KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ◇ poznámky k přednášce

10. Lexikální uzávěry

verze z 3. prosince 2019

1 Akumulace na posloupnostech

V této části ještě nebudeme mluvit přímo o lexikálních uzávěrech, ale na příkladech ukážeme, že funkce můžeme chápat jako hodnoty reprezentující jiné, například matematické objekty.

První příklad se bude týkat **sčítání prvků posloupnosti**. V minulosti jsme už psali funkci podobnou této:

```
(defun sum-numbers (from to)
  (if (< to from)
      0
    (+ from (sum-numbers (+ from 1) to))))
```

Funkce sečte všechna celá čísla počínaje prvním a konče druhým argumentem (neboli celá čísla ze zadaného intervalu):

```
CL-USER 10 > (sum-numbers 0 100)
5050
```

Funkce sama o sobě není užitečná, protože prvky aritmetické posloupnosti můžeme sečíst pomocí vzorečku. Poslouží nám ale jako základ k dalším úvahám.

Co kdybychom chtěli sečíst druhé mocniny celých čísel ze zadaného intervalu? Funkce, která to dělá, by mohla vypadat například takto:

```
(defun sum-squares (from to)
  (if (< to from)
      0
    (+ (expt from 2) (sum-squares (1+ from) to))))
```

Funkce sčítá členy *posloupnosti druhých mocnin celých čísel*. Je to posloupnost, jejíž člen o indexu n je číslo n^2, tedy posloupnost

$$0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121 \ldots$$

(člen o indexu 0 je 0, další 1, $2^2 = 4$, $3^2 = 9$, atd.; posloupnosti obvykle indexujeme od nuly)

Například součet členů počínaje třetím a konče šestým je $9 + 16 + 25 + 36 = 86$, jak můžeme ověřit aplikací funkce:

```
> (sum-squares 3 6)
86
```

Funkci můžeme přepsat pomocí funkce `power2`, kterou jsme si kdysi napsali,

```
(defun power2 (x)
  (* x x))
```

takto:

```
(defun sum-squares (from to)
  (if (< to from)
      0
    (+ (power2 from) (sum-squares (1+ from) to))))
```

Obecně, součet členů posloupnosti čísel je jednou z nejvíce v praxi řešených matematických úloh.

Z našeho pohledu může být posloupnost dána funkcí, která pro dané přirozené číslo n vrátí jako hodnotu n-tý člen posloupnosti. Například posloupnost druhých mocnin přirozených čísel je dána (říkáme také *reprezentována*) funkcí `power2`.

Zdrojový kód funkce `sum-numbers` a `sum-squares` se od sebe liší jen velmi málo. To ještě vynikne, pokud v první funkci použijeme funkci `identity`, která funguje takto:

```
CL-USER 13 > (identity 2)
2

CL-USER 14 > (identity nil)
NIL

CL-USER 15 > (identity 'a)
A
```

a mohla by být napsána takto:

```
(defun my-identity (x)
  x)
```

Funkci `sum-numbers` můžeme s její pomocí přepsat:

```
(defun sum-numbers (from to)
  (if (< to from)
      0
    (+ (identity from) (sum-numbers (+ from 1) to))))
```

Posloupnost přirozených čísel včetně nuly, tedy posloupnost

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \ldots$$

totiž lze zadat (reprezentovat) funkcí `identity`. Vidíme, že i takové triviální funkce jako `identity` mohou být užitečné. Dnes se setkáme ještě s jednou takovou.

Dalším příkladem posloupnosti reprezentované funkcí je už známá Fibonacciho posloupnost zadaná funkcí

```
(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 2)) (fib (- n 1))))))
```

Samozřejmě můžeme vymyslet mnoho dalších funkcí, které reprezentují nějakou zajímavou posloupnost.

Pro přístup k prvkům posloupnosti si napíšeme funkci `mem` (jako *member*). Vypadá to jako formalita, ale funkce nám pomůže zlepšit čitelnost kódu a případné budoucí změny v reprezentaci posloupností.

```
(defun mem (seq index)
  (funcall seq index))
```

Následující obecná funkce sčítá členy dané posloupnosti v zadaném intervalu. Posloupnost je dána funkcí, která ji reprezentuje (parametr `seq`):

```
(defun sum-sequence (seq from to)
  (if (< to from)
      0
    (+ (mem seq from)
       (sum-sequence seq (+ from 1) to))))
```

Součet čísel v daném intervalu teď můžeme vypočítat takto:

```
CL-USER 4 > (sum-sequence #'identity 0 100)
5050
```

Součet druhých mocnin takto:

```
CL-USER 5 > (sum-sequence #'power2 0 100)
338350
```

Funkce `sum-sequence` je samozřejmě dalším příkladem funkce vyššího řádu. Pojďme ji ale ještě dále zobecnit. Prvky posloupnosti nemusíme sčítat, ale můžeme na ně aplikovat libovolnou jinou operaci. Snadno přijdeme na to, že případě sčítání je operace dána funkcí `+` a číslem `0`:

```
(defun sum-sequence (seq from to)
  (if (< to from)
      0
    (+ (mem seq from)
       (sum-sequence seq (+ from 1) to))))
```

To nás inspiruje k napsání funkce `accumulate`:

```
(defun accumulate (seq from to combiner null-val)
  (if (< to from)
      null-val
    (funcall combiner
             (mem seq from)
             (accumulate seq (1+ from) to combiner null-val))))
```

Tato funkce nám dává široké možnosti. Samozřejmě pomocí ní můžeme přepsat funkci `sum-sequence`:

```
(defun sum-sequence (seq from to)
  (accumulate seq from to #'+ 0))
```

ale také třeba napsat funkci na výpočet faktoriálu:

```
(defun fact (n)
  (accumulate #'identity 1 n #'* 1))
```

2 Co víme o uživatelsky definovaných funkcích

Víme, že funkce jsou hodnoty (to jsme se dozvěděli minule), které lze *aplikovat* na jiné hodnoty. Uživatelsky definované funkce se definují makrem `defun` a od minule také makrem `labels`. Při definici funkce se zadává její

- jméno
- seznam parametrů
- tělo

Funkci samotnou můžeme chápat jako datovou strukturu, která obsahuje z těchto informací druhou a třetí, tedy seznam parametrů (na minulé přednášce byla ukázána jeho nová podoba s klíčovým slovem `&rest`) a tělo, které jsme nejprve definovali jako jeden výraz, nyní to může být (kvůli možnosti vedlejšího efektu) seznam výrazů.

Svůj název si funkce pamatovat nemusí (později na této přednášce zavedeme i funkce, které žádný název nemají); funkce je ale uložena jako hodnota funkční vazby svého názvu v nějakém prostředí.

Aplikace uživatelsky definované funkce probíhá s využitím seznamu parametrů a těla funkce. Jak bylo vysvětleno na druhé přednášce:

Aplikace uživatelské funkce

Při aplikaci uživatelské funkce se vytvoří nové prostředí s vazbami parametrů na hodnoty argumentů. Předkem tohoto prostředí se stane globální prostředí. V tomto prostředí se pak sekvenčně vyhodnotí prvky těla funkce. Nové prostředí se vytváří při každé aplikaci znovu.

3 Lexikální uzávěr

Na předchozí přednášce jsem se zmínil (poprvé u funkce `scale-list`), že některé příklady se speciálním operátorem `labels` by vlastně neměly fungovat — tedy aspoň na základě informací, které dosud máme o funkcích.

Ukážu k tomu jednoduchý příklad. Napíšeme funkci `adder`, která k danému číslu vytvoří funkci přičítající toto číslo:

```
(defun adder (add)
  (labels ((res (x) (+ x add)))
    #'res))
```

Test:

```
CL-USER 1 > (setf a (adder 5))
#<Closure (LABELS FUN) subfunction of ADDER 4060000BCC>

CL-USER 2 > (setf b (adder 10))
#<Closure (LABELS FUN) subfunction of ADDER 4060000C2C>

CL-USER 3 > (funcall a 1)
6

CL-USER 4 > (funcall a 10)
15

CL-USER 5 > (funcall b 1)
11

CL-USER 6 > (funcall b 10)
20
```

Jak vidíme, funkce `adder` vrátila jako svůj výsledek pokaždé jinou funkci. Funkce uložená v proměnné `a` je jistě jiná než funkce uložená v proměnné `b`, protože se různě chovají. Jak je to ale možné?

Víme, že každá uživatelská funkce obsahuje dvě informace: svůj seznam parametrů a tělo. Při aplikaci funkce na argumenty se nejprve vytvoří prostředí, ve kterém se parametry funkce navážou na (vyhodnocené) argumenty a jehož předkem je globální prostředí, a pak se tělo funkce v tomto prostředí vyhodnotí.

Podívejme se znovu na uvedený příklad, tedy funkci `adder`:

```
(defun adder (add)
  (labels ((res (x) (+ x add)))
    #'res))
```

Výrazy `(adder 5)` a `(adder 10)` se oba vyhodnotí na funkci. Jak bude každá z těchto funkcí vypadat?

- Seznam parametrů obou funkcí je `(x)`,
- tělo obou funkcí je `((+ x add))`.

Funkce tedy mají stejný seznam parametrů a stejné tělo. Přesto vracejí různé výsledky. (Což si my samozřejmě přejeme.)

Je to zařízeno pomocí prostředí, ve kterém se tělo funkce (při její aplikaci) vyhodnocuje. (Prezentace k přednášce na tomto místě obsahuje víc informací, které pomohou tento závěr osvětlit.)

Upřesníme informaci o tom, co je výsledkem definice funkce. Výrazy, jejichž vyhodnocením vznikají nové funkce — tedy výrazy s operátorem `defun` nebo `labels`

— se stejně jako všechny ostatní výrazy vždy vyhodnocují v nějakém prostředí (a to prostředí je v ten moment tzv. **aktuální**). Každý z operátorů `defun` a `labels` pak určuje prostředí, kterému říkáme **prostředí vzniku funkce** (nebo prostředí, ve kterém funkce vznikla).

- U operátoru `defun` je to prostředí, ve kterém byl výraz s operátorem `defun` vyhodnocen (obvykle je to tedy globální prostředí), tj. **aktuální prostředí**.
- U operátoru `labels` je to **prostředí tímto operátorem vytvořené** (jehož je, jak víme, aktuální prostředí předkem).

Pokud se tedy vrátíme k naší definici funkce `adder`:

```
(defun adder (add)
  (labels ((res (x) (+ x add)))
    #'res))
```

pak prostředí vzniku funkce `adder` je prostředí, ve kterém byl tento `defun`-výraz vyhodnocen. To je obvykle **globální prostředí**, protože výrazy s operátorem `defun` obvykle vyhodnocujeme v Listeneru nebo je zapisujeme do zdrojového souboru, jak jsme zvyklí.

U lokální funkce `res` v tomto příkladě je prostředím vzniku prostředí vytvořené speciálním operátorem `labels`. Když například aplikujeme funkci `adder` na hodnotu `3`, tj. když vyhodnotíme výraz `(adder 3)`, bude to následující prostředí:

Globální prostředí

symbol	hodnota	typ vazby
`adder`	`#<Function ...>`	*funkční*
`pi`	`3.141592653589793D0`	*hodnotová*
⋮	⋮	⋮

Prostředí funkce `adder`

symbol	hodnota	typ vazby
`add`	3	*hodnotová*

Prostředí operátoru `labels`

symbol	hodnota	typ vazby
`res`	`#<Closure ... >`	*funkční*

Pro zopakování: každá funkce obsahuje následující informace

- seznam parametrů,
- tělo,

- prostředí vzniku.

Díky tomu můžeme vylepšit naše pravidla aplikace uživatelské funkce:

Aplikace uživatelské funkce

Při aplikaci uživatelské funkce se vytvoří nové prostředí s vazbami parametrů na hodnoty argumentů. Předkem tohoto prostředí se stane prostředí vzniku funkce. V tomto prostředí se pak vyhodnotí tělo. Nové prostředí se vytváří při každé aplikaci znovu.

Funkce, která nese informaci o prostředí svého vzniku (v Lispu je to každá), se nazývá *lexikální uzávěr* (stručně *uzávěr*).

Na prezentaci z přednášky je na tomto místě rozebraný příklad, který nám pomůže upravená pravidla pochopit a který ukazuje, že vedou k očekávaným výsledkům.

4 Anonymní funkce

Jak už jsem řekl, ne každá funkce musí mít název (být pojmenovaná). Funkce, které ho nemají, se nazývají *anonymní*. Takové funkce se nevytvářejí operátory `defun` nebo `labels`, protože ty vytvářejí funkce pojmenované.

K vytváření funkcí anonymních slouží makro `lambda`. Například vyhodnocením výrazu

```
(lambda (b h) (* 1/2 b h))
```

vznikne funkce, která počítá obsah trojúhelníka s danou základnou a výškou:

```
CL-USER 9 > (funcall (lambda (b h) (* 1/2 b h))
                     5
                     6)
15
```

Složený výraz, jehož prvním prvkem je symbol `lambda` se nazývá *λ-výraz*. Symbol `lambda` je makro, výraz musí mít nejméně jeden argument: *seznam parametrů*, což je seznam ve tvaru, který už známe. Za seznamem parametrů následuje libovolný počet výrazů, které dohromady tvoří *tělo*.

```
    seznam parametrů
(lambda (b h) . ((* 1/2 b h)))
                  tělo funkce
```

Vyhodnocením λ-výrazu vznikne nová funkce, jejíž seznam parametrů a tělo budou dány argumenty λ-výrazu. Prostředím vzniku nové funkce bude prostředí, ve kterém byl λ-výraz vyhodnocen.

Na začátku přednášky jsme si ukazovali funkci `sum-sequence` na sčítání prvků posloupnosti. Takto můžeme s pomocí anonymní funkce sečíst členy posloupnosti druhých mocnin přirozených čísel:

```
CL-USER 6 > (sum-sequence (lambda (x) (expt x 2)) 1 100)
338350
```

Pomocí makra `lambda` můžeme zjednodušit naši funkci `adder`. Funkce původně vracela funkci vzniklou jako lokální funkce `res` pomocí speciálního operátoru `labels`.

```
(defun adder (add)
  (labels ((res (x) (+ x add)))
    #'res))
```

Název funkce `res` ale není k ničemu potřeba, můžeme ji vytvořit jako funkci anonymní:

```
(defun adder (add)
  (lambda (x) (+ x add)))
```

Výsledkem musí být (a je) lexikální uzávěr, protože funkce si musí pamatovat vazbu symbolu `add` ze svého prostředí vzniku.

A konečně, funkce `scale-list` byla na minulé přednášce definována takto:

```
(defun scale-list (list factor)
  (labels ((prod (x) (* x factor)))
    (mapcar #'prod list)))
```

Takto ji můžeme přepsat pomocí anonymní funkce:

```
(defun scale-list (list factor)
  (mapcar (lambda (x) (* x factor))
          list))
```

5 Dynamické vytváření nových funkcí

Jak jsme si řekli minule, funkcí vyššího řádu rozumíme funkci, která je určena k práci s jinými funkcemi. Konkrétně může funkce přijímat jako argumenty, nebo je vracet jako výsledek. Dosud jsme se zabývali převážně prvním případem, jedinou výjimkou byla funkce `adder`. Nyní se podíváme podrobněji na případ druhý a předvedeme si několik funkcí vyššího řádu, které dynamicky (tj. za běhu programu) vytvářejí funkce nové. K tomu používají operátor `lambda`.

Příklady se budou týkat posloupností, kterými jsme tuto přednášku začali. Abychom mohli s posloupnostmi pracovat, napíšeme si nejprve funkci, která vytiskne několik prvních členů (řekněme 20) dané posloupnosti:

```
(defun print-sequence (seq)
  (labels ((iter (index)
             (if (>= index 20)
                 (princ "... ")
               (progn (princ (mem seq index))
                 (princ ", ")
                 (iter (1+ index))))))
    (iter 0)
    nil))
```

Funkci můžeme otestovat například na posloupnosti čtverců:

```
CL-USER 8 > (print-sequence (lambda (x) (* x x)))
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, ...
NIL
```

nebo na Fibonacciho posloupnosti:

```
CL-USER 9 > (print-sequence #'fib)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, ...
NIL
```

V dalším už nebudeme ukazovat, co funkce `print-sequence` kdy vytiskne, abychom zbytečně neplýtvali místem. Ale kdykoliv a pro libovolný příklad posloupnosti, o kterém bude dále řeč, si to můžete sami vyzkoušet.

Teď bude následovat několik příkladů, jak lze pomocí anonymních funkcí s posloupnostmi pracovat.

Posloupnost, jejíž všechny členy jsou stejné (tzv. *konstantní posloupnost*) můžeme pomocí funkce zadat snadno. Například posloupnost, jejíž všechny členy jsou 10, zadáme funkcí

```
(defun constantly-10 (n)
  10)
```

(Při kompilaci této a další funkce nám překladač nahlásí warning. To je nežádoucí, ale výjimečně to budeme ignorovat. V dalších semestrech se dozvíme, co v takové situaci dělat.)

Tak můžeme zadat konstantní posloupnost s libovolnou hodnotou jejích členů. To je ovšem trochu nešikovné. Podíváme se tedy po nějaké abstrakci, která umožní zadávat konstantní posloupnosti jednodušeji.

Napíšeme funkci, která požadovanou konstantní posloupnost vrátí jako svou hodnotu:

```
(defun my-constantly (c)
  (lambda (n) c))
```

(Ano, funkce `constantly` v Lispu už je.) Otestovat funkci `my-constantly` můžete například vyhodnocením výrazu

```
(print-sequence (my-constantly 5))
```

Součtem dvou posloupností rozumíme posloupnost, jejíž členy jsou součty příslušných členů původních posloupností:

```
(defun seq-+ (seq1 seq2)
  (lambda (n)
    (+ (mem seq1 n) (mem seq2 n))))
```

Posloupnost, jejíž člen o indexu n je dán předpisem

$$a_n = n^2 + 1,$$

je tedy reprezentována funkcí vzniklou vyhodnocením výrazu

```
(seq-+ #'power2 (constantly 1))
```

Obecná operace se dvěma posloupnostmi:

```
(defun seq-op (op seq1 seq2)
  (lambda (n)
    (funcall op (mem seq1 n) (mem seq2 n))))
```

Aritmetický průměr dvou posloupností:

```
(seq-op (lambda (x y) (/ (+ x y) 2)) seq1 seq2)
```

Hledání v posloupnosti. Funkce `first-index` vrací index prvního členu posloupnosti, který splňuje zadanou podmínku:

```
(defun first-index (seq condition)
  (labels ((iter (index)
             (if (funcall condition (mem seq index))
                 index
               (iter (1+ index)))))
    (iter 0)))
```

(Funkce se zacyklí, pokud žádný člen posloupnosti podmínku nesplňuje.)

Kolikáté Fibonacciho číslo je větší než 100?

```
(first-index #'fib (lambda (x) (> x 100)))
```

Kolikáté Fibonacciho číslo je dělitelné 14 (kromě nuly)?

```
(first-index #'fib (lambda (x)
                     (and (> x 0)
                          (= (rem x 14) 0))))
```

6 Příklad: bankovní účet

Vklady na účet

Vkládáme každý měsíc 10000 Kč na bankovní účet. Měsíce číslujeme od nuly, první vklad uděláme hned v měsíci č. 0 (při založení účtu). Posloupnost vkladů je dána funkcí vytvořenou výrazem

```
(constantly 10000.0)
```

Je to funkce, která pro každý měsíc vrátí 10000.0:

```
CL-USER 2 > (print-sequence (constantly 10000.0))
10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000.0,
10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000.0,
10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000.0, ...
NIL
```

Funkci si můžeme uložit do proměnné:

```
CL-USER 22 > (setf deposits (constantly 10000))
#<Closure 1 subfunction of SYSTEM::CONSTANTLY-AUX 40600039AC>
```

Předpokládejme, že nám banka začátkem následujícího měsíce připisuje úrok $\frac{1}{12}$ % ze zůstatku na účtu. Pro měsíc n je zůstatek B_n (B jako *balance*) na účtu tedy

$$B_0 = d_n$$
$$B_n = B_{n-1} \cdot (1 + i) + d_n \text{ pro } n > 0,$$

kde

B_n: zůstatek v měsíci n
i: měsíční úrok (např. $\frac{1}{12}$ %)
d_n: vklad v měsíci n (např. 10000 Kč)

Napíšeme funkci, která z posloupnosti vkladů vytvoří posloupnost zůstatků tím, že v každém měsíci připíše úroky:

```
(defun balances (deposits interest-rate)
  (lambda (month)
    (if (= month 0)
        (mem deposits 0)
      (+ (* (mem (balances deposits interest-rate) (- month 1))
            (+ interest-rate 1))
         (mem deposits month)))))
```

Funkce samozřejmě vrací jako výsledek posloupnost (to je to, co jsme chtěli). Tohle třeba je posloupnost zůstatků v našem příkladě, když je měsíční úroková míra rovna $\frac{1}{12}$ %:

```
CL-USER 12 > (setf balances (balances deposits 1/1200))
#<Closure 1 subfunction of BALANCES 406000083C>

CL-USER 13 > (print-sequence balances)
10000.0, 20008.334, 30025.008, 40050.03, 50083.41, 60125.15,
70175.26, 80233.74, 90300.61, 100375.87, 110459.52, 120551.58,
130652.05, 140760.94, 150878.25, 161003.99, 171138.16, 181280.78,
191431.86, 201591.39, ...
NIL
```

Za jak dlouho bude zůstatek alespoň milion?

```
(first-index balances (lambda (x) (>= x 1000000)))
```

Otázky a úkoly na cvičení

1. Co bude hodnotou následujícího výrazu?

   ```
   (funcall (lambda (x) (funcall x 10)) (lambda (x) 20))
   ```

2. Přepište výraz

   ```
   (let ((x a))
     tělo)
   ```

   tak, aby neobsahoval speciální operátory `let`, `let*` a přitom si zachoval týž (hlavní i vedlejší) efekt.

3. Vysvětlete podrobně, jaká prostředí vznikají při aplikaci funkce `scale-list` z minulé přednášky. Místo vestavěné funkce `mapcar` uvažujte naši uživatelsky definovanou funkci `my-mapcar`.

4. Totéž udělejte pro funkci `scale-list` z této přednášky.

5. Pomocí funkce `accumulate` napište funkci, která vrátí největší prvek části dané číselné posloupnosti ohraničené zadanými indexy.

6. Napište predikát `constant-seq-p`, který zjistí, zda prvních k zadaných členů dané posloupnosti je stejných. Funkce přijímá jako argumenty posloupnost a číslo k (abyste příklad pochopili, můžete si posloupnost vytisknout):

   ```
   CL-USER 19 > (constant-seq-p (lambda (n) (if (< n 20) 0 n))
                                20)
   T

   CL-USER 20 > (constant-seq-p (lambda (n) (if (< n 20) 0 n))
                                21)
   NIL
   ```

7. Napište predikát `increasing-seq?`, který zjistí, zda prvních k členů zadané posloupnosti roste, tj. zda každý člen je větší než předchozí. Predikát bude přijímat dva argumenty: posloupnost a číslo k.

8. Napište funkci `even-members`, která vrátí posloupnost obsahující pouze členy zadané posloupnosti se sudým indexem:

```
CL-USER 13 > (print-sequence (even-members #'identity))
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
34, 36, 38, ...
NIL

CL-USER 14 > (print-sequence (even-members #'fib))
0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711, 46368,
121393, 317811, 832040, 2178309, 5702887, 14930352, 39088169,
...
NIL

CL-USER 15 > (print-sequence (even-members (lambda (n) (if
(evenp n) 5 10))))
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
...
NIL
```

9. Funkce mohou reprezentovat i jiné objekty než posloupnosti. Například funkce *`tbl`* přijímající dva argumenty může představovat tabulku čísel s 10 řádky a 10 sloupci tak, že hodnota (`funcall` *`tbl row column`*) je číslo na řádku s indexem *`row`* a sloupci s indexem *`column`* (číslujeme jako obvykle od nuly). Napište predikát `zero-row-p`, který přijímá dva argumenty, a to funkci a index od 0 do 9, a zjistí, zda tabulka představovaná zadanou funkcí má na řádku se zadaným indexem samé nuly.

10. Tabulku reprezentujeme funkcí stejně jako v předchozím příkladě. Napište funkci `transpose-table`, která k zadané tabulce vrátí tabulku transponovanou, tj. s vyměněnými řádky a sloupci. Například k tabulce nalevo tedy vrátí tabulku napravo:

$$\begin{array}{cccccc} 0 & 1 & 2 & \cdots & 9 \\ 0 & 2 & 4 & \cdots & 18 \\ 0 & 3 & 6 & \cdots & 27 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 10 & 20 & \cdots & 90 \end{array} \qquad \begin{array}{cccccc} 0 & 0 & 0 & \cdots & 0 \\ 1 & 2 & 3 & \cdots & 10 \\ 2 & 4 & 6 & \cdots & 20 \\ \vdots & \vdots & \vdots & & \vdots \\ 9 & 18 & 27 & \cdots & 90 \end{array}$$