struktur: nutná víceúrovňová nepřímá adresace
  - kopírování struktur: bez problémů
  - jednodušší algoritmy usnadňují i optimalizace
- **Lokalita přístupů do paměti**
  - sdílení struktur: přístupy rozptýleny po paměti
  - kopírování struktur: lokalizované přístupy
  - při stránkování paměti – rozptýlení vyžaduje přístup k více stránkám
- Z praktického hlediska neexistuje mezi těmito přístupy zásadní rozdíl



# Řízení výpočtu

## ● Dopředný výpočet

- po úspěchu (úspěšná redukce)
  - jednotlivá volání procedur skončí úspěchem
- klasické volání rekurzivních procedur

# Řízení výpočtu

## ● Dopředný výpočet

- po úspěchu (úspěšná redukce)
  - jednotlivá volání procedur skončí úspěchem
- klasické volání rekurzivních procedur

## ● Zpětný výpočet (backtracking)

- po neúspěchu vyhodnocení literálu (neúspěšná redukce)
  - nepodaří se unifikace aktuálních a formálních parametrů hlavy
- návrat do bodu, kde zůstala nevyzkoušená alternativa výpočtu
  - je nutná obnova původních hodnot jednotlivých proměnných
  - po nalezení místa s dosud nevyzkoušenou klauzulí pokračuje dále dopředný výpočet

# Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**

# Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**
- Dopředný výpočet
  - stav výpočtu v okamžiku volání procedury
  - aktuální parametry
  - lokální proměnné
  - pomocné proměnné ('a la registry)

# Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**
- Dopředný výpočet
  - stav výpočtu v okamžiku volání procedury
  - aktuální parametry
  - lokální proměnné
  - pomocné proměnné ('a la registry)
- Zpětný výpočet (backtracking)
  - hodnoty parametrů v okamžiku zavolání procedury
  - následující klauzule pro zpracování při neúspěchu

# Aktivační záznam a roll-back

● Neúspěšná klauzule mohla nainstanciovat nelokální proměnné

●  $a(X) \text{ :- } X = b(c, Y), Y = d.$        $?- W = b(Z, e), a(W).$

# Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné

●  $a(X) \text{ :- } X = b(c, Y), Y = d.$        $?- W = b(Z, e), a(W).$       (viz instanciace Z)

# Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
    - $a(X) \text{ :- } X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad (\text{viz instanciace } Z)$
  - Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
  - Využijeme vlastností logických proměnných
    - instanciovat lze pouze volnou proměnnou
    - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu
- ⇒ původní hodnoty všech proměnných odpovídají volné proměnné



# Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
  - $a(X) \text{ :- } X = b(c, Y), Y = d. \quad \text{?- } W = b(Z, e), a(W). \quad (\text{viz instanciace } Z)$
- Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
- Využijeme vlastností logických proměnných
  - instanciovat lze pouze volnou proměnnou
  - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu

⇒ původní hodnoty všech proměnných odpovídají volné proměnné
- **Stopa** (trail): zásobník s adresami instanciovaných proměnných
  - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
  - při neúspěchu jsou hodnoty proměnných na stopě v úseku mezi aktuálním a uloženým vrcholem zásobníku změněny na „volná“

# Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
  - $a(X) :- X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad$  (viz instanciace Z)
- Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
- Využijeme vlastností logických proměnných
  - instanciovat lze pouze volnou proměnnou
  - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu

⇒ původní hodnoty všech proměnných odpovídají volné proměnné
- **Stopa** (trail): zásobník s adresami instanciovaných proměnných
  - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
  - při neúspěchu jsou hodnoty proměnných na stopě v úseku mezi aktuálním a uloženým vrcholem zásobníku změněny na „volná“
- **Globální zásobník**: pro uložení složených termů
  - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
  - při neúspěchu vrchol zásobníku snížen podle uschované hodnoty v aktivačním záznamu

# Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku  $\Rightarrow$  **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu

# Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku  $\Rightarrow$  **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

# Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku  $\Rightarrow$  **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

## Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)

# Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku  $\Rightarrow$  **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

## Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)
- alokace pouze okolí pro deterministické procedury

# Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku  $\Rightarrow$  **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

## Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)
- alokace pouze okolí pro deterministické procedury
- možnost odstranění okolí po úspěšném vykonání (i nedeterministické) procedury (pokud okolí následuje po bodu volby dané procedury)
  - pokud je okolí na vrcholu zásobníku

# Řez

● Prostředek pro ovlivnění běhu výpočtu programátorem

● `a(X) :- b(X), !, c(X).      a(3).`  
`b(1).            b(2).`  
`c(1).            c(2).`



# Řez

- Prostředek pro ovlivnění běhu výpočtu programátorem

- $a(X) \text{ :- } b(X), !, c(X). \quad a(3).$   
 $b(1). \quad b(2).$   
 $c(1). \quad c(2).$

- Řez: neovlivňuje dopředný výpočet, má vliv pouze na zpětný výpočet

- Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

- tj. odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

⇒ změna ukazatele na „nejmladší“ bod volby

# Řez

- Prostředek pro ovlivnění běhu výpočtu programátorem

- $a(X) \text{ :- } b(X), !, c(X). \quad a(3).$   
 $b(1). \quad b(2).$   
 $c(1). \quad c(2).$

- Řez: neovlivňuje dopředný výpočet, má vliv pouze na zpětný výpočet

- Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

- tj. odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

⇒ změna ukazatele na „nejmladší“ bod volby

⇒ Vytváření deterministických procedur

⇒ Optimalizace využití zásobníku

# Interpret Prologu

Základní principy:

- klauzule uloženy jako termy
- **programová databáze**
  - pro uložení klauzulí
  - má charakter haldy
    - umožňuje modifikovatelnost prologovských programů za běhu (assert)
- klauzule zřetězeny podle pořadí načtení
  - triviální zřetězení

# Interpret Prologu

Základní principy:

- klauzule uloženy jako termy
- **programová databáze**
  - pro uložení klauzulí
  - má charakter haldy
    - umožňuje modifikovatelnost prologovských programů za běhu (assert)
- klauzule zřetězeny podle pořadí načtení
  - triviální zřetězení

Vyhodnocení dotazu: volání procedur řízené unifikací

# Interpret – Základní princip

1. Vyber redukovaný literál („první“, tj. nejlevější literál cíle)
2. Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál
3. V případě nalezení klauzule založ bod volby procedury
4. Založ dále okolí první klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)

# Interpret – Základní princip

1. Vyber redukovaný literál („první“, tj. nejlevější literál cíle)
2. Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál
3. V případě nalezení klauzule založ bod volby procedury
4. Založ dále okolí první klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)
5. Proved' unifikaci literálu a hlavy klauzule
6. Úspěch  $\Rightarrow$  přidej všechny literály klauzule k cíli („doleva“, tj. na místo redukovaného literálu).  
Tělo prázdné  $\Rightarrow$  výpočet se s úspěchem vrací do klauzule, jejíž adresa je v aktuálním okolí.
7. Neúspěch unifikace  $\Rightarrow$  z bodu volby se obnoví stav a pokračuje se v hledání další vhodné klauzule v databázi.

# Interpret – Základní princip

1. Vyber redukovaný literál („první“, tj. nejlevější literál cíle)
2. Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál
3. V případě nalezení klauzule založ bod volby procedury
4. Založ dále okolí první klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)
5. Proved' unifikaci literálu a hlavy klauzule
6. Úspěch  $\Rightarrow$  přidej všechny literály klauzule k cíli („doleva“, tj. na místo redukovaného literálu).  
Tělo prázdné  $\Rightarrow$  výpočet se s úspěchem vrací do klauzule, jejíž adresa je v aktuálním okolí.
7. Neúspěch unifikace  $\Rightarrow$  z bodu volby se obnoví stav a pokračuje se v hledání další vhodné klauzule v databázi.
8. Pokud není nalezena odpovídající klauzule, výpočet se vrací na předchozí bod volby (krátí se lokální i globální zásobník).
9. Výpočet končí neúspěchem: neexistuje již bod volby, k němuž by se výpočet mohl vrátit.
10. Výpočet končí úspěchem, jsou-li úspěšně redukovány všechny literály v cíli.

# Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.



# Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus  
Současně poznačí adresy instanciovaných proměnných na stopu.

# Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus  
Současně poznačí adresy instanciovaných proměnných na stopu.

- „Interpret“:

```
interpret(Query, Vars) :- call(Query), success(Query, Vars).  
interpret(_,_) :- failure.
```

- dotaz vsazen do kontextu této speciální nedeterministické procedury
- tato procedura odpovídá za korektní reakci systému v případě úspěchu i neúspěchu

# Optimalizace: Indexace

- Zřetězení klauzulí podle pořadí načtení velmi neefektivní
- Provázání klauzulí se stejným funktorem a aritou hlavy (tvoří jednu **proceduru**)
  - tj., **indexace procedur**
- Hash tabulka pro vyhledání první klauzule
- Možno rozhodnout (parciálně) determinismus procedury

# Indexace argumentů

`a(1) :- q(1).`

`a(a) :- b(X).`

`a([A|T]) :- c(A,T).`

- Obecně nedeterministická
- Při volání s alespoň částečně instanciováním argumentem vždy deterministická (pouze jedna klauzule může uspět)

# Indexace argumentů

$a(1) \text{ :- } q(1) .$

$a(a) \text{ :- } b(X) .$

$a([A|T]) \text{ :- } c(A,T) .$

- Obecně nedeterministická
- Při volání s alespoň částečně instanciováním argumentem vždy deterministická (pouze jedna klauzule může uspět)
- **Indexace podle prvního argumentu**

Základní typy zřetězení:

- podle pořadí klauzulí (aktuální argument je volná proměnná)
- dle konstant (aktuální je argument konstanta)
- formální argument je seznam (aktuální argument je seznam)
- dle struktur (aktuální argument je struktura)

# Indexace argumentů II

- Složitější indexační techniky
  - podle všech argumentů
  - podle nejvíce diskriminujícího argumentu
  - kombinace argumentů (indexové techniky z databází)
    - zejména pro přístup k faktům

# Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze  $\Rightarrow$  lineární paměťová náročnost cyklů

# Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze  $\Rightarrow$  lineární paměťová náročnost cyklů

**Optimalizace koncové rekurze (*Tail Recursion Optimisation*), TRO:**

Okolí se odstraní **před** rekurzivním voláním posledního literálu klauzule, pokud je klauzule resp. její volání deterministické.

Řízení se nemusí vracet:

- v případě úspěchu se rovnou pokračuje
- v případě neúspěchu se vrací na předchozí bod volby („nad“ aktuální klauzulí)
  - aktuální klauzule nemá dle předpokladu bod volby

Rekurzivně volaná klauzule může být volána přímo z kontextu volající klauzule.



# TRO – příklad

Program:

```
append([], L, L).
```

```
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

# TRO – příklad

Program:

```
append([], L, L).
```

```
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

append volán rekurzivně 4krát

- bez TRO: 4 okolí, lineární paměťová náročnost
- s TRO: 1 okolí, konstatní paměťová náročnost

# Optimalizace posledního volání

TRO pouze speciální případ

obecné **optimalizace posledního volání** (*Last Call Optimization*), LCO

# Optimalizace posledního volání

TRO pouze speciální případ

obecné **optimalizace posledního volání** (*Last Call Optimization*), LCO

Okolí (před redukcí posledního literálu)

odstraňováno vždy, když leží na vrcholu zásobníku.

# Optimalizace posledního volání

TRO pouze speciální případ

obecné **optimalizace posledního volání** (*Last Call Optimization*), LCO

Okolí (před redukcí posledního literálu)

odstraňováno vždy, když leží na vrcholu zásobníku.

## Nutné úpravy interpretu

● disciplina směřování ukazatelů

● vždy „mladší“ ukazuje na „starší“ („mladší“ budou odstraněny dříve)

● z lokálního do globálního zásobníku

vyhneme se vzniku „visících odkazů“ při předčasném odstranění okolí

# Optimalizace posledního volání

TRO pouze speciální případ

obecné **optimalizace posledního volání** (*Last Call Optimization*), LCO

Okolí (před redukcí posledního literálu)

odstraňováno vždy, když leží na vrcholu zásobníku.

## Nutné úpravy interpretu

● disciplina směřování ukazatelů

● vždy „mladší“ ukazuje na „starší“ („mladší“ budou odstraněny dříve)

● z lokálního do globálního zásobníku

vyhneme se vzniku „visících odkazů“ při předčasném odstranění okolí

● „globalizace“ lokálních proměnných: lokální proměnné posledního literálu

● nutno přesunout na globální zásobník

● pouze pro neinstanciované proměnné

# Warrenův abstraktní počítač, WAM I.

Navržen D.H.D. Warrenem v roce 1983, modifikace do druhé poloviny 80. let

Datové oblasti:

## ● **Oblast kódu** (programová databáze)

- separátní oblasti pro uživatelský kód (modifikovatelný) a vestavěné predikáty (nemění se)
- obsahuje rovněž všechny statické objekty (texty atomů a funktorů apod.)

## ● **Lokální zásobník (*Stack*)**

## ● **Stopa (*Trail*)**

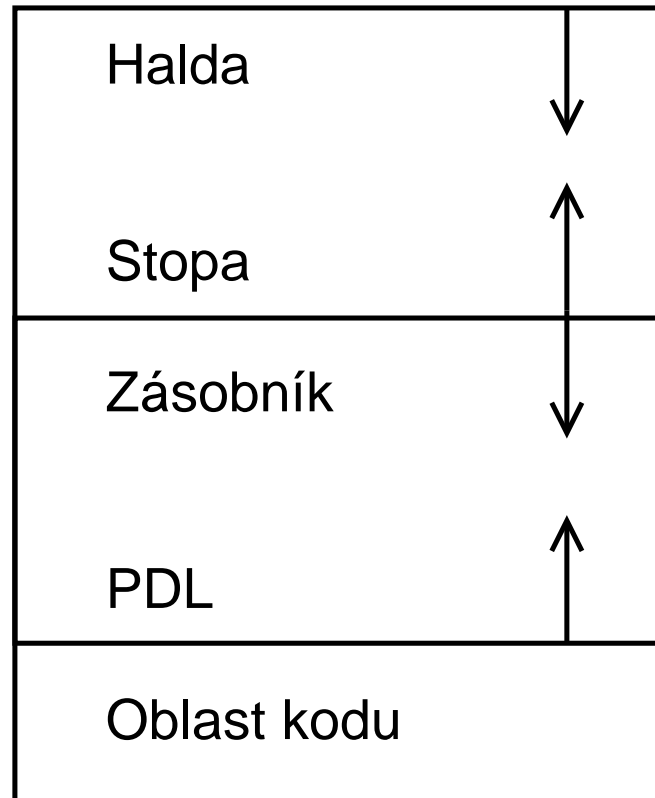
## ● **Globální zásobník n. halda (*Heap*)**

## ● **Pomocný zásobník (*Push Down List, PDL*)**

- pracovní paměť abstraktního počítače
- použitý v unifikaci, syntaktické analýze apod.

# Rozmístění datových oblastí

## ● Příklad konfigurace



## ● Halda i lokální zásobník musí růst stejným směrem

- lze jednoduše porovnat stáří dvou proměnných srovnáním adres  
využívá se při zabrání vzniku visících odkazů



# Registry WAMu

## ● Stavové registry:

P čítač adres (Program counter)

CP adresa návratu (Continuation Pointer)

E ukazatel na nejmladší okolí (Environment)

B ukazatel na nejmladší bod volby (Backtrack point)

TR vrchol stopy (TRail)

H vrchol haldy (Heap)

HB vrchol haldy v okamžiku založení posledního bodu volby (Heap on Backtrack point)

S ukazatel, používaný při analýze složených termů (Structure pointer)

CUT ukazatel na bod volby, na který se řezem zařízne zásobník

● **Argumentové registry:** A1, A2, . . . (při předávání parametrů n. pracovní registry)

● **Registry pro lokální proměnné:** Y1, Y2, . . .

● abstraktní znázornění lok. proměnných na zásobníku

# Typy instrukcí WAMu

- **put instrukce** – příprava argumentů před voláním podcíle
  - žádná z těchto instrukcí nevolá obecný unifikační algoritmus
- **get instrukce** – unifikace aktuálních a formálních parametrů
  - vykonávají činnost analogickou instrukcím `unify`
  - obecná unifikace pouze při `get_value`
- **unify instrukce** – zpracování složených termů
  - jednoargumentové instrukce, používají registr `S` jako druhý argument
  - počáteční hodnota `S` je odkaz na 1. argument
  - volání instrukce `unify` zvětší hodnotu `S` o jedničku
  - obecná unifikace pouze při `unify_value` a `unify_local_value`
- **Indexační instrukce** – indexace klauzulí a manipulace s body volby
- **Instrukce řízení běhu** – předávání řízení a explicitní manipulace s okolím

# Instrukce put a get: příklad

Příklad:  $a(X,Y,Z) :- b(f,X,Y,Z).$

get\_var      A1,A5

get\_var      A2,A6

get\_var      A3,A7

put\_const    A1,f

put\_value    A2,A5

put\_value    A3,A6

put\_value    A4,A7

execute      b/4

# WAM – optimalizace

1. Indexace klauzulí
2. Generování optimální posloupnosti instrukcí WAMu
3. Odstranění redundancí při generování cílového kódu.

# WAM – optimalizace

1. Indexace klauzulí
2. Generování optimální posloupnosti instrukcí WAMu
3. Odstranění redundancí při generování cílového kódu.

● Příklad:  $a(X,Y,Z) :- b(f,X,Y,Z)$ .

naivní kód (vytvoří kompilátor pracující striktně zleva doprava) vs.

optimalizovaný kód (počet registrů a tedy i počet instrukcí/přesunů v paměti snížen):

get_var	A1,A5		get_var	A3,A4
get_var	A2,A6		get_var	A2,A3
get_var	A3,A7		get_var	A1,A2
put_const	A1,f		put_const	A1,f
put_value	A2,A5		execute	b/4
put_value	A3,A6			
put_value	A4,A7			
execute	b/4			

# Instrukce WAMu

## get instrukce

get_var	Ai, Y
get_value	Ai, Y
get_const	Ai, C
get_nil	Ai
get_struct	Ai, F/N
get_list	Ai

## put instrukce

put_var	Ai, Y
put_value	Ai, Y
put_unsafe_value	Ai, Y
put_const	Ai, C
put_nil	Ai
put_struct	Ai, F/N
put_list	Ai

## unify instrukce

unify_var	Y
unify_value	Y
unify_local_value	Y
unify_const	C
unify_nil	
unify_void	N

## instrukce řízení

allocate	
deallocate	
call	Proc/N, A
execute	Proc/N
proceed	

## indexační instrukce

try_me_else	Next	try	Next
retry_me_else	Next	retry	Next
trust_me_else	fail	trust	fail
cut_last		switch_on_term	Var, Const, List, Struct
save_cut	Y	switch_on_const	Table
load_cut	Y	switch_on_struct	Table

# WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
  - první klauzule: `try_me_else`; založí bod volby
  - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
  - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu

# WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
  - první klauzule: `try_me_else`; založí bod volby
  - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
  - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
  - `try`
  - `retry`
  - `trust`



# WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
  - první klauzule: `try_me_else`; založí bod volby
  - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
  - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
  - `try`
  - `retry`
  - `trust`
- „Rozskokové” instrukce (dle typu a hodnoty argumentu):
  - `switch_on_term Var, Const, List, Struct`  
výpočet následuje uvedeným návěstím podle typu prvního argumentu
  - `switch_on_YY`: hashovací tabulka pro konkrétní typ (konstanta, struktura)

# Příklad indexace instrukcí

## Proceduře

a(atom) :- body1.

a(1) :- body2.

a(2) :- body3.

a([X|Y]) :- body4.

a([X|Y]) :- body5.

a(s(N)) :- body6.

a(f(N)) :- body7.

## odpovídají instrukce

a: switch\_on\_term L1, L2, L3, L4

L2: switch\_on\_const atom :L1a

1 :L5a

2 :L6a

L3: try L7a

trust L8a

L4: switch\_on\_struct s/1 :L9a

f/1 :L10a

L1: try\_me\_else L5

L1a: body1

L5: retry\_me\_else L6

L5a: body2

L6: retry\_me\_else L7

L6a: body3

L7: retry\_me\_else L8

L7a: body4

L8: retry\_me\_else L9

L8a: body5

L9: retry\_me\_else L10

L9a: body6

L10: trust\_me\_else fail

L10a: body7

# WAM – řízení výpočtu

 `execute Proc`: ekvivalentní příkazu `goto`

# WAM – řízení výpočtu

- `execute Proc`: ekvivalentní příkazu `goto`
- `proceed`: zpracování faktů

# WAM – řízení výpočtu

- `execute Proc`: ekvivalentní příkazu `goto`
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)

# WAM – řízení výpočtu

- `execute Proc`: ekvivalentní příkazu `goto`
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)
- `deallocate`: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)
- `call Proc,N`: zavolá `Proc`, `N` udává počet lok. proměnných (odpovídá velikosti zásobníku)

# WAM – řízení výpočtu

- `execute Proc`: ekvivalentní příkazu `goto`
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)
- `deallocate`: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)
- `call Proc,N`: zavolá `Proc`, `N` udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

`a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.`

```
generujeme instrukce    allocate
                           call b/1,4
                           call c/2,3
                           call d/1,2
                           call e/1,1
                           deallocate
                           execute f/0
```

# WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad:  $?- a(X) .$  může být nedeterministické, ale  $?- a(1) .$  může být deterministické



# WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: `?- a(X) .` může být nedeterministické, ale `?- a(1) .` může být deterministické

`cut_last: B := CUT`      `save_cut Y: Y := CUT`      `load_cut Y: B := Y`

# WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: `?- a(X) .` může být nedeterministické, ale `?- a(1) .` může být deterministické

`cut_last: B := CUT`      `save_cut Y: Y := CUT`      `load_cut Y: B := Y`

Hodnota registru B je uchovávána v registru CUT instrukcemi `call` a `execute`.

Je-li řez prvním predikátem klauzule, použije se rovnou `cut_last`. V opačném případě se použije jako první instrukce `save_cut Y` a v místě skutečného volání řezu se použije `load_cut Y`.

# WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: `?- a(X) .` může být nedeterministické, ale `?- a(1) .` může být deterministické

```
cut_last:   B := CUT           save_cut Y:   Y := CUT           load_cut Y:   B := Y
```

Hodnota registru B je uchovávána v registru CUT instrukcemi `call` a `execute`.

Je-li řez prvním predikátem klauzule, použije se rovnou `cut_last`. V opačném případě se použije jako první instrukce `save_cut Y` a v místě skutečného volání řezu se použije `load_cut Y`.

Příklad: `a(X,Z) :- b(X), !, c(Z) .`

`a(2,Z) :- !, c(Z) .`

`a(X,Z) :- d(X,Z) .`

odpovídá

```
save_cut Y2; get A2,Y1; call b/1,2; load_cut Y2; put Y1,A1; execute c/1
```

```
get_const A1,2; cut_last; put A2,A1; execute c/1
```

```
execute d/2
```