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**

# 7. Lexikální uzávěry

## 1 Posloupnosti

Minule jsme uvedli nový speciální operátor, který slouží k vytváření nových procedur, — speciální operátor `lambda`. Zopakujme si, jak vypadá výraz s tímto speciálním operátorem, tedy tzv. **λ-výraz**.

```
          seznam parametrů
(lambda (p1 p2 ... pn) expression)
                        tělo
```

- *seznam parametrů:* parametry vytvářené procedury
- *tělo:* tělo vytvářené procedury

Výsledkem vyhodnocení λ-výrazu je nová procedura.

Jedním z hlavních příkladů na minulé přednášce byl příklad na posloupnosti čísel. Budeme v něm ještě chvíli pokračovat. Připomeňme, že posloupnost můžeme *zadat* (*reprezentovat*) procedurou, která při aplikaci na číslo $n$ vrátí $n$-tý člen posloupnosti. (Čísla $n$ uvažujeme z oboru přirozených čísel včetně nuly.)

Posloupnost druhých mocnin přirozených čísel (včetně nuly) je například zadána vestavěnou procedurou `sqr` (tato procedura je sice vestavěná, ale víme, že bychom ji uměli sami naprogramovat). Posloupnost Fibonacciho čísel je dána již známou procedurou

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#true (+ (fib (- n 1)) (fib (- n 2))))))
```

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

Abychom mohli s posloupnostmi pracovat, napíšeme si nejprve proceduru, která vytiskne několik prvních členů (řekněme 20) dané posloupnosti:

```
(define (display-sequence seq)
  (begin (display-seq-iter seq 0 20)
         (newline)))

(define (display-seq-iter seq index count)
  (if (>= index count)
      (display "...")
      (begin (display (seq index))
             (display ", ")
             (display-seq-iter seq (+ index 1) count))))
```

Proceduru můžeme otestovat například na posloupnosti čtverců:

```
> (display-sequence sqr)
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, ...
```

nebo na Fibonacciho posloupnosti:

```
> (display-sequence fib)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, ...
```

V dalším už nebudeme ukazovat, co procedura `display-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 procedur s posloupnostmi pracovat. Něco jsme viděli už na minulé přednášce. K tomu se nebudeme vracet a ukážeme si raději něco nového.

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

```
(define (constantly-10 n) 10)
```

Jak víme z minula, proceduru můžeme definovat i pomocí $\lambda$-výrazu:

```
(define constantly-10
  (lambda (n) 10))
```

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 proceduru, která požadovanou konstantní posloupnost vrátí jako svou hodnotu:

```
(define (constantly c)
  (lambda (n) c))
```

Otestovat proceduru `constantly` můžete například vyhodnocením výrazu

```
(display-sequence (constantly 5))
```

Součet dvou posloupností:

```
(define (seq-+ seq1 seq2)
  (lambda (n)
    (+ (seq1 n) (seq2 n))))
```

Posloupnost, jejíž $n$-tý člen je dán předpisem

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

je tedy reprezentována procedurou `(seq-+ sqr (constantly 1))`.

Obecná operace se dvěma posloupnostmi:

```
(define (seq-op seq1 seq2 op)
  (lambda (n)
    (op (seq1 n) (seq2 n))))
```

Aritmetický průměr dvou posloupností

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

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

```
(define (first-index seq condition)
  (first-index-iter seq condition 0))

(define (first-index-iter seq condition index)
  (if (condition (seq index))
      index
      (first-index-iter seq condition (+ index 1))))
```

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)
                        (= (remainder x 14) 0))))
```

# 2 Příklad: bankovní účet

### Vklady na účet

Vkládáme každý měsíc 10000 Kč na bankovní účet. Měsíce číslujeme od nuly, první vklad uděláme v měsíci č. 1. Posloupnost vkladů je dána procedurou vytvořenou výrazem

```
(constantly 10000)
```

Je to procedura, která pro každý měsíc vrátí 10000:

```
> (display-sequence (constantly 10000))
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, ...
```

Proceduru bychom samozřejmě mohli i pojmenovat:

```
(define deposits (constantly 10000))
```

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

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

kde

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

Procedura, která pro daný měsíc vypočítá zůstatek na účtu, vychází přímo z uvedeného vzorce:

```
(define (balances-1 month)
  (if (= month 0)
      0
      (+ (* (balances-1 (- month 1))
            (+ #i1/1200 1))
         (deposits month))))
```

Procedura samozřejmě definuje posloupnost (to je to, co jsme chtěli). Bude tak k použití s procedurami, které obecně s posloupnostmi pracují. Například se můžeme podívat, kdy náš zůstatek splní nějakou podmínku, třeba převýší zadanou hodnotu:

Za jak dlouho převýší zůstatek 1000000?

```
(first-index balances-1 (lambda (x) (> x 1000000)))
```

Procedura `balances-1` ovšem pracuje jen s naším konkrétním účtem s konkrétním měsíčním vkladem a úročením. Co kdybychom ji chtěli zobecnit, aby pracovala s libovolným účtem?

To můžeme udělat tak, že proceduru změníme, aby nereprezentovala posloupnost zůstatků, ale aby pro danou posloupnost měsíčních vkladů a dané úročení posloupnost zůstatků **vytvořila**. Tedy aby vytvořila proceduru, která posloupnost zůstatků definuje:

```
(define (balances deposits interest)
  (lambda (month)
    (if (= month 0)
        0
        (+ (* ((balances deposits interest) (- month 1))
              (+ interest 1))
           (deposits month)))))
```

Testy procedury:

```
> (balances deposits #i1/1200)
#<procedure:...R1/07 src/07.rkt:12:2>
> ((balances deposits #i1/1200) 10)
#i100375.8345498272
> ((balances (lambda (month) 5000) #i1/1200) 10)
#i50187.9172749136
```

# 3 Lexikální uzávěr

Když se podíváme podrobněji na příklady, které jsme si uvedli, dojdeme k překvapivému závěru, že by vlastně neměly fungovat. Aspoň podle toho, co zatím víme o procedurách. Zopakujme si to stručně:

Každá uživatelská procedura si pamatuje svůj seznam parametrů a tělo. Při aplikaci procedury na argumenty se nejprve vytvoří prostředí, ve kterém se parametry procedury navážou na argumenty, a pak se tělo procedury v tomto prostředí vyhodnotí.

Podívejme se na nejjednodušší příklad, který jsme si ukázali na začátku, a to na proceduru `constantly`:

```
(define (constantly c)
  (lambda (n) c))
```

Výrazy `(constantly 5)` a `(constantly 10)` se oba vyhodnotí na anonymní proceduru. Jak bude každá z těchto procedur vypadat?

- Seznam parametrů obou anonymních procedur je `(n)`,
- tělo obou procedur je `c`.

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

Je to zařízeno pomocí prostředí, ve kterém se tělo anonymní procedury (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 vyhodnocení $\lambda$-výrazu. Jak víme, $\lambda$-výrazy mají následující tvar:

```
          seznam parametrů
(lambda (p1 p2 ... pn) expression )
                        tělo
```

$\lambda$-výrazy (stejně jako všechny ostatní výrazy) se vždy vyhodnocují v nějakém prostředí. Hodnotou $\lambda$-výrazu v daném prostředí je anonymní procedura, která si kromě seznamu parametrů a těla pamatuje i toto prostředí. Nazýváme ho *prostředím, ve kterém daná procedura vznikla*, neboli *prostředím vzniku procedury.*

Pro zopakování: každá procedura si pamatuje

- své parametry,
- své tělo,

- prostředí, ve kterém vznikla.

Díky tomu můžeme vylepšit naše pravidla aplikace uživatelské procedury:

**Aplikace uživatelské procedury**

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

Procedura, která nese informaci o prostředí svého vzniku (ve Scheme je to každá), se nazývá *lexikální uzávěr*.

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

# 4 Lokální procedury

Díky uvedeným poznatkům teď můžeme začít při programování procedur používat tzv. *lokální procedury*. Podívejme se například na proceduru

```
(define (first-index seq condition)
  (first-index-iter seq condition 0))

(define (first-index-iter seq condition index)
  (if (condition (seq index))
      index
      (first-index-iter seq condition (+ index 1))))
```

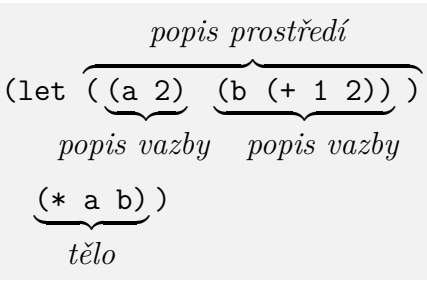
Chtěli bychom se vyhnout nutnosti psát jednoúčelovou pojmenovanou proceduru `first-index-iter`, kterou beztak používáme jen uvnitř procedury `first-index`. Navrhneme tedy následující úpravu:

```
(define (first-index seq condition)
  (let ((iter (lambda (index)
                (if (condition (seq index))
                    index
                    (iter (+ index 1))))))
    (iter 0)))
```

Bude tato úprava fungovat? Abychom to pochopili, musíme si jednak uvědomit, v jakém prostředí se bude vyhodnocovat tělo pomocné procedury `iter` a jednak, v jakém prostředí tato procedura vznikla.

K tomu je dobré si zopakovat, jak přesně pracuje speciální operátor `let`:

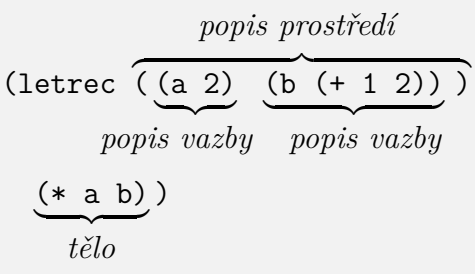
**Speciální operátor `let`: vyhodnocení**




1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.
2. Vytvoří se nové prostředí a v něm vazby tak, že každá první položka popisu vazby (která musí být symbolem) se naváže na hodnotu druhé položky.
3. Předkem nového prostředí se učiní aktuální prostředí.
4. Tělo se vyhodnotí v tomto novém prostředí. Výsledek se vrátí jako hodnota celého výrazu.

Pokud jste tento popis opravdu pochopili, je vám jasné, že naše řešení fungovat nebude. (Můžete si to samozřejmě i vyzkoušet.) K účelu vytvoření lokální procedury je připraven speciální operátor `letrec`.

**Speciální operátor `letrec`: vyhodnocení**




1. Vytvoří se nové prostředí.
2. Předkem nového prostředí se učiní aktuální prostředí.
3. V novém prostředí se vyhodnotí všechny druhé položky popisů vazeb a do prostředí se vždy přidá vazba příslušného symbolu na získanou hodnotu.
4. Tělo se vyhodnotí v tomto novém prostředí. Výsledek se vrátí jako hodnota celého výrazu.

Pomocí něj už lze proceduru `first-index` napsat tak, aby používala lokální proceduru:

```
(define (first-index seq condition)
  (letrec ((iter (lambda (index)
                   (if (condition (seq index))
                       index
                       (iter (+ index 1))))))
    (iter 0)))
```

Pro úplnost nyní dodáme ještě jeden speciální operátor, který je předchozím dvěma podobný:

### Speciální operátor `let*`: vyhodnocení

```
                 popis prostředí
(let* ( (a 2)   (b (+ a 2)) )
      popis vazby  popis vazby
  (* a b) )
    tělo
```

Tento výraz se vyhodnotí jako

```
(let ((a 2))
  (let ((b (+ a 2)))
    (* a b)))
```

# Otázky a úkoly na cvičení

1. Napište proceduru `seq-even-members`, která vrátí k dané posloupnosti novou posloupnost, která bude složena ze členů původní posloupnosti se sudým indexem:

```
> (display-sequence (seq-even-members sqr))
0, 4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676,
784, 900, 1024, 1156, 1296, 1444, ...
> (display-sequence (seq-even-members fib))
0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711, 46368,
121393, 317811, 832040, 2178309, 5702887, 14930352, 39088169,
...
```

2. Napište proceduru `seq-shift`, která k dané posloupnosti vrátí novou („posunutou“) posloupnost, jejíž členy začínají členem původní posloupnosti zadaného indexu. Například

```
> (display-sequence (seq-shift sqr 4))
16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361, 400, 441, 484, 529, ...
```

3. Vylepšete proceduru `seq-shift` tak, aby jako druhý argument (index) akceptovala i zápornou hodnotu a přidejte jí další parametr, který bude určovat defaultní hodnotu pro členy, které v původní posloupnosti nejsou:

```
> (display-sequence (seq-shift sqr -10 -1))
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, 1, 4, 9, 16, 25,
36, 49, 64, 81, ...
```

4. Bylo by možné naprogramovat Fibonacciho posloupnost jako součet dvou vhodně posunutých Fibonacciho posloupností? Toto sampzřejmě nebude fungovat:

```
(define fib
  (seq-+ (seq-shift fib -2 0)
         (seq-shift fib -1 1)))
```

5. Přepište několik rekurzivních procedur, které používají pomocnou proceduru (většinou jsou to procedury, které generují iterativní výpočetní proces), tak, aby místo ní používaly lokální proceduru. Můžete začít procedurou `display-sequence`.