Paradigmata programování 1

Rekurze a indukce

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 8

Přednáška 8: Přehled

1. Rekurze a indukce jako obecné (matematické) principy
   - rekurzivní definice matematických funkcí a procedur
   - princip matematické indukce
   - strukturální rekurze a strukturální indukce
2. Lineární rekurzivní a iterativní výpočetní proces
   - rekurzivní procedury a rekurzivní výpočetní procesy
   - lineární rekurzivní výpočetní proces
   - koncová rekurze, lineární iterativní výpočetní proces
3. Příklady rekurzivních procedur
   - efektivní mocnění čísel, . . .
   - práce se seznamy, . . .

Co jsou rekurze a indukce?

Rekurze – mnoho významů

- pro nás – metoda definice (matematických) funkcí a procedur,
- jiný význam – viz kursy *Vyčíslitelnost a složitost*, *Matematická logika*

Indukce – dokazovací princip

- *matematická indukce* – přes přirozená čísla
- *strukturální indukce* – obecnější princip (indukce přes „strukturu seznamu“)

Poznámky:

- rekurze + indukce . . . obecné principy (nejen Scheme)
- rekurzivní procedura = **procedura**, která **aplikuje sebe sama**
- znalost a pochopení patří k „malé násobilce informatika“

Motivace: Faktoriál

Definice $n!$

$$n! = \begin{cases} 1 & \text{pokud } n \leq 1, \\ n \cdot (n-1)! & \text{jinak.} \end{cases}$$

Hodnoty faktoriálu pro $n = 0, 1, 2, 3, 4, \ldots$

$$\begin{aligned}
0! &= 1 \\
1! &= 1 \\
2! &= 2 \cdot 1! = 2 \cdot 1 = 2 \\
3! &= 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 = 6 \\
4! &= 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \\
\vdots \;\; & \;\; \vdots
\end{aligned}$$

Otázka: Jsou správné pro každé n?

Motivace: Fibonacciho čísla

Definice F_n

$$F_n = \begin{cases} 0 & \text{pokud } n = 0, \\ 1 & \text{pokud } n = 1, \\ F_{n-1} + F_{n-2} & \text{jinak.} \end{cases}$$

Hodnoty F_n pro $n = 0, 1, 2, 3, 4, \ldots$

$$\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= F_1 + F_0 = 0 + 1 = 1 \\
F_3 &= F_2 + F_1 = (F_1 + F_0) + F_1 = (1 + 0) + 1 = 2 \\
F_4 &= F_3 + F_2 = (F_2 + F_1) + (F_1 + F_0) = \\
&= ((F_1 + F_0) + F_1) + (F_1 + F_0) = ((1 + 0) + 1) + (1 + 0) = 3 \\
&\vdots \qquad \vdots
\end{aligned}$$

Otázka: Jsou správné pro každé n?

Matematická indukce

Označení: $P(n)$: číslo n má vlastnost P

Věta (princip indukce přes přirozená čísla)

K tomu abychom ověřili, že vlastnost P platí pro každé $n \in \mathbb{N}_0$, stačí prokázat platnost následujících dvou bodů:

1. *platí $P(0)$,*
2. *pokud platí $P(i)$, pak platí $P(i+1)$.*

Důkaz.

Sporem. Nechť platí 1 a 2 a vezměme nejmenší přirozené číslo, které nemá vlastnost P. Buď $n = 0$ – spor. Nebo $P(n-1)$ – spor. $\square$

Příklad: $P(n)$: „hodnota $n!$ je jednoznačně definovaná“.

Strukturální rekurze a indukce – principy nejen pro čísla

Uvažujme množiny:

- $\mathcal{L}_{()}$ ... množina všech seznamů,
- $\mathcal{L}$ ... množina všech neprázdných seznamů,
- $\mathcal{E}$ ... množina všech elementů jazyka Scheme, ...

Konstruktory a selektory seznamů jako zobrazení:

$$\begin{aligned} \mathrm{cons}\colon \mathcal{E} \times \mathcal{L}_{()} &\to \mathcal{L} \\ \mathrm{car}\colon \mathcal{L} &\to \mathcal{E} \\ \mathrm{cdr}\colon \mathcal{L} &\to \mathcal{L}_{()} \end{aligned}$$

Jak chápat?

$$\mathrm{cons}(d, \mathrm{cons}(c, \mathrm{cons}(a, b)))$$
$$\mathrm{cons}(\mathrm{cons}(x, z), \mathrm{cons}(c, \mathrm{cons}(a, b)))$$
$$\vdots$$

Motivace: délka seznamu

Definice $\mathrm{length}\colon \mathcal{L}_{()} \to \mathbb{N}_0$

$$\mathrm{length}(l) = \begin{cases} 0 & \text{pokud } l \text{ je prázdný seznam,} \\ 1 + \mathrm{length}(\mathrm{cdr}(l)) & \text{jinak.} \end{cases}$$

Myšlenka: každý seznam l je buď (i) prázdný, nebo (ii) $\mathrm{cdr}(l)$ je (jednodušší) seznam.

Hodnoty $\mathrm{length}(l)$:

$$\begin{aligned}
\mathrm{length}(\texttt{()}) &= 0 \\
\mathrm{length}(\texttt{(a)}) &= 1 + \mathrm{length}(\mathrm{cdr}(\texttt{(a)})) = 1 + \mathrm{length}(\texttt{()}) = 1 + 0 = 1 \\
\mathrm{length}(\texttt{(a b)}) &= 1 + \mathrm{length}(\mathrm{cdr}(\texttt{(a b)})) = 1 + \mathrm{length}(\texttt{(b)}) = \\
&= 1 + (1 + \mathrm{length}(\mathrm{cdr}(\texttt{(b)}))) = 1 + (1 + \mathrm{length}(\texttt{()})) = \\
&= 1 + (1 + 0) = 2 \\
&\;\;\vdots
\end{aligned}$$

Motivace: spojení dvou seznamů

Definice append2: $\mathcal{L}_{()} \times \mathcal{L}_{()} \to \mathcal{L}_{()}$

$$\mathrm{append2}(l,k) = \begin{cases} k & \text{pokud } l \text{ je prázdný seznam,} \\ \mathrm{cons}(\mathrm{car}(l), \mathrm{append2}(\mathrm{cdr}(l), k)) & \text{jinak.} \end{cases}$$

$$\begin{aligned}
\mathrm{append2}(\texttt{(a b)}, \texttt{(1 2 3)}) &= \\
&= \mathrm{cons}(\mathrm{car}(\texttt{(a b)}), \mathrm{append2}(\mathrm{cdr}(\texttt{(a b)}), \texttt{(1 2 3)})) \\
&= \mathrm{cons}(\texttt{a}, \mathrm{append2}(\texttt{(b)}, \texttt{(1 2 3)})) \\
&= \mathrm{cons}(\texttt{a}, \mathrm{cons}(\mathrm{car}(\texttt{(b)}), \mathrm{append2}(\mathrm{cdr}(\texttt{(b)}), \texttt{(1 2 3)}))) \\
&= \mathrm{cons}(\texttt{a}, \mathrm{cons}(\texttt{b}, \mathrm{append2}(\texttt{()}, \texttt{(1 2 3)}))) \\
&= \mathrm{cons}(\texttt{a}, \mathrm{cons}(\texttt{b}, \texttt{(1 2 3)})) \\
&= \mathrm{cons}(\texttt{a}, \texttt{(b 1 2 3)}) \\
&= \texttt{(a b 1 2 3)}.
\end{aligned}$$

Strukturální indukce (přes seznamy)

Věta (princip strukturální indukce přes seznamy)

K tomu abychom ověřili, že vlastnost P platí pro každý seznam $l \in \mathcal{L}_{()}$, stačí prokázat platnost následujících dvou bodů:

1. *platí $P(())$, to jest P platí pro prázdný seznam,*
2. *pokud platí $P(l)$, pak pro každý element e platí $P(\mathrm{cons}(e, l))$.*

Důkaz.

Sporem. Nechť platí 1 a 2 a vezměme nejkratší seznam l, který nemá vlastnost P (existuje obecně víc takových seznamů). Buď l je prázdný – spor. Nebo existuje ostře kratší seznam, který má vlastnost P – spor. $\square$

Příklad: $P(l)$: „hodnota $\mathrm{length}(l)$ je rovna délce seznamu“.

Rekurzivní procedury

Proceduře budeme říkat **rekurzivní procedura,** pokud při vyhodnocení jejího těla dochází (v některých případech) k aplikaci sebe sama. Aplikaci „sebe sama“ budeme dále nazývat **rekurzivní aplikace procedury.**

Základní části rekurzivní procedury:

1. **limitní podmínka rekurze** – podmínka, po jejímž splnění je vyhodnocen výraz jež *nezpůsobí* další aplikaci rekurzivní procedury (může jich být víc)
   - vymezuje *triviální případy aplikace procedury,*
   - např.: $0! = 1$, $\mathrm{length}(())= 0$
2. **předpis rekurze** – část těla procedury, při jejímž vyhodnocení *dochází k rekurzivní aplikaci procedury* (může jich být víc)
   - vyjadřuje, jak je stanoven výsledek aplikace rekurzivní procedury pomocí rekurzivní aplikace téže procedury s jinými (jednoduššími) argumenty
   - např.: $n! = n \cdot (n-1)!$, $\mathrm{length}(l) = 1 + \mathrm{length}(\mathrm{cdr}(l))$

Rekurzivní procedura pro výpočet n-té mocniny

$$x^n = \begin{cases} 1 & \text{pokud } n = 0, \\ x \cdot x^{n-1} & \text{jinak.} \end{cases}$$

```
(define expt
  (lambda (x n)
    (if (= n 0)
        1
        (* x (expt x (- n 1))))))
```

Proč funguje?

- aplikace expt z těla expt je možná (ve smyslu lexikálního rozsahu platnosti)
- každá rekurzivní aplikace vytváří nové prostředí

Složitost (za předpokladu, že aritmetické operace jsou elementární):

- časová $O(n)$,
- prostorová $O(n)$.

Příklad (Schematický průběh výpočtu expt)

`(expt 8 4)`	vyvolání 1. aplikace procedury
`(* 8 (expt 8 3))`	**navíjení**: vyvolání 2. aplikace
`(* 8 (* 8 (expt 8 2)))`	**navíjení**: vyvolání 3. aplikace
`(* 8 (* 8 (* 8 (expt 8 1))))`	**navíjení**: vyvolání 4. aplikace
`(* 8 (* 8 (* 8 (* 8 (expt 8 0)))))`	**navíjení**: vyvolání 5. aplikace
`(* 8 (* 8 (* 8 (* 8 1))))`	dosažení **limitní podmínky**
`(* 8 (* 8 (* 8 8)))`	stav po **odvinutí** 4. aplikace
`(* 8 (* 8 64))`	stav po **odvinutí** 3. aplikace
`(* 8 512)`	stav po **odvinutí** 2. aplikace
`4096`	**výsledná hodnota**

Fáze navíjeni a odvíjení

1. **fáze navíjení** – první fáze výpočetního procesu:
   - dochází k postupné rekurzivní aplikaci
   - jsou vytvářena nová prostředí v nichž jsou uloženy informace o vazbách formálních argumentů rekurzivně aplikované procedury
   - prostředí v sobě udržují informaci o pomyslném *odloženém výpočtu*
   - fáze navíjení končí dosažením limitní podmínky rekurze
2. **fáze odvíjení** – druhá fáze výpočetního procesu:
   - nastává po dosažení limitní podmínky rekurze
   - dochází k dokončení vyhodnocení těla procedury v prostředích, která vznikla v předchozí fázi navíjení
   - postupuje se zpětně
   - po dokončení fáze odvíjení je vrácen celkový výsledek
   - během fáze odvíjení může v některých případech *znovu nastat fáze navíjení*

Rekurzivní výpočetní procesy

Definice (rekurzivní výpočetní proces)

Výpočetní proces nazýváme **rekurzivní výpočetní proces,** pokud se jedná o proces *generovaný rekurzivní procedurou* nebo *několika procedurami, které se vzájemně aplikují,* u kterého lze rozlišit fáze **navíjení** a **odvíjení** (může být triviální).

Příklad (Vzájemně se aplikující procedury)

```
(define fa (lambda (n) (if (<= n 1) 1 (* n (fb (- n 1))))))
(define fb (lambda (n) (if (<= n 1) 1 (* n (fa (- n 1))))))
```

Důležité:

- rekurzivní procedura $\times$ rekurzivní výpočetní proces
- kvalitativně různé rekurzivní výpočetní procesy (náročnost na zdroje)

Výpočet n-té mocniny rychleji

Vyjádření x^n v závislosti na tvaru exponentu:

$$x^n = \begin{cases} 1 & \text{pokud } n = 0, \\ (x^{\frac{n}{2}})^2 & \text{pokud je } n \text{ sudé,} \\ x \cdot x^{n-1} & \text{pokud je } n \text{ liché.} \end{cases}$$

Odpovídající kód:

```
(define na2
  (lambda (x) (* x x)))

(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

Příklad (Průběh výpočtu rychlé verze expt)

```
(expt 2 25)
(* 2 (expt 2 24))
(* 2 (na2 (expt 2 12)))
(* 2 (na2 (na2 (expt 2 6))))
(* 2 (na2 (na2 (na2 (expt 2 3)))))
(* 2 (na2 (na2 (na2 (* 2 (expt 2 2))))))
(* 2 (na2 (na2 (na2 (* 2 (na2 (expt 2 1)))))))
(* 2 (na2 (na2 (na2 (* 2 (na2 (* 2 (expt 2 0))))))))
(* 2 (na2 (na2 (na2 (* 2 (na2 (* 2 1)))))))
(* 2 (na2 (na2 (na2 (* 2 (na2 2))))))
(* 2 (na2 (na2 (na2 (* 2 4)))))
(* 2 (na2 (na2 (na2 8))))
(* 2 (na2 (na2 64)))
(* 2 (na2 4096))
(* 2 16777216)
33554432
```

Analýza složitosti expt

$$x^n = \begin{cases} 1 & \text{pokud } n = 0, \\ (x^{\frac{n}{2}})^2 & \text{pokud je } n \text{ sudé}, \\ x \cdot x^{n-1} & \text{pokud je } n \text{ liché}. \end{cases}$$

Nejprve analyzujeme „optimistický případ“ (samé sudé exponenty až na 1): během první, druhé, třetí . . . aplikace vypočteny x^0, x^1, x^2, x^4, x^8, x^{16}, x^{32}

Obecně: k-tá aplikace expt ($k \geq 2$) vypočte hodnotu $x^{2^{k-2}}$

Při kolikáté aplikaci je vypočtena hodnota x^n?

$$x^n = x^{2^{k-2}}, \qquad k = 2 + \frac{\log n}{\log 2} = 2 + \log_2 n.$$

Nejhorší případ pro n: sudý exponent střídá lichý

Počet kroků je dvojnásobkem $2 + \log_2 n$.

Asymptotická časová složitost v nejhorším případě $O(\log_2 n)$.

Lineární rekurzivní výpočetní proces

lineární rekurzivní výpočetní proces – má *netriviální fáze navíjení a odvíjení*

- během fáze navíjení je budována „série odložených výpočtů“, přitom *počet prostředí roste* (*nejvýš*) *lineárně* vzhledem k velikosti vstupních argumentů
- po dosažení limitní podmínky nastává fáze odvíjení, ve které je zpětně dokončeno vyhodnocování výrazů, které bylo započato a „odloženo“ ve fází navíjení

Charakteristická vlastnosti:

- činnost lineárně rekurzivního výpočetního proces „nelze přerušit“,
- není (jednoduše) možný „výskok z rekurze“ (ve skutečnosti možný je).

Kam se nedokončené výpočty ukládají?

- abstraktní interpret Scheme – uchovány v prostředích jednotlivých aplikací,
- obecně – zásobník s aktivačními záznamy (C, Java a podobně).

```scheme
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

```scheme
(define expt
  (lambda (x n)    ←— velmi neefektivní; proč?
    (cond ((= n 0) 1)
          ((even? n) (* (expt x (/ n 2)) (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

```scheme
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((even? n) (let ((result (expt x (/ n 2))))
                       (* result result)))
          (else (* x (expt x (- n 1)))))))
```

Jak dále (ne)urychlit?

```
(define expt
  (lambda (x n)
    (cond ((= n 0) 1)
          ((= (modulo n 3) 0) (na3 (expt x (/ n 3))))
          ((even? n) (na2 (expt x (/ n 2))))
          (else (* x (expt x (- n 1)))))))
```

Pomůže?

I kdyby byl pro n počet kroků $\log_3 n$ (hodně optimistické), tak:

$$\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{1}{\log_2 3} \cdot \log_2 n \approx 0.6309 \log_2 n$$

Takže pořád řádově v $O(\log_2 n)$, ... proto značíme $O(\log_n)$.

Vylepšení tohoto typu „nestojí za námahu".

Příklad (Motivace pro iterativní výpočetní proces)

Nový pohled na výpočet $n!$:

pro $f(n,k) = \begin{cases} k & \text{pokud } n \leq 1, \\ f(n-1, k \cdot n) & \text{jinak,} \end{cases}$ máme $f(n,k) = k \cdot n!$

```
(define fac-iter
  (lambda (i accum)
    (if (<= i 1)
        accum
        (fac-iter (- i 1) (* accum i)))))

(define fac
  (lambda (n)
    (fac-iter n 1)))
```

Jak vypadají prostředí?

Koncově rekurzivní procedury a jejich aplikace

Definice (koncová pozice, koncově rekurzivní procedura)

Množina **koncových pozic λ-výrazu** Λ je definována následovně:

1. poslední výraz v těle výrazu Λ je v koncové pozici výrazu Λ;
2. je-li `(if` $\langle test \rangle$ $\langle důsledek \rangle$ $\langle náhradník \rangle$`)` v koncové pozici výrazu Λ, pak $\langle důsledek \rangle$ i $\langle náhradník \rangle$ (pokud je přítomen) je v koncové pozici výrazu Λ;
3. analogicky pro `cond`, `and` a `or`, ...

Koncová aplikace procedury vzniklé vyhodnocením λ-výrazu Λ je aplikace vyvolaná z *koncové pozice výrazu* Λ.

Rekurzivní procedura se nazývá **koncově rekurzivní,** pokud aplikuje sebe sama pouze z *koncových pozic* λ-výrazu jehož vyhodnocením vznikla.

Aplikace z koncových pozic

Optimalizace na koncovou rekurzi

- angl. *tail-recursion optimization* (TRO)
- při aplikaci koncově rekurzivních procedur se nevytvářejí nová prostředí
- jazyk Scheme vyžaduje ve své specifikaci
- obecný pojem (nejen Scheme, dále třeba Haskell)

Praktický význam:

- (pouze) na úrovni programovacího jazyka je nepodstatné,
- na úrovni interpretu/překladače je podstatné,
- důsledek – aplikace z koncových pozic je „velmi levná“

Příklad (Průběh výpočetního procesu pro fac-iter)

```
(define fac-iter
  (lambda (i accum)
    (if (<= i 1)
        accum
        (fac-iter (- i 1) (* accum i)))))
```

(fac 5)	
(fac-iter 5 1)	**navíjení**: vyvolání 1. aplikace
(fac-iter 4 5)	**navíjení**: vyvolání 2. aplikace
(fac-iter 3 20)	**navíjení**: vyvolání 3. aplikace
(fac-iter 2 60)	**navíjení**: vyvolání 4. aplikace
(fac-iter 1 120)	**navíjení**: vyvolání 5. aplikace
120	**dosažení limitní podmínky a vrácení hodnoty**

Lineární iterativní výpočetní proces

lineární iterativní výpočetní proces – výpočetní proces generovaný koncově rekurzivními procedurami

Základní rysy:

- nedochází při něm k vytváření odložených výpočtů
- během výpočtu může být přerušen a posléze opět obnoven
- argumenty lze rozdělit na **střádače** a **čítače**
- hraje analogickou roli jako cyklus v procedurálních jazycích

Během své činnosti *jednoznačně určen*:

1. vazbami symbolů v prostředí $\mathcal{P}$ svého běhu,
2. předpisem, jak změnit stav vazeb v $\mathcal{P}$ na základě aktuálních vazeb,
3. limitní podmínkou ukončující iterativní proces.

Příklad (Další příklady)

Problém výpočtu n-té mocniny: $x^{2n} = (x^n)^2 = (x^2)^n$

```
(define expt-iter
  (lambda (x n accum)
    (cond ((= n 0) accum)
          ((even? n) (expt-iter (* x x) (/ n 2) accum))
          (else (expt-iter x (- n 1) (* accum x))))))

(define expt
  (lambda (x n)
    (expt-iter x n 1)))
```

Složitost (za předpokladu, že aritmetické operace jsou elementární):

- časová $O(\log n)$,
- prostorová $O(1)$.

Příklad (Průběh výpočtu `expt-iter`)

`(expt 2 25)`	
`(expt-iter 2 25 1)`	**navíjení**: vyvolání 1. aplikace
`(expt-iter 2 24 2)`	**navíjení**: vyvolání 2. aplikace
`(expt-iter 4 12 2)`	**navíjení**: vyvolání 3. aplikace
`(expt-iter 16 6 2)`	**navíjení**: vyvolání 4. aplikace
`(expt-iter 256 3 2)`	**navíjení**: vyvolání 5. aplikace
`(expt-iter 256 2 512)`	**navíjení**: vyvolání 6. aplikace
`(expt-iter 65536 1 512)`	**navíjení**: vyvolání 7. aplikace
`(expt-iter 65536 0 33554432)`	**navíjení**: vyvolání 8. aplikace
`33554432`	**dosažení limitní podmínky**

Definice (speciální forma `let`, *pojmenovaná verze*)

Speciální forma *pojmenovaný* `let` se používá se třemi nebo více argumenty ve tvaru

```
(let ⟨jméno⟩ ((⟨symbol_1⟩ ⟨výraz_1⟩)
                  ⋮
              (⟨symbol_n⟩ ⟨výraz_n⟩))
  ⟨tělo_1⟩ ··· ⟨tělo_k⟩),
```

kde $\langle jméno \rangle$ je symbol (ostatní jako u `let`). Její aplikace je ekvivalentní:

```
(let ()
  (define ⟨jméno⟩
    (lambda (⟨symbol_1⟩ ··· ⟨symbol_n⟩)
      ⟨tělo_1⟩ ··· ⟨tělo_k⟩))
  (⟨jméno⟩ ⟨výraz_1⟩ ··· ⟨výraz_n⟩))
```

Příklad (Iterativní verze expt pomocí pojmenovaného let)

```
(define expt
  (lambda (x n)
    (let iter ((x x)
               (n n)
               (accum 1))
      (cond ((= n 0) accum)
            ((even? n) (iter (* x x) (/ n 2) accum))
            (else (iter x (- n 1) (* accum x)))))))
```

Poznámky:

- pojmenovaný `let` lze použít obecně pro jednorázovou aplikaci
- procedura nemusí být (jen) iterativní

Příklad (Rekurze na seznamech: procedura length)

```
(define length
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l))))))

(define length
  (lambda (l)
    (let iter ((l l)
               (steps 0))
      (if (null? l)
          steps
          (iter (cdr l) (+ steps 1))))))
```

Příklad (Procedury append a list-ref)

```
(define append2
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append2 (cdr l1) l2)))))

(define list-ref
  (lambda (l index)
    (if (= index 0)
        (car l)
        (list-ref (cdr l) (- index 1)))))
```

Příklad (Procedury map1 a build-list)

```
(define map1
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l))
              (map1 f (cdr l))))))

(define build-list
  (lambda (n f)
    (let build-next ((i 0))
      (if (= i n)
          '()
          (cons (f i) (build-next (+ i 1)))))))
```

Příklad (Procedury rev-append2 a reverse)

```
(define rev-append2
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (rev-append2 (cdr l1)
                     (cons (car l1) l2)))))

(define reverse
  (lambda (l)
    (rev-append2 l '())))
```