Test existence

- vyskytuje se v seznamu alespoň jeden prvek s danou vlastností?
- ```
  test_existence(Hodnoty, [Hlava|_]) :-
      vlastnost(Hodnoty, Hlava).
  test_existence(Hodnoty, [_|Ocas]) :-
      test_existence(Hodnoty, Ocas).
  ```
- příklad: hledáme určitý prvek

  ```
  obsahuje(N, [N|_]).                /* zkratka za N=Hlava */
  obsahuje(N, [_|Ocas]) :- obsahuje(N, Ocas).
  ```
  ```

Test všech prvků

- mají všechny prvky seznamu danou vlastnost?
- ```
  vsichni(Hodnoty, []).
  vsichni(Hodnoty, [Hlava|Ocas]) :-
                          vlastnost(Hodnoty, Hlava),
                          vsichni(Hodnoty, Ocas).
  ```
- příklad: seznam obsahuje jen samohlásky

  ```
  samohlasky([]).
  samohlasky([Hlava|Ocas]) :-
   obsahuje(Hlava, [a, e, i, o, u, y]), samohlasky(Ocas).
  ```
  ```

Vracení hodnoty (1)

- vyžaduje jeden argument navíc – pro výsledek
- 1. alternativa – hodnota vychází z nalezeného prvku seznamu (prvku, který má určitou vlastnost)
-
```
dej_vysledek(Hodnoty,[Hlava|_], Vysledek) :-
    vlastnost(Hodnoty, Hlava),
    urci_vysledek(Hodnoty, Hlava, Vysledek).
dej_vysledek(Hodnoty,[_|Ocas], Vysledek) :-
    dej_vysledek(Hodnoty, Ocas, Vysledek).
```

Příklad: Pořadí v seznamu

- hledáme pořadové číslo prvku seznamu (kolikátý je)
- číslujeme od 1
-
```
kolikaty(N, [N|_], 1).
kolikaty(N, [_|Ocas], Poradi) :-
                    kolikaty(N, Ocas, Por),
                    Poradi is Por + 1.
```

Vracení hodnoty (2)

- 2. alternativa – hodnota se určuje ze všech prvků seznamu
-
```
vsechno(Hodnoty, [], Vysledek).
/* výsledek prázdného seznamu bývá pevně dán */
vsechno(Hodnoty, [Hlava|Ocas], Vysledek) :-
    vsechno(Hodnoty, Ocas, Castecny),
    kombinuj(Hodnoty, Hlava, Castecny, Vysledek).
```

Příklad: součet čísel v seznamu

- opět binární predikát soucet(X,Y) – X je seznam čísel a Y jejich součet
-
```
soucet([], 0).
soucet([Hlava|Ocas], Suma) :-
    soucet(Ocas, SumOc),
    Suma is Hlava + SumOc.
```

Filtr

- hodnoty ze vstupního seznamu upravuje a propouští do výstupního
- ```
  filtr(_,[],[]).
  filtr(Hodnoty, [Hlava|Ocas], [ModHl|ModOc]) :-
      uprav_hodnotu(Hodnoty, Hlava, ModHl),
      filtr(Hodnoty, Ocas, ModOc).
  ```
  ```

Příklad: Vynásobit seznam číslem

- ```
  nasob(_, [], []).
  nasob(N, [Hlava|Ocas], [Soucin|NovyOcas] ) :-
                                  Soucin is Hlava * N,
                                  nasob(N, Ocas, NovyOcas).
  ```
- budování struktury v hlavičce pravidla – struktura se skládá při návratu z rekurze
  ```

Alternativní řešení

- budování struktury v těle pravidla – buduje se při dopředném chodu rekurze
- přidáme argument, v něm se postupně buduje výsledek; zde obsahuje již vynásobená čísla
- na konci se jen pasivně předá
- ```
  nasob2(_, [], Vysledek, Vysledek).
  nasob2(N, [Hlava|Ocas], Upraveno, Vysledek ) :-
          Soucin is Hlava * N,
          nasob2(N, Ocas, [Soucin|Upraveno], Vysledek).
  ```
  ```

Problémy alternativního řešení

- obrací pořadí – Upraveno obsahuje dříve zpracovaná čísla (byla před Hlavou)
- nutno inicializovat přidaný argument
- řešení: obalující predikát, který se postará o inicializaci přidaného argumentu a úpravu výsledku do požadované podoby
-
```
nasob(N, Seznam, Vysledek) :-
    nasob2(N, Seznam, [], Mezivysledek),
    reverse(Mezivysledek, Vysledek).
```

Příklad: kladná čísla ze seznamu

- binární predikát pozitivni(X,Y), kde X je seznam čísel a Y seznam kladných čísel z X
- ```
  pozitivni([], []).
  pozitivni([Hlava|Ocas], [Hlava|PozOcas]) :-
                  Hlava>0, pozitivni(Ocas, PozOcas).
  pozitivni([Hlava|Ocas], PozOcas) :-
                  Hlava=<0, pozitivni(Ocas, PozOcas).
  ```
  ```

Příklad: Spojení seznamů

- predikát append(L1,L2,L3), kde seznam L3 vznikne připojením seznamu L2 na konec seznamu L1
- append([], L2, L2).
  append([Hlava|Ocas], L2, [Hlava|L3]) :-
  append(Ocas, L2, L3).
- prvky L1 se přesouvají do výsledku, až se L1 vyprázdní; připojením L2 za prázdný seznam vznikne L2

Vestavěné operace se seznamy

- is_list(+X) — je X seznam?
- length(+L, ?N) — N je délka seznamu L
- member(?X, ?L) — X je prvkem L
- select(?X, ?L1, ?L2) — L2 vznikl odebráním X z L1
- nth0(?N, ?L, ?X)
nth1(?N, ?L, ?X) — X je Ntým prvkem L, počítáno od 0 nebo od 1
- append(?L1, ?L2, ?L3) — L3 je spojením L1 a L2
- reverse(?L1, ?L2) — L2 je L1 v opačném pořadí

Vestavěné operace se seznamy

- sort(+L, –Serazen) Serazen obsahuje alfabeticky seřazené prvky L, bez duplikátů
  msort s duplikáty
  keysort řadí dvojice klíč-hodnota
  predsort řadí podle zadaného predikátu
- subset(+L1, +L2) L1 je podmnožinou L2
- intersection(+L1, +L2, –L3) L3 je průnikem L1 a L2
  union(+L1, +L2, –L3) sjednocením
  subtract(+L1, +L2, –L3) rozdílem

Vzory volání

- argument predikátu může mít tři vzory volání:
  - **vstupní** (označován v dokumentaci +)
  - **výstupní** (–)
  - **neurčený** (?) – může být vstupní i výstupní
- např. následník:
  naslednik(X,Y) :- Y is X + 1.
  X je vstupní, Y může být vstupní i výstupní
- záhodno vyznačit v dokumentaci: naslednik(+X, ?Y)

Reverzibilita

- predikáty v Prologu vyjadřují vztah mezi svými argumenty – často je lze používat i „proti srsti“
- příklad: vymazání prvku ze seznamu:

```
del(_,[],[]).
del(Prvek,[Prvek|Ocas], Ocas).
del(Prvek,[Hlava|Ocas], [Hlava|Smazano]) :-
                                del(Prvek, Ocas, Smazano).
```

- klasické použití `del(2, [1, 2, 3], L)` ale lze i `del(2, L, [3, 4, 5])` nebo `del(X, [1, 2, 3], [1, 3])`

Řez

- zapisuje se jako vykřičník !
- vždy splněn (true), ale omezuje backtracking
- po přechodu přes řez
  - se při hledání řešení (či jeho opakování) daného predikátu nesmí vrátit před něj – veškeré vazby proměnných na hodnoty provedené před řezem zůstávají neměnné
  - nesmí použít pro daný predikát jinou klauzuli, návrat k predikátu (portem redo) způsobí neúspěch (port fail), dojde k opuštění predikátu a odvolání řezu

Příklad

```
p(X) :- r(X), !, s(X).
p(X) :- t(X).
r(a).
r(b).
s(a).
s(b).
t(c).
```

```
?- p(X).
X = a.
false
```

Příklad: Prvek seznamu

- stávající predikát obsahuje() uspěje opakovaně, pokud se prvek v seznamu vyskytuje vícekrát
- v některých aplikacích může vadit
- obsahuje(Hlava, [Hlava|_]) :- !.
  obsahuje(X, [_|Ocas]) :- obsahuje(X, Ocas).
- nedá se už ale použít jako generátor
  obsahuje(X, [raz, dva, tri]) dosadí jen první prvek

Determinovaný predikát

- má nanejvýš jedno řešení
- často se používá řez, aby se po nalezení řešení zabránilo hledání alternativ

Příklad: Součet prvních N čísel

- ```
  soucet(1, 1).
  soucet(N, Suma) :-
          M is N – 1,
          soucet(M, SumaM),
          Suma is N + SumaM.
  ```
- ```
  soucet(3, X)
  ```
  ; vede k nekonečné smyčce
- není determinovaný: backtracking po soucet(1,1) použije další klauzuli a vydá se do záporných čísel

Oprava

- determinovaná verze – jakmile uplatněním soucet(1, 1) najde řešení, nebude se pokoušet o žádné další
-
```
soucet(1, 1) :- !.
soucet(N, Suma) :-
        M is N – 1,
        soucet(M, SumaM),
        Suma is N + SumaM.
```

Oprava jinak

- před zmenšením otestovat, zda je hodnota přípustná (větší než 1)
-
```
soucet(1, 1).
soucet(N, Suma) :-
        N > 1,
        M is N – 1,
        soucet(M, SumaM),
        Suma is N + SumaM.
```
- lepší, řeší i dotazy typu `soucet(–5,X)`

Řez a negace (1)

- máme-li k dispozici fakta muz(), lze vytvořit predikát zena():
  zena(X) :- muz(X), !, fail.
  zena(X).
- pokud je X mužem, nemůže být ženou – řez zabrání hledání alternativ pro zena(X) a následné selhání určí výsledek
- technika označovaná jako **řez-selhání (cut-fail)**

Řez a negace (2)

- totéž lze stručněji pomocí negace
  ```
  zena(X) :- \+(muz(X)).
  ```
- negace by se dala definovat jako
  ```
  \+(Cíl) :-
      call(Cíl), !, fail.
  \+(Cíl).
  ```
- call() je predikát, který vyvolá rezoluční mechanismus pro svůj argument; výsledek rezoluce true/fail je i výsledkem predikátu

Vstup a výstup

- **read(X)** načte term ze vstupu a unifikuje jej s X, **vstup ukončit tečkou**
- **write(X)** vypíše hodnotu X
- **nl** odřádkuje ve výstupu
- při redo selžou (implementace se zde mohou lišit)

Příklad: testovací cyklus

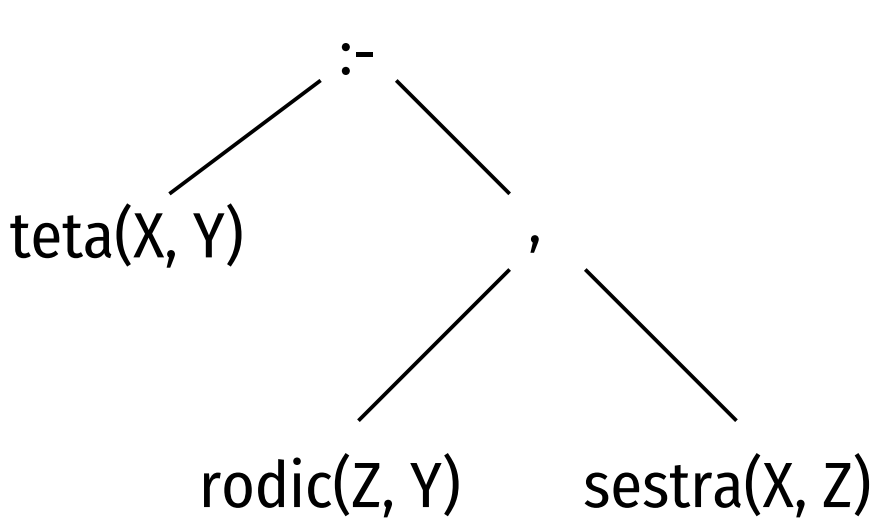
- interaktivní testování predikátu pro různé hodnoty
-
```
predikat(X, Y) :- Y is X * X  * X.
test :-
        read(X),          /* načte testovanou hodnotu */
        X > -100,         /* hodnota ≤ –100 zastaví */
        predikat(X, Y),
        write(Y), nl,     /* vypíše výsledek */
        test.             /* vzhůru na další hodnotu */
```

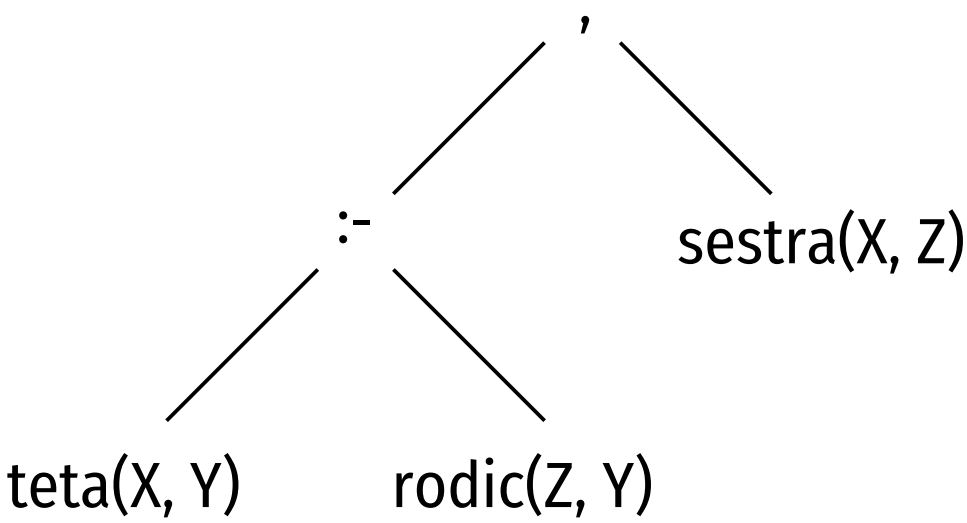
Operátory

- 3 základní typy:
- **infixové** – operátor mezi operandy
  2 + 3, X < Y, konjunkce (,), :-
- **prefixové** – nejprve operátor, pak operandy
  \+ vetsi(X, Y), –10
- **postfixové** – nejprve operandy, pak operátor
  v Prologu není žádný předdefinován, lze vytvořit např. X je_prvocislo

Priorita (1)

- jak chápat sekvenci operátorů, např.
  teta(X, Y) :- rodic(Z, Y), sestra(X, Z).
- je třeba definovat, co má přednost



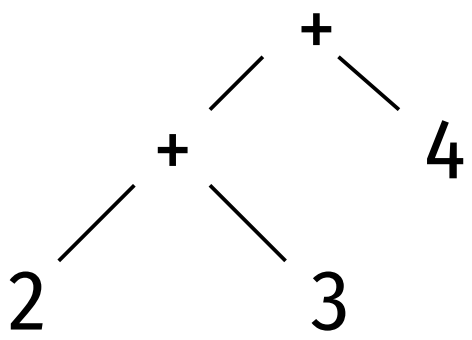


Priorita (2)

- priority vybraných operátorů:
  - `:-` 1200
  - `,` 1000
  - `\+` 900
  - porovnání 700
  - `+ -` 500
  - `* /` 400

Asociativita

- jak vyhodnocovat operátory se stejnou prioritou
- zápis v Prologu:
  - **yfx** asociativní zleva, např. +, x~operátor s nižší prioritou, y~operátor s prioritou nižší nebo stejnou
  - **xfy** asociativní zprava, např. ,
  - **xfx** není asociativní, např. :-
- 2 + 3 + 4



Definice operátoru

- **op(*priorita*, *asociativita*, *jméno*)**
- příklad: definice obvyklých logických operátorů

```
op(1000, xfy, and).
X and Y :- call(X), call(Y).
op(1050, xfy, or).
X or Y :- call(X).
X or Y :- call(Y).
```

- operátor umožňuje infixový zápis bez závorek, lze zavést i prefixové (fy, fx) a postfixové (yf, xf)

Obecná schémata predikátů (1)

- **generátor – test**
  - generuj_a_testuj(Data, X) :-
    ..., generuj(Data, X), testuj(Data, X), ...
  - generuj() postupně generuje alternativy; testuj() ověří, zda mají požadované vlastnosti; pokud uspěje, zpracuje se hodnota; v opačném případě se backtracking vrátí ke generuj() a vygeneruje další hodnotu

Příklad: Čísla s celočíselnou odmocninou

```
int(1).                                 /* generátor celých čísel */
int(N) :- int(M), N is M + 1.           /* každé redo vrací o 1 větší */

hledej_odmocnitelne(Limit) :-
        int(X),
        testuj_odmocninu(X),
        vypis(X, Limit),
        X > Limit.

testuj_odmocninu(X) :- Y is round(sqrt(X)),  X is Y * Y.

vypis(X, Limit) :- X =< Limit, write(X), nl.
vypis(_, _).
```

Obecná schémata predikátů (2)

- **test – zpracování**
  - testuj_a_zpracuj(Data, X, Y) :-
    testuj(Data, X), zpracuj(Data, X, Y).
  - testuj() ověří, zda vstupní parametr X má kýžené vlastnosti; pokud ano, zpracuj() použije jeho hodnotu, typicky k určení výstupní hodnoty Y

Příklad: konverze predikátů

- na základě faktů muz(X) definovat predikát pohlavi(X, Y)
- pohlavi(X, Pohlavi) :- muz(X), Pohlavi = muz.
  pohlavi(X, Pohlavi) :- \+(muz(X)), Pohlavi = zena.
- zjednodušeně:
  pohlavi(X, muz) :- muz(X).
  pohlavi(X, zena) :- \+(muz(X)).

Konstruktivní selhání

- úmyslné selhání vynutí hledání dalšího řešení
- konstruktivni_selhani(Data) :-
  generuj(Data, X), pouzij(X), fail.
  konstruktivni_selhani(_).
- pouzij() bývá koncipováno jako vedlejší efekt – vydá true, ale provede nějakou další akci, jež přímo nesouvisí s hledáním řešení; např. write(X)
- generuj() musí po čase selhat, aby nebylo nekonečné

Příklad: Čísla s celočíselnou odmocninou

```
int(1).                                  /* generátor celých čísel */
int(N) :- int(M), N is M + 1.    /* každé redo o 1 větší */

hledej_odmocnitelne(Limit) :-
        int(X),
        testuj_odmocninu(X),
        vypis(X, Limit).

testuj_odmocninu(X) :- Y is sqrt(X),  X is Y * Y.

vypis(X, Limit) :- X =< Limit, write(X), nl, fail.
vypis(X, Limit) :- X > Limit.
```

Doporučení (1)

- dodržujte obvyklé zásady slušnosti
  - mnemotechnické identifikátory
  - dekompozice – složitou věc raději rozdělit do několika jednodušších
  - triky důsledně komentovat
- komentáře
  - /* komentář */
  - % komentář až do konce řádku

Doporučení (2)

- řez
  - používejte opatrně
  - pokud je to možné, dávejte přednost \+
- alternativy
  - raději dvojici klauzulí než logické nebo (;)
- vedlejší efekty
  - jsou nebezpečné, raději se jim vyhýbejte

vytvořeno s podporou
projektu ESF