Rodina – rekurzivní data

- osoby jsou reprezentovány uzly stromu
- každá osoba (uzel) je struktura obsahující:
  - otec – uzel
  - matka – uzel
  - jméno a případná další „normální“ data
- struktury ve strukturách
- je třeba zastavit rekurzi – někde uzly neexistují (neznámý rodič)

Příklad rodinného stromu

jan

petra

roman

zuzana

karel

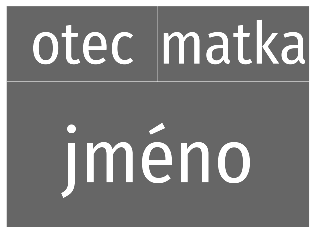
ivo

tereza

alena

ivana

martin



Kód rodinného stromu

```
(define-struct uzel (otec matka jmeno))
(define jan (make-uzel '() '() “Jan“))
(define petra (make-uzel '() '() “Petra“))
(define karel (make-uzel jan petra “Karel“))
(define ivo (make-uzel jan petra “Ivo“))
(define roman (make-uzel '() '() “Roman“))
(define zuzana (make-uzel '() '() “Zuzana“))
(define tereza (make-uzel roman zuzana “Tereza“))
(define alena (make-uzel ivo tereza “Alena“))
(define ivana (make-uzel ivo tereza “Ivana“))
(define martin (make-uzel ivo tereza “Martin“))
```

Hledání předka

- má uzel předka daného jména?
- rekurzivní implementace
  - základní krok: má otec nebo matka takového předka?
  - zastavit pokud je uzel prázdný (neúspěch)
  - zastavit pokud se jméno uzlu shoduje (úspěch)

Hledání předka

```
; predek?:  řetězec  uzel ->  boolean
; má daný uzel předka dotyčného jména?
(define  (predek?  jmeno  uz)
   (cond
      [(empty?  uz)  #f]
      [(string=?  jmeno  (uzel-jmeno  uz))  #t]
      [else  (or   (predek?  jmeno  (uzel-otec  uz))
                   (predek?  jmeno  (uzel-matka  uz)))]
   ))
```

Přehled předků

- vytvořit seznam jmen všech předků daného uzlu

```
; predci:  uzel -> seznam řetězců
; seznam jmen všech předků daného uzlu
(define  (rodina  uz)
   (cond
      [(empty?  uz)  '()]
      [else  (append   (rodina  (uzel-otec  uz))
                       (rodina  (uzel-matka  uz))
                       (list  (uzel-jmeno  uz))]
   ))
```

Zpracování dvou seznamů

- dva seznamy L1, L2 stejné délky
- chceme porovnat dvojice čísel na stejných pozicích a sestavit seznam z menších hodnot
- postup:
  - prvním prvkem výsledku je menší z hodnot (car L1), (car L2)
  - zavolá sama sebe pro zbytek obou seznamů L1, L2
  - podmínka ukončení: prázdný seznam

Funkce min-list

```
; min-list: seznam seznam -> seznam
(define  (min-list  L1  L2)
   (cond
      [(empty?  L1)  '()]
      [else (cons  (min (car  L1) (car L2))
                   (min-list  (cdr  L1)  (cdr  L2)))]
   ))
```

Problémové situace

- předpokládáme shodnou délku obou seznamů
- pokud se liší, máme problém – jestliže bude délka L1 větší než délka L2, program zhavaruje (v podmínce 2. větve zavolá car na prázdný seznam)
- bylo by vhodné ošetřit
- jak porovnat číslo s ničím?
  - výsledkem je nic
  - výsledkem je číslo

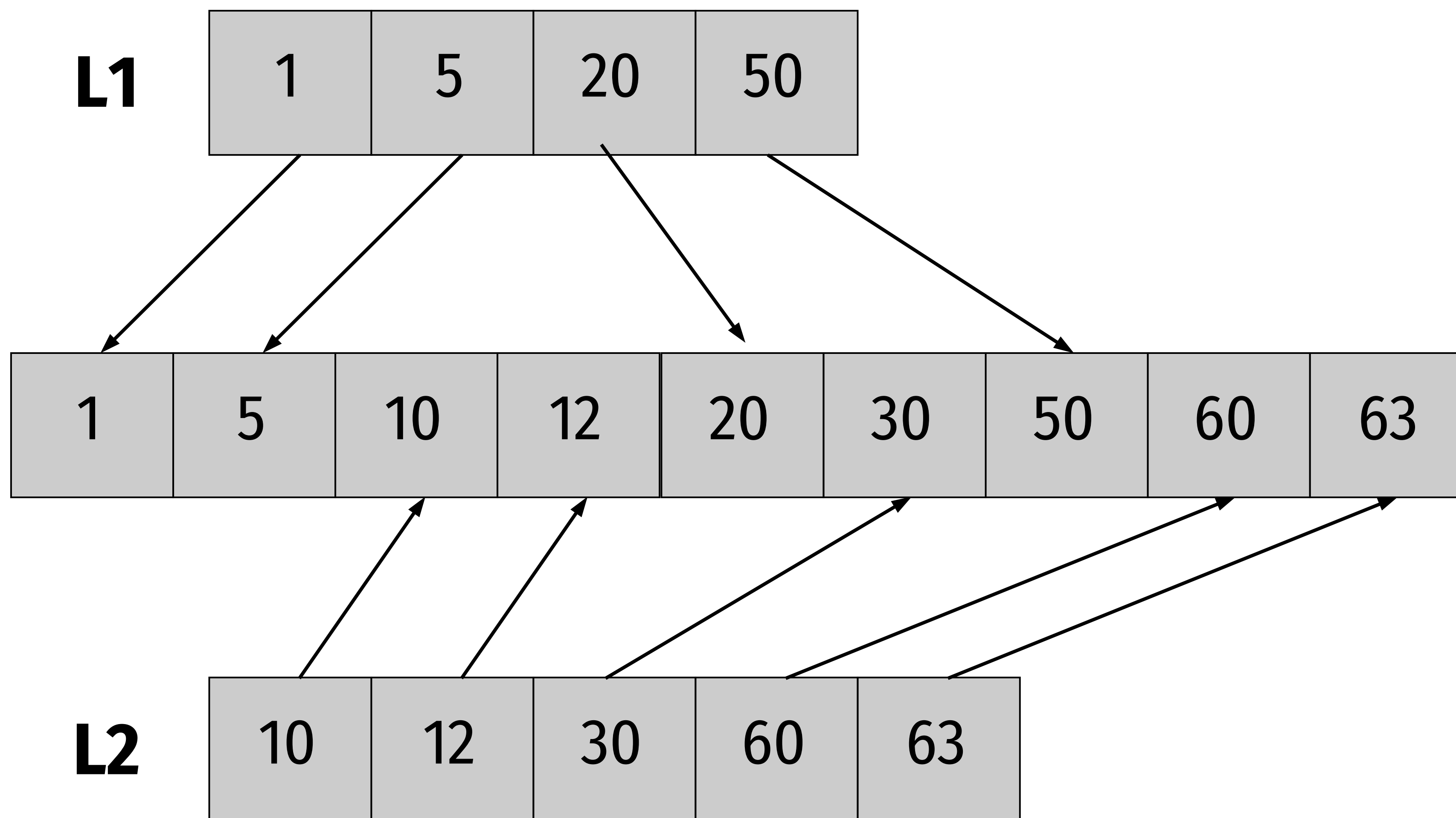
Řešení problémových situací

- **výsledkem je nic**
  - ignorovat zbytek delšího seznamu
  - 1. větev: [(or (empty? L1) (empty? L2)) '()]
- **výsledkem je číslo**
  - zařadit zbytek delšího seznamu do výsledku
  - 1. větev se rozdvojí:
    [(empty? L1) L2]
    [(empty? L2) L1]

Spojení seřazených seznamů

- máme dva seznamy čísel, oba vzestupně seřazené podle velikosti
- chceme čísla z nich spojit do jednoho (opět seřazeného) seznamu
- částečně podobné předchozí úloze, ale
  - v každém kroku se předává do výsledku jen jeden prvek a zkracuje se jen jeden seznam
  - jeden seznam vždy skončí dříve, zbytek druhého se pak předá do výsledku

Spojení seřazených seznamů



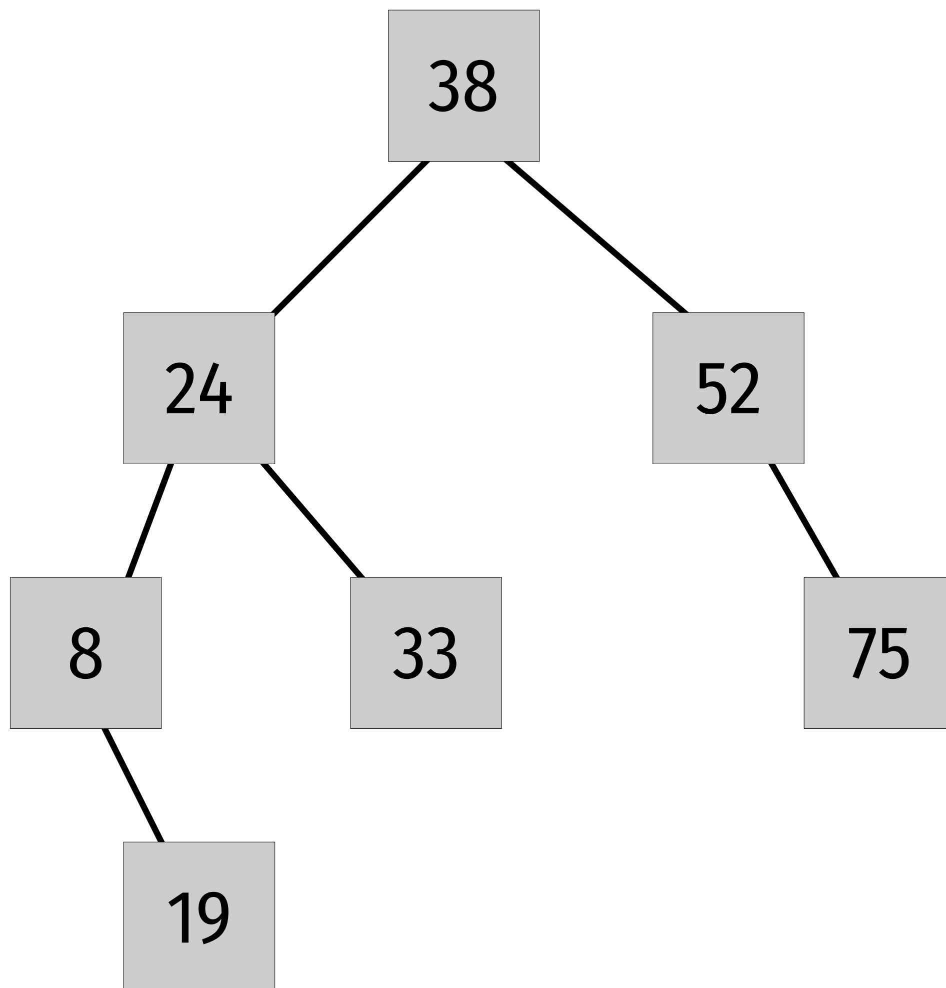
Spojovací funkce

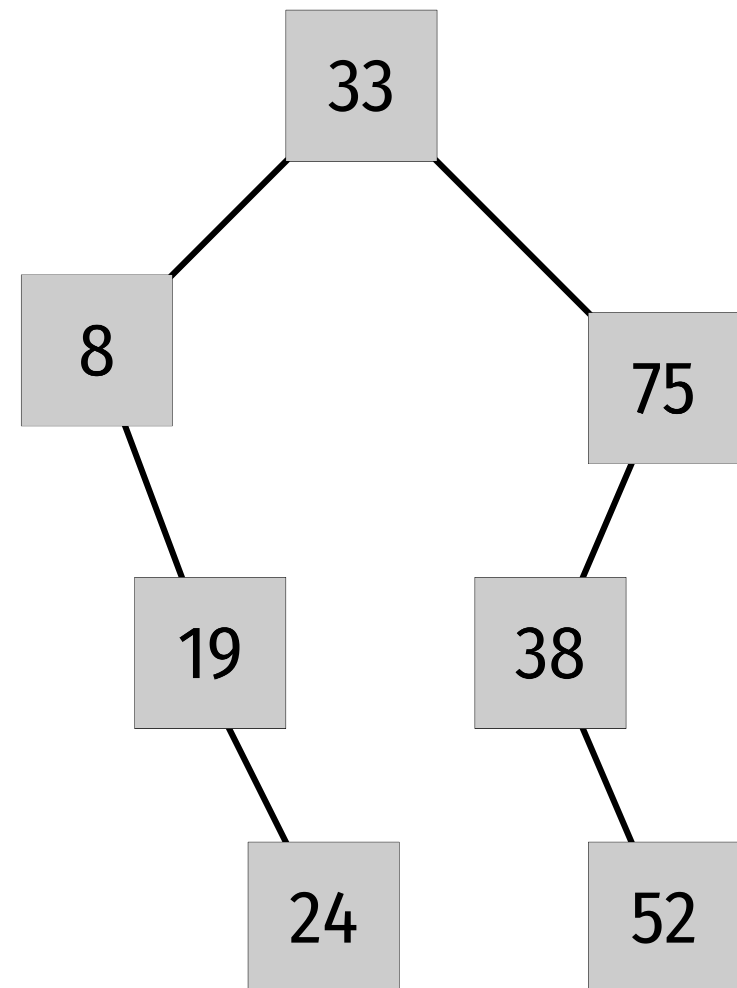
```
; merge: seznam seznam -> seznam
(define (merge L1 L2)
  (cond
    [(empty? L1) L2]
    [(empty? L2) L1]
    [(<= (car L1) (car L2))
      (cons (car L1) (merge (cdr L1) L2))]
    [else
      (cons (car L2) (merge L1 (cdr L2)))]
  ))
```

Binární vyhledávací strom

- každý uzel obsahuje
  - klíčovou hodnotu rozhodující pro uspořádání stromu
  - levý a pravý podstrom → rekurzivní data
  - „užitečné“ hodnoty identifikované klíčem, zde vynecháme
  - (define-struct **uzelBVS** (klic levy pravy))
- všechny klíče uzlů **v levém podstromu jsou menší** než klíč daného uzlu
- všechny klíče uzlů **v pravém podstromu jsou větší** než klíč daného uzlu

Není jednoznačné





Hledání hodnoty

- porovnání hledané hodnoty s klíčem uzlu je rozhodující pro další postup stromem

```
(define (hledej? N strom)
  (cond
    [(empty? strom) #f]
    [(= N (uzelBVS-klic strom)) #t]
    [(< N (uzelBVS-klic strom))
                (hledej? N (uzelBVS-levy strom))]
    [else (hledej? N (uzelBVS-pravy strom))]
  ))
```

Vlastnosti hledání

- hledání je velmi rychlé
- rychlost závisí na hloubce stromu
- ideální strom (co nejširší) obsahující N prvků má hloubku $\mathbf{\log_2 N}$
- např. strom obsahující tisíc uzlů má hloubku 10
- **vyvážený strom:** každý uzel má podstromy skoro stejné velikosti

Vytváření stromu

```
(define (pridej N strom)
    (cond
        [(empty? strom) (make-uzelBVS N '() '())] ;vytvořit nový uzel
        [(< N (uzelBVS-klic strom))       ;přidat do levého podstromu
            (make-uzelBVS    (uzelBVS-klic strom)
                             (pridej N (uzelBVS-levy strom))
                             (uzelBVS-pravy strom))]
        [(> N (uzelBVS-klic strom))       ;přidat do pravého podstromu
            (make-uzelBVS    (uzelBVS-klic strom)
                             (uzelBVS-levy strom)
                             (pridej N (uzelBVS-pravy strom)))]
        [else strom]                      ;hodnota je už ve stromě
    ))
```

Odstranění vrcholu (1)

- opět nejprve nutno vyhledat
- problém s uzly, které mají potomky
  - má-li jen jednoho potomka, zastoupí jej potomek
  - má-li oba potomky, převezme klíč z jednoho z podstromů: buď největší (nejpravější) z levého podstromu, nebo nejmenší z pravého

Odstranění vrcholu (2)

```
(define (odeber N strom)
    (cond
        [(empty? strom) '()]                    ;uzel neexistuje
        [(< N (uzelBVS-klic strom))             ;odebrat z levého podstromu
            (make-uzelBVS   (uzelBVS-klic strom)
                            (odeber N (uzelBVS-levy strom))
                            (uzelBVS-pravy strom))]
        [(> N (uzelBVS-klic strom))             ;odebrat z pravého podstromu
            (make-uzelBVS   (uzelBVS-klic strom)
                            (uzelBVS-levy strom)
                            (odeber N (uzelBVS-pravy strom)))]
        [else (zrus-koren strom)]                   ;vlastní odstranění
    ))
```

Odstranění vrcholu (3)

```
(define (zrus-koren strom)
  (cond
    [(empty? (uzelBVS-levy strom)) (uzelBVS-pravy strom)]
    [(empty? (uzelBVS-pravy strom)) (uzelBVS-levy strom)]
    [else  (make-uzelBVS
             (maxklic (uzelBVS-levy strom))
             (odeber (maxklic (uzelBVS-levy strom))
                     (uzelBVS-levy strom))
             (uzelBVS-pravy strom))]

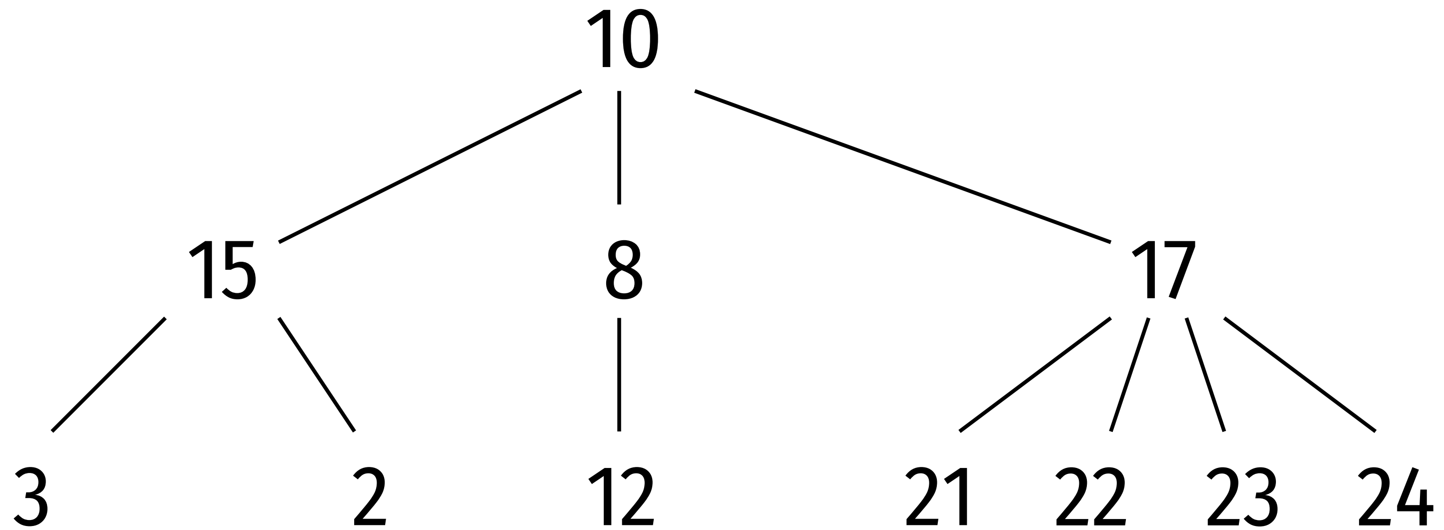
 ))

(define (maxklic strom)
  (if (empty? (uzelBVS-pravy strom)) (uzelBVS-klic strom)
    (maxklic (uzelBVS-pravy strom))))
```

Obecný strom – vnořené seznamy

- v binárním stromě má každý uzel maximálně dva podstromy (potomky)
- v obecném stromě není počet podstromů omezen
- v konvenčních jazycích zpravidla řešeno jako lineární seznam odkazů na podstromy
- implementujme uzel pomocí seznamu
  - první prvek: hodnota (jedna nebo datová struktura obsahující potřebné datové položky)
  - další prvky: podstromy (vnořené seznamy)

Příklad obecného stromu



(10 (15 (3) (2))
(8 (12))
(17 (21) (22) (23) (24)))

Hledání v obecném stromu

- použijeme pomocnou funkci prohledávající podstromy (nepřímá rekurze)

```
(define (hledej? N strom)
  (cond
    [(empty? strom) #f]
    [(= N (car strom)) #t]
    [else (hledejvpodstromech? N (cdr strom))]
  ))
```

Prohledání podstromů

- máme seznam podstromů
- standardní prohledávací funkce (rekurzivně volající předchozí)

```
(define (hledejvpodstromech? N stromy)
  (cond
    [(empty? stromy) #f]
    [(hledej? N (car stromy)) #t]
    [else (hledejvpodstromech? N (cdr stromy))]
  ))
```

Kombinace typů

- fantazii se meze nekladou
  - struktury v seznamech (telefonní seznam)
  - seznamy v seznamech (obecný strom, složky na disku)
  - struktury ve strukturách (binární strom)
  - seznamy ve strukturách (polygon)
  - seznamy ve strukturách v seznamech
    - jednoduchá modifikace obecného stromu
      (define-struct uzelOB (hodnota podstromy))
- závisí na povaze problému

Řešení hrubou silou (1)

- některé úlohy vyžadují probrat všechny možnosti
- příklad: máme množinu bodů v rovině a chceme najít největší vzdálenost mezi libovolnou dvojicí bodů
- reprezentace bodu:
  `(define-struct bod (x y))`
- množina bodů bude seznam těchto struktur

Řešení hrubou silou (2)

- koncept řešení:
  - je-li seznam prázdný, je výsledek 0
  - jinak buď je první bod v seznamu členem nejvzdálenějšího páru, nebo ne
  - najdeme největší vzdálenost mezi prvním bodem a všemi ostatními a největší vzdálenost mezi dvojicemi ve zbytku seznamu
  - větší z těchto hodnot je celkovým maximem

Řešení hrubou silou (3)

- implementace předchozího konceptu ve Scheme
- ```
  (define (maxvzdal body)
    (if (empty? body)
        0
        (max (maxzeseznamu (car body) (cdr body))
             (maxvzdal (cdr body)))))
  ```
  ```

Řešení hrubou silou (4)

- největší vzdálenost mezi bodem a libovolným bodem ze seznamu podobně: prázdný→0, jinak vzít větší ze vzdálenosti od prvního bodu seznamu a maxima ze vzdáleností od ostatních
-
```
(define (maxzeseznamu A body)
  (if (empty? body)
      0
      (max (vzdalenost A (car body))
           (maxzeseznamu A (cdr body)))))
```

Řešení hrubou silou (5)

- zbývá funkce pro vzdálenost dvou bodů v rovině – Pythagorova věta
-
```
(define (vzdalenost A B)
  (sqrt (+ (sqr (- (bod-x A) (bod-x B)))
           (sqr (- (bod-y A) (bod-y B))))))
```

Řešení hrubou silou (6)

- obecná šablona:
  - nejprve otestovat koncovou podmínku – kdy lze úlohu triviálně vyřešit
  - následně najít nejlepší variantu, kdy první prvek je součástí řešení, a nejlepší variantu bez něj (typicky rekurzivní volání sama sebe se zkráceným seznamem)
  - lepší z těchto variant je řešením

Problém batohu (1)

- naplnit batoh dané nosnosti zbožím s největší cenou
- zboží: seznam dvojic (cena hmotnost)
- pomocné funkce:

```
(define (cena vec) (car vec))
(define (vaha vec) (second vec))
(define (cena-batohu seznam)
  (if (empty? seznam) 0
      (+ (cena (car seznam))
         (cena-batohu (cdr seznam)))))
```

Problém batohu (2)

- hlavní funkce

```
(define (batoh nosnost obsah veci)
  (cond
    [(empty? veci) obsah]
    [(> (vaha (car veci)) nosnost)    ;1. zboží se nevejde
          (batoh nosnost obsah (cdr veci))]
    [else (lepsi
      (batoh nosnost obsah (cdr veci))        ;bez 1. zboží
      (batoh (- nosnost (vaha (car veci)))   ;s 1. zbožím
          (cons (car veci) obsah)
          (cdr veci)))]))
```

Problém batohu (3)

- porovnání dvou batohů

```
(define (lepsi batoh1 batoh2)
  (if (> (cena-batohu batoh1) (cena-batohu batoh2))
      batoh1
      batoh2))
```

vytvořeno s podporou
projektu ESF