Paradigmata programování 1

Hloubková rekurze na seznamech

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 9

Přednáška 9: Přehled

1. Metody zastavení rekurze
   - podmínené výrazy využívající `if` a `cond`
   - podmínené výrazy využívající `and` a `or`
   - zastavení rekurze u vzájemně se aplikujících procedur
2. Stromová rekurze
   - stromově rekurzivní výpočetní proces
   - metody stanovení složitosti
   - hloubková rekurze na seznamech
   - hloubková akumulace
3. Další témata k rekurzi
   - rekurze bez použití `define`
   - lokální definice rekurzivních procedur

Opakování: Rekurze a indukce

Rekurze – obecný definiční princip + programovací technika

- vyřešení problému redukcí na *problém stejného typu*, ale *menšího rozsahu*
- **rekurzivní procedura** = procedura aplikující sebe sama
- **výpočetní procesy** generované rekurzivními procedurami:
  - **lineární rekurzivní výpočetní proces**
    - fáze navíjení (budování nedokončeného výpočtu)
    - fáze odvíjení (zpětné dosazování a dokončení výpočtu)
  - **lineární iterativní výpočetní proces**
    - generován koncově rekurzivními procedurami
    - fáze odvíjení = vrácení hodnoty
    - analogie cyklu, aplikace probíhá v jednom prostředí

Indukce – dokazovací princip

- matematická indukce; strukturální indukce (obecnější)

Příklad (lineárně rekurzivní a iterativní verze build-list)

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

(define build-list-iter
  (lambda (n f)
    (let iter ((i (- n 1))
               (accum '()))
      (if (< i 0)
          accum
          (iter (- i 1) (cons (f i) accum))))))
```

Metody zastavení rekurze

Limitní podmínka rekurze

- limitní podmínky musí být dosaženo po konečně mnoha krocích výpočtu
- předěl mezi fázemi navíjení a odvíjení

Jak vyjádřit limitní podmínku?

1. podmíneným výrazem pomocí speciálních forem `if` a `cond`
2. podmínkou vyjádřenou pomocí speciálních forem `and` a `or`
3. využitím triviálního případu aplikace jiné procedury

Poznámka: rekurze bez limitní podmínky

- implicitní definice nekonečného proudu (líné vyhodnocování)
- více kurs *Paradigmata programování 2*

Limitní podmínka vyjádřená pomocí `if` a `cond`

Jako doposud:

- vyjádření limitní podmínky pomocí podmíněného výrazu
- tvar: `(if` ⟨*limitní podmínka*⟩ ⟨*triviální případ*⟩ ⟨*předpis rekurze*⟩`)`
- obecněji pomocí `cond` nebo vnořených `if`

Příklad (predikát `list?` testující, jestli je argument seznam)

```
(define list?
  (lambda (l)
    (if (null? l)
        #t
        (and (pair? l)
             (list? (cdr l))))))
```

Proč musí být `if` speciální forma?

Vytvoření `if` jako pocedury:

```
(define if-proc
  (lambda (condition expr altex)
    (if condition expr altex)))
```

Faktoriál pomocí `if-proc`:

```
(define fak-loop
  (lambda (n)
    (if-proc (= n 1)
             1
             (* n (fak-loop (- n 1))))))
```

`(fak-loop 5)` $\Longmapsto$ ∞

- `if` jako procedura *nezastaví rekurzi*

Limitní podmínka vyjádřená pomocí and a or

Předchozí implementace:

```
(define list?
  (lambda (l)
    (if (null? l)
        #t
        (and (pair? l)
             (list? (cdr l))))))
```

Elegantnější řešení:

```
(define list?
  (lambda (l)
    (or (null? l)
        (and (pair? l)
             (list? (cdr l))))))
```

Pár poznámek o `list?`

Seznamy jako rekurzivní datové struktury:

Definice (Seznam, viz Přednášku 5)

Seznam je každý element L splňující právě jednu z následujících podmínek:

1. L je prázdný seznam (to jest L je element vzniklý vyhodnocením `'()`), nebo
2. L je pár ve tvaru `(`E . L'`)`, kde E je libovolný element a L' je seznam.

Předchozí `list?` testuje přesně body předchozí definice.

- **pro:** `list?` je pravdivý, právě když je element (skutečně) seznam
- **proti:** `list?` je pomalý (složitost $O(n)$ vzhledem k délce seznamu)

Urchlení `list?` za cenu nesouladu s definicí seznamu

```
(define list*? (lambda (l) (or (null? l) (pair? l))))
```

Příklad (predikáty forall a exists; iterativní verze)

Všeobecný kvantifikátor:

```
(define forall
  (lambda (f l)
    (or (null? l)
        (and (f (car l))
             (forall f (cdr l))))))
```

Existenční kvantifikátor:

```
(define exists
  (lambda (f l)
    (and (not (null? l))
         (or (f (car l))
             (exists f (cdr l))))))
```

Příklad (příklad „ukryté limitní podmínky“)

Varianta `map` zachovávající strukturu seznamu

```
(define depth-map
  (lambda (f l)
    (map (lambda (x)
           (if (list? x)
               (depth-map f x)
               (f x)))
         l)))
```

Příklady použití:

```
(depth-map - '(((1 2)) ((3 4))))   ⟾   (((-1 -2)) ((-3 -4)))
(depth-map - '(((1) 2) (3 (4))))   ⟾   (((-1) -2) (-3 (-4)))
(depth-map - '((((1) 2)) 3 (4)))   ⟾   ((((-1) -2)) -3 (-4))
```

Motivace pro stromovou rekurzi: 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 $n = 0, 1, 2, 3, 4, \ldots$

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

Příklad (procedura pro výpočet Fibonacciho čísel)

Definice F_n:

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

Přímočarý přepis ve Scheme:

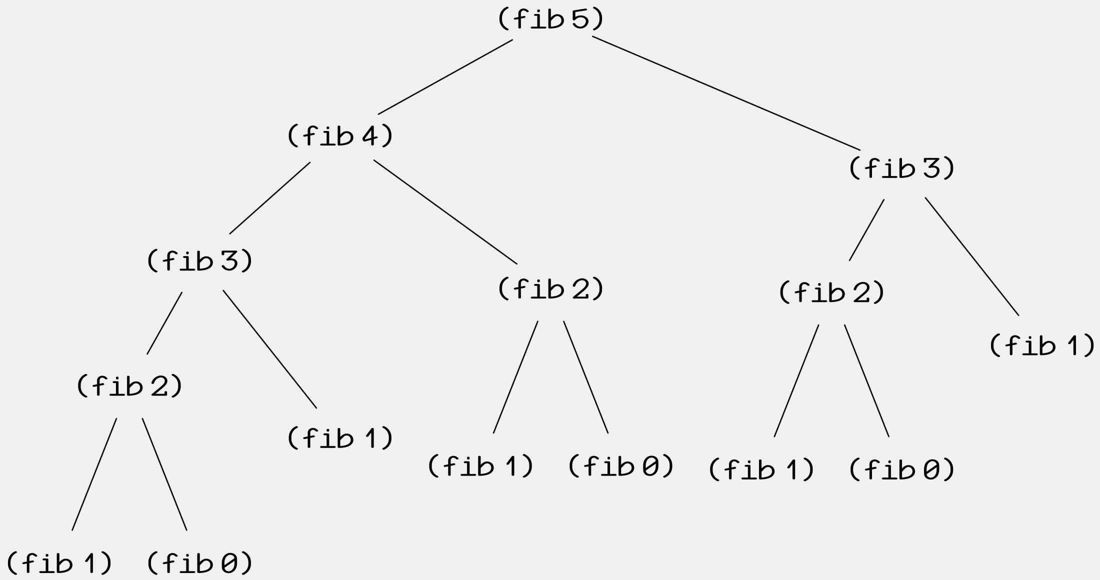
```
(define fib
  (lambda (n)
    (if (<= n 1)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

Otázky: Je efektivní? Jak probíhá výpočet?

Příklad (průběh výpočtu F_5)

```
(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (fib 3))
(+ (+ (+ (fib 2) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ (fib 1) (fib 0)) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ 1 (fib 0)) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ (+ 1 0) (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ 1 (fib 1)) (fib 2)) (fib 3))
(+ (+ (+ 1 1) (fib 2)) (fib 3))
(+ (+ 2 (fib 2)) (fib 3))
(+ (+ 2 (+ (fib 1) (fib 0))) (fib 3))
(+ (+ 2 (+ 1 (fib 0))) (fib 3))
(+ (+ 2 (+ 1 0)) (fib 3))
(+ (+ 2 1) (fib 3))
(+ 3 (fib 3))
(+ 3 (+ (fib 2) (fib 1)))
(+ 3 (+ (+ (fib 1) (fib 0)) (fib 1)))
(+ 3 (+ (+ 1 (fib 0)) (fib 1)))
(+ 3 (+ (+ 1 0) (fib 1)))
(+ 3 (+ 1 (fib 1)))
(+ 3 (+ 1 1))
(+ 3 2)
5
```

Příklad (strom zachycující aplikace fib)



Stromově rekurzivní výpočetní proces

Stromově rekurzivní výpočetní proces

- proces generovaný rekurzivními procedurami, které ve svých rekurzivních předpisech vyvolají *dvě nebo více aplikací sebe sama*
- *střídání fází* navíjení a odvíjení
- průběh výpočtu lze znázornit jako strom

Stanovení složitosti:

- časová složitost – odvíjí se od *počtu uzlů* ve stromu
- prostorová složitost – odvíjí se od *hloubky stromu*

Příklad (složitost předchozí verze `fib`)

- časová $O(2^n)$ (exponenciální; lze zpřesnit pomocí „zlatého řezu“)
- prostorová $O(n)$ (lineární)

Příklad (Fibonacciho čísla iterativně)

Fibonacciho posloupnost:

n:	0	1	2	3	4	5	6	7	8	9	10 ...
$F(n)$:	1	1	2	6	24	120	720	5040	40320	362880	3628800 ...

```scheme
(define fib
  (lambda (n)
    (let iter ((a 0)
               (b 1)
               (i n))
      (if (<= i 0)
          a
          (iter b (+ a b) (- i 1))))))
```

- časová složitost $O(n)$
- prostorová složitost $O(1)$

Problém počtu atomů v seznamu

Seznam × atom

- za atom považujeme každý element, který není seznam,
- e je atom p. k. `(list? e)` $\Longmapsto$ `#f`
- někdy možno obecněji: atom je cokoliv kromě páru

Délka seznamu × počet atomů

- obecně jiné číslo, například pro `((a b c))`

Příklad (počet atomů v seznamu)

```
(atoms '())                           ⟾  0
(atoms '(1))                          ⟾  1
(atoms '((a)))                        ⟾  1
(atoms '(1 ((2))))                    ⟾  2
(atoms '(1 () 2 (a (() (b (c)) (d) e))) ⟾  7
```

Příklad (implementace počtu atomů v seznamu)

Analýza pro seznam l:

- pokud je l prázdný, pak má 0 atomů,
- první prvek l je atom $\Longmapsto$ $1+$ počet atomů ve zbytku l
- první prvek l je seznam $\Longmapsto$ sečteme počty atomů v prvním prvku l i zbytku l

Implementace:

```
(define atoms
  (lambda (l)
    (cond ((null? l) 0)
          ((list? (car l)) (+ (atoms (car l))
                              (atoms (cdr l))))
          (else (+ 1 (atoms (cdr l)))))))
```

Časová složitost: $O(n)$ (kde n je počet párů za předpokladu, že `list?` je rychlý)

Intermezzo: problémy se „zapeklitými seznamy“

Co je vlastně počet párů/atomů? Možné dva pohledy:

- logický: počet atomů, které vidíme v externí reprezentaci seznamu
- fyzický: počet atomů uložených v interní reprezentaci seznamu

Příklad (generátor zapeklitých seznamů)

```
(define iffy-list
  (lambda (n value)
    (let build ((i n))
      (if (<= i 0)
          (cons value '())
          (let ((result (build (- i 1))))
            (cons result result))))))
```

Pozorování: `(atoms (iffy-list` n `#f))` $\Longmapsto$ 2^n (**!!**)

Jak testovat (fyzickou) rovnost párů?

Predikát `eqv?` (primitivní procedura):

- slabší verze `equal?`

  `(eqv?` $\langle elem_1 \rangle$ $\langle elem_2 \rangle$`)` $=$ `#t` implikuje `(equal?` $\langle elem_1 \rangle$ $\langle elem_2 \rangle$`)` $=$ `#t`
- pokud jsou $\langle elem_1 \rangle$ a $\langle elem_2 \rangle$ páry, pak `eqv?` vrací `#t` právě tehdy, když jsou $\langle elem_1 \rangle$ a $\langle elem_2 \rangle$ stejný (fyzický) element

Příklad (příklady použití `eqv?`)

```
(equal? (list 'a 10) (list 'a 10)) ⟾ #t
(eqv? (list 'a 10) (list 'a 10)) ⟾ #f

(let ((x (list 'a 10)))
  (eqv? x x)) ⟾ #t
```

`memv` – test existence prvku v seznamu (porovnává pomocí `eqv?`)

Stanovení fyzického počtu atomů

```
(define atoms
  (lambda (l)
    (car (%atoms l '()))))

(define %atoms
  (lambda (l found)
    (if (memv l found)
        (cons 0 found)
        (%atoms-process l (cons l found)))))

(define %atoms-process
  (lambda (l found)
    (cond ((null? l) (cons 0 found))
          ((pair? (car l)) (%atoms-depth l found))
          (else (%atoms-tail l 1 found)))))
```

Stanovení fyzického počtu atomů (pomocné procedury)

```
(define %atoms-depth
  (lambda (l found)
    (let* ((result (%atoms (car l) found))
           (count (car result))
           (found (cdr result)))
      (%atoms-tail l count found))))

(define %atoms-tail
  (lambda (l count1 found)
    (let* ((result (%atoms (cdr l) found))
           (count2 (car result))
           (found (cdr result)))
      (cons (+ count1 count2) found))))
```

Příklad (Příklady výpočtu fyzického počtu atomů)

```
(atoms (iffy-list 0 #f)) ⟹ 1
(atoms (iffy-list 1 #f)) ⟹ 1
(atoms (iffy-list 2 #f)) ⟹ 1
(atoms (iffy-list 3 #f)) ⟹ 1

(atoms '(#f))                                    ⟹ 1
(atoms '((#f) #f))                               ⟹ 2
(atoms '(((#f) #f) (#f) #f))                     ⟹ 4
(atoms '((((#f) #f) (#f) #f) ((#f) #f) (#f) #f)) ⟹ 8

(let ((x (list 'a (list 'b) 'c)))
  (atoms (list x (list x (list x)) x))) ⟹ 3
```

Problém linearizace seznamu

Lineární seznam

- seznam, jehož prvky jsou výhradně atomy

Linearizace seznamu

- pro daný seznam vytvoř lineární seznam atomů ve stejném pořadí

Příklad (linearizace seznamu)

```
(linearize '())                          ⟾  ()
(linearize '(1))                         ⟾  (1)
(linearize '((1)))                       ⟾  (1)
(linearize '(1 ((2))))                   ⟾  (1 2)
(linearize '(1 ((2)) () (3 (4) 5)))      ⟾  (1 2 3 4 5)
```

Příklad (implementace linearizace seznamu)

Analýza:

- linearizace prázdného seznamu je prázdný seznam,
- první prvek seznamu je atom $\Longmapsto$ připojíme k linearizaci zbytku
- první prvek seznamu je seznam $\Longmapsto$ linearizujeme první prvek i zbytek a spojíme

Implementace:

```
(define linearize
  (lambda (l)
    (cond ((null? l) '())
          ((list? (car l)) (append (linearize (car l))
                                   (linearize (cdr l))))
          (else (cons (car l) (linearize (cdr l)))))))
```

Časová složitost: $O(n^2)$ (kvůli append)

Podobnost `linearize` a `atoms`

```
(define linearize
  (lambda (l)
    (cond ((null? l) '())
          ((list? (car l)) (append (linearize (car l))
                                   (linearize (cdr l))))
          (else (cons (car l) (linearize (cdr l)))))))

(define atoms
  (lambda (l)
    (cond ((null? l) 0)
          ((list? (car l)) (+ (atoms (car l))
                              (atoms (cdr l))))
          (else (+ 1 (atoms (cdr l)))))))
```

Zobecnení pomocí abstraktní hloubkové akumulační procedury, . . .

Hloubková akumulační procedura

Procedura `depth-accum`

- lze chápat jako „hloubkovou verzi" procedury `foldr`

Implementace:

```
(define depth-accum
  (lambda (combine nil modifier l)
    (cond
      ((null? l) nil)
      ((list? (car l))
       (combine (depth-accum combine nil modifier (car l))
                (depth-accum combine nil modifier (cdr l))))
      (else
       (combine (modifier (car l))
                (depth-accum combine nil modifier (cdr l)))))))
```

Příklad (zjednodušení pomocí pojmenovaného let)

```
(define depth-accum
  (lambda (combine nil modifier l)
    (let accum ((l l))
      (cond ((null? l) nil)
            ((list? (car l))
             (combine (accum (car l)) (accum (cdr l))))
            (else
             (combine (modifier (car l)) (accum (cdr l))))))))
```

Příklad (linearizace a počet atomů pomocí depth-accum)

```
(define s '(a (b c (d)) e))
(depth-accum + 0 (lambda (x) 1) s)   ⟹   5
(depth-accum append '() list s)      ⟹   (a b c d e)
```

Příklad (další použití hloubkové akumulace)

```
(define depth-map
  (lambda (f l)
    (depth-accum cons '() f l)))

(define depth-find
  (lambda (x l)
    (depth-accum (lambda (x y) (or x y))
                 #f
                 (lambda (y) (equal? x y))
                 l)))

(define depth-reverse
  (lambda (l)
    (depth-accum (lambda (x y) (append y (list x)))
                 '() (lambda (x) x) l)))
```

Příklad (alternativní zavedení linearize a atoms)

Jiný přístup k hloubkové rekurzi:

- pokud je l seznam, na každý jeho prvek aplikujeme sebe sama
- výsledek je agregován pomocí apply (možná i verze s foldr)
- pokud l není seznam, vracíme modifikovaný prvek

```
(define linearize
  (lambda (l)
    (if (list? l)
        (apply append (map linearize l))
        (list l))))
(define atoms
  (lambda (l)
    (if (list? l)
        (apply + (map atoms l))
        1)))
```

Příklad (alternativní zavedení hloubkové akumulace)

```
(define depth-accum
  (lambda (combine modifier l)
    (if (list? l)
        (apply combine
               (map (lambda (x)
                      (depth-accum combine modifier x))
                    l))
        (modifier l))))

(define depth-accum
  (lambda (combine modifier l)
    (let accum ((l l))
      (if (list? l)
          (apply combine (map accum l))
          (modifier l)))))
```

Příklad (hloubková akumulace přes vnořené páry)

Chování v případě obecných párů:

```
(linearize '((a ((b) . c) d)))    ⟼  (a ((b) . c) d)
(atoms '((a ((b) . c) d)))        ⟼  3
```

Další možné řešení:

```
(define tree-accum
  (lambda (combine nil modifier l)
    (let accum ((l l))
      (cond ((null? l) nil)
            ((pair? l) (combine (accum (car l))
                                (accum (cdr l))))
            (else (modifier l)))))))
```

Nebo verze bez terminátoru: `(tree-accum combine modifier l)`

Rekurzivní procedury bez použití `define`

Co víme:

- procedury – vznikají vyhodnocováním λ-výrazů
- speciální forma `define` nemá nic společného se vznikem procedur

Rekurzivní procedury:

- zatím vždy vytvářeny jako „pojmenované“
- navázání procedury na symbol pomocí `define` bezprostředně po vytvoření
- procedura v těle může aplikovat sebe sama pomocí vazby v nadřazeném prostředí

Otázka: Lze uvažovat rekurzivní procedury bez `define`?

Ano, . . . (**!!**)

Příklad (kudy cesta nevede, . . .)

Následující použití `let` ani `let*`:

```
(let ((fak (lambda (n)
             (if (= n 0)
                 1
                 (* n (fak (- n 1)))))))
  (fak 6))
```

nevede ke zdárnému cíli, protože:

```
((lambda (fak)
   (fak 6))
 (lambda (n)
   (if (= n 0)
       1
       (* n (fak (- n 1))))))
```

Y-kombinátor

```
(lambda (y)
  (y y ⟨argument₁⟩ ⟨argument₂⟩ ··· ⟨argumentₙ⟩))
```

- předchozí proceduru aplikujeme s jednou procedurou jako argumentem
- argument bude aplikován se sebou samým jako prvním argumentem

Příklad (faktorál pomocí Y-kombinátoru)

```
((lambda (y)
   (y y 6))
 (lambda (fak n)
   (if (= n 0)
       1
       (* n (fak fak (- n 1))))))
```

Definice (speciální forma `letrec`)

Speciální forma `letrec` se používá ve stejném tvaru jako speciální forma `let*`:

$$\texttt{(letrec ((}\langle symbol_1\rangle\ \langle element_1\rangle\texttt{)}\cdots\texttt{(}\langle symbol_n\rangle\ \langle element_n\rangle\texttt{))}$$
$$\langle v\acute{y}raz_1\rangle\cdots\langle v\acute{y}raz_m\rangle\texttt{)}$$

Její aplikace je ekvivalentní:

$$\texttt{(let ((}\langle symbol_1\rangle\ undefined\texttt{)}\cdots\texttt{(}\langle symbol_n\rangle\ undefined\texttt{))}$$
$$\texttt{(define }\langle symbol_1\rangle\ \langle element_1\rangle\texttt{)}$$
$$\vdots$$
$$\texttt{(define }\langle symbol_n\rangle\ \langle element_n\rangle\texttt{)}$$
$$\langle v\acute{y}raz_1\rangle\cdots\langle v\acute{y}raz_m\rangle\texttt{)},$$

kde $undefined$ je speciální element jazyka zastupující „*nedefinovanou hodnotu*“.

Příklad (specifika speciální formy `letrec`)

Pro více vazeb se chová podobně jako `let*`:

```
(letrec ((x 10)
         (y (+ x 2)))
  (list x y))  ⟾  (10 12)
```

se přepisuje na následující výraz, který je vyhodnocen:

```
(let ((x undefined)
      (y undefined))
  (define x 10)
  (define y (+ x 2))
  (list x y))  ⟾  (10 12)
```

Příklad (specifika speciální formy `letrec`)

Rozdíl oproti `let*`:

- je možné se odkazovat nejen „dozadu“, ale i „dopředu“
- důsledek: možné odkazovat se na sebe sama
- možnost předání nedefinované hodnoty

```
(letrec ((x x)) x)   ⟹   undefined

(letrec ((x y)
         (y 10))
  (list x y))   ⟹   (undefined 10)
```