Strukturovaná data

- struktura ve Scheme obsahuje několik datových položek – podobá se záznamu (record) v Pascalu
- definice:
  (define-struct *jméno* (*seznam jmen položek*))
- vytvoří následující funkce:
  - make-*jméno* – konstruktor pro vytváření těchto struktur
  - *jméno-položka* – selektor pro přístup k položkám
  - *jméno?* – predikát pro testování

Příklad struktury

- (define-struct **osoba** (jmeno telefon))
- definuje:
  - make-osoba
  - osoba-jmeno, osoba-telefon
  - osoba?
- (define klot (make-osoba "klotylda" 123456789))
  vytvoří strukturu typu osoba naplněnou daty
- (osoba-telefon klot)
  vydá 123456789

Struktura versus objekt

- podobnosti
  - položky různých typů
  - přístup k nim prostřednictvím jmen
  - používají se přístupové funkce (selektory)
- odlišnosti
  - objekt má navíc metody
  - selektory jsou pouze pro čtení
  - změna položky je realizována vytvořením nové struktury

Příklad

```
(define-struct clovek (jmeno vek plat))
(define eva (make-clovek "Eva Smutná" 32 26500))
(define (dalsi-rok kdosi)
   (make-clovek
      (clovek-jmeno kdosi)
      (+ 1 (clovek-vek kdosi))
      (clovek-plat kdosi)
   ))
(define starsi-eva (dalsi-rok eva))
(clovek-vek starsi-eva) ..... 33
```

Konzistence dat

- Scheme nemá silnou typovou kontrolu, nekontroluje přípustnost dat automaticky
- (make-clovek 100 'ahoj "opravdu VELKÝ")
  je syntakticky korektní, ale naplní datové položky nesmysly...
- požadované typy lze popsat v komentáři, pokud je uživatel kódu nedodrží, je sám svého štěstí strůjcem
- existují i silnější metody

Zajištění konzistence dat

- udělej si sám – konstruktor lze zabalit do funkce zajišťující kontrolu dat

```
(define (novy-clovek jmeno vek plat)
 (cond
    [(not (string? jmeno)) (error 'novy-clovek "jmeno musí být řetězec")]
    [(not (number? vek)) (error 'novy-clovek "vek musí být číslo")]
    [(not (number? plat)) (error 'novy-clovek "plat musí být číslo")]
    [else (make-clovek jmeno vek plat)]
))
(define eva (novy-clovek "Eva Smutná" 32 26500))
```

Testování typů struktur

```
(define-struct kruh (polomer))

(define-struct obdelnik (stranaA stranaB))

(define (plocha tvar)
    (cond
      [(kruh? tvar)
        (* pi (sqr (kruh-polomer tvar)))]
      [(obdelnik? tvar)
        (* (obdelnik-stranaA tvar)
           (obdelnik-stranaB tvar))]
    ))
```

Struktura versus seznam

	seznam	**struktura**
počet hodnot	libovolný	pevný
přístup k hodnotám	jen první	libovolná
identifikace hodnot	pořadí	selektorové funkce (jménem)
existence	vestavěné	nutno definovat
specifičnost	obecné	specifické pro daný problém

Seznam nebo struktura?

- závisí na konkrétním problému
- pevný počet položek, jednotlivé položky mají specifický význam (např. osobní data – jméno, datum narození,...)

  —› **struktura**
- proměnlivý počet hodnot, podobný význam hodnot (naměřené teploty, účet v obchodě,...)

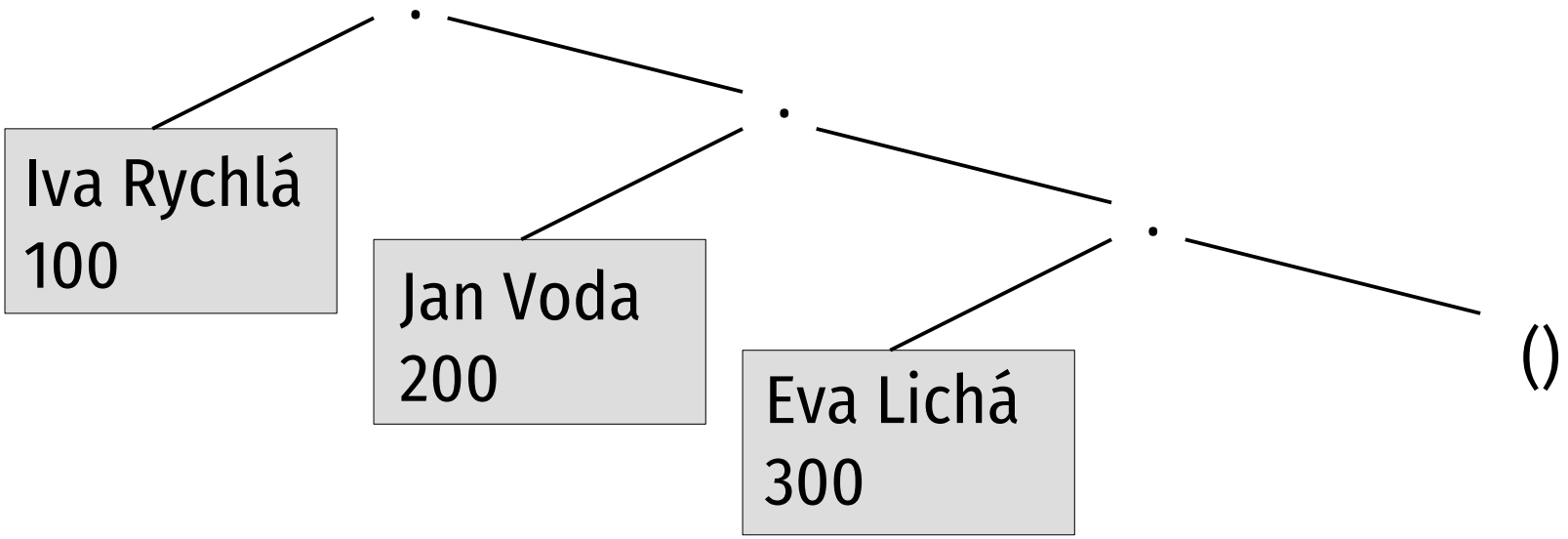
  —› **seznam**

Seznam struktur

- analogie populárního pole záznamů/objektů
- mějme definici struktury
  (define-struct **osoba** (jmeno telefon))
- tyto struktury mohou být zařazeny do seznamu
- (list (make-osoba "Otto Tuhý" 12345))
  vytvoří seznam obsahující jednu osobu
- použitím cons lze přidávat další osoby na začátek seznamu

Telefonní seznam

- (define **telseznam**
  (list (make-osoba "Iva Rychlá" 100)
  (make-osoba "Jan Voda" 200)
  (make-osoba "Eva Lichá" 300)))



Hledání telefonního čísla

- obecné hledání:
  - pokud je seznam prázdný – konec (neúspěch)
  - pokud první prvek vyhovuje kritériím – konec (úspěch)
  - jinak prohledat zbytek seznamu
- jen prostřední případ vyžaduje přístup k datovým položkám prvního prvku:
  - (osoba-jmeno (car seznam))
  - (osoba-telefon (car seznam))

Hledání telefonního čísla

```
; hledej-telefon: řetězec seznam-osob -> číslo
; hledá zadané jméno v telefonním seznamu, pokud
; najde, vrátí odpovídající telefonní číslo, jinak nulu
(define (hledej-telefon jmeno L)
   (cond
      [(empty? L) 0]
      [(string=? jmeno (osoba-jmeno (car L)))
                              (osoba-telefon (car L))]
      [else (hledej-telefon jmeno (cdr L))]
))
```

Přidávání osob

- ; pridej-osobu: řetězec číslo seznam -> seznam
  ; vytvoří novou osobu a přidá ji na začátek seznamu,
  ; vrátí upravený seznam

```
(define (pridej-osobu jmeno cislo L)
   (cons   (make-osoba jmeno cislo)
           L)
))
```

Přidávání do seřazeného seznamu

```
; pridej-osobu: řetězec číslo seznam -> seznam
; vytvoří novou osobu a vloží ji do seřazeného
; seznamu, vrátí upravený seznam
(define (pridej-osobu jmeno cislo L)
   (cond
      [(empty? L) (list (make-osoba jmeno cislo))]
      [(string<? jmeno (osoba-jmeno (car L)))
         (cons (make-osoba jmeno cislo) L)]
      [else (cons (car L)
                  (pridej-osobu jmeno cislo (cdr L)))]
))
```

Odebírání osob

- musíme dotyčnou osobu najít a vytvořit nový seznam bez ní
- ; odeber-osobu: řetězec seznam -> seznam
  ; odebere ze seznamu osobu daného jména

```
(define (odeber-osobu jmeno L)
    (cond
        [(empty? L) '()]
        [(string=? jmeno (osoba-jmeno (car L))) (cdr L)]
        [else (cons (car L)
                    (odeber-osobu jmeno (cdr L)))]
))
```

Zachování dat

- telefonní seznam je typický příklad datové struktury, kterou je třeba uchovávat v paměti a měnit
- funkce pridej-osobu a odeber-osobu se chovají funkcionálně – dostanou seznam v argumentu a upravený vrátí jako výsledek – nutno uložit
- funkce **set!** změní hodnotu konstanty na novou
- (set! telseznam
  (pridej-osobu "Daniel Sudý" 400 telseznam))

Změny datových struktur

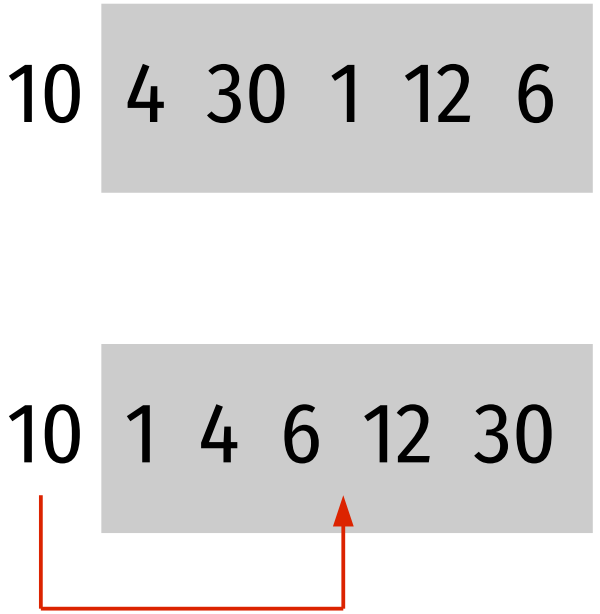
- při nastavení Advanced Student define-struct přidává navíc funkce set-*struktura-položka*!
- umožňují změnit hodnotu položky
- označovány jako **mutátory**
- (define-struct kontakt (jmeno telefon))
  (define eva (make-kontakt "Eva" 12345))
  (**set-kontakt-telefon!** eva 98745)

Řazení čísel

- seznam obsahující čísla má být uspořádán podle velikosti (vzestupně)
- existuje řada metod, některé jsou pro Scheme ideální, jiné se pomocí tohoto jazyka implementují složitě
- základní vlastnosti Scheme:
  - chybí přístup k libovolnému prvku seznamu
  - přirozené pro rekurzivní algoritmy

Třídění vkládáním (1)

- insert sort
- dáme stranou 1. prvek
- ostatní setřídíme
- pak mezi ně zařadíme bývalý první prvek na patřičné místo podle jeho velikosti



Třídění vkládáním (2)

- podmínka ukončení: prázdný seznam (je seřazen, není co řešit)
- ; insertsort: seznam -> seznam
  ; seřadí čísla v seznamu podle velikosti

```
(define  (insertsort  L)
    (cond
        [(empty?  L)  '()]
        [else  (zarad-cislo  (car  L)
                             (insertsort  (cdr  L)))]
))
```

rekurzivní volání

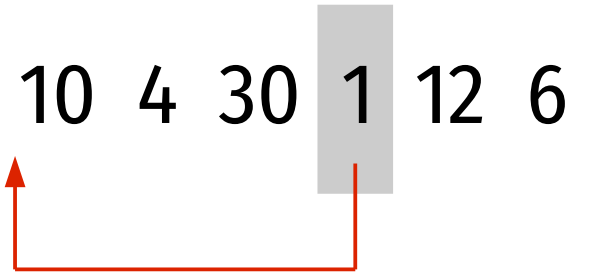
pomocná funkce

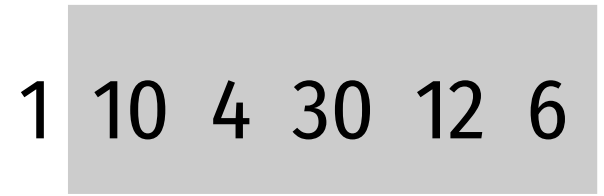
Třídění vkládáním (3)

- vkládání čísel – seznam je seřazen, zbývá najít odpovídající místo
-
```
(define (zarad-cislo N L)
   (cond
      [(empty? L) (list N)]
      [(<= N (car L)) (cons N L)]
      [else (cons (car L)
                  (zarad-cislo N (cdr L)))]
))
```

Třídění výběrem (1)

- select sort
- vybereme nejmenší hodnotu
- přesuneme ji na začátek
- pak setřídíme zbývající hodnoty





Třídění výběrem (2)

```
(define (selectsort L)
   (cond
      [(empty? L) '()]
      [else (cons (nejmensi L)
                  (selectsort
                     (odeber-cislo (nejmensi L) L)))]
   ))
```

nejmenší hodnota na začátek

setřídit zbytek (bez nejmenší hodnoty)

Třídění výběrem (3)

- odstranění čísla
- ; odeber-cislo: cislo seznam -> seznam

```
(define (odeber-cislo N L)
   (cond
      [(empty? L) '()]
      [(= N (car L)) (cdr L)]
      [else (cons  (car L)
                   (odeber-cislo N (cdr L)))]
))
```

Třídění výběrem (4)

- nejmenší z jednoprvkového seznamu je triviální
- jinak vezmeme menší z hodnot: první prvek seznamu a nejmenší ze zbytku – lze využít min

```
(define (nejmensi L)
  (cond
     [(empty? (cdr L)) (car L)]
     [else (min (car L)
                (nejmensi (cdr L)))]
  ))
```

Shrnutí

- řazení vkládáním je pro Scheme **mnohem** přirozenější
- pokud je třeba zpracovat seznam, je vhodné definovat pomocnou rekurzivní funkci
- ve valné většině případů je ukončující podmínkou prázdný seznam
  - výsledkem prázdný seznam, pokud funkce vytváří seznam
  - výsledkem nějaká hodnota, pokud funkce počítá hodnotu

Určení hodnoty

- **zdola nahoru**
  - výsledek rekurzivního volání využijeme ve výrazu stanovícím výsledek
  - přirozená a častější metoda
- **shora dolů**
  - „polotovar“ výstupní hodnoty určíme a předáme v argumentu, na konci rekurze jen vrátíme ve výsledku
  - vyžaduje argument navíc – často vytváříme obalovou funkci, která zajistí počáteční inicializaci

Nejmenší shora dolů (1)

- potřebujeme kandidáta – vezměme první hodnotu
- potom procházíme zbytek seznamu a porovnáváme kandidáta s prvními prvky
- kdykoli narazíme na menší hodnotu, stane se novým kandidátem
- až dorazíme na konec, víme, že aktuální kandidát je skutečně nejmenším číslem v seznamu

Nejmenší shora dolů (2)

```
(define (nejmensi L)
    (hledejmin (car L) (cdr L)))

(define (hledejmin kandidat L)
    (cond
      [(empty? L) kandidat]
      [else (hledejmin (min kandidat (car L))
                       (cdr L))]
  ))
```

Nejmenší shora dolů (3)

- připomíná průchod polem v imperativních jazycích:

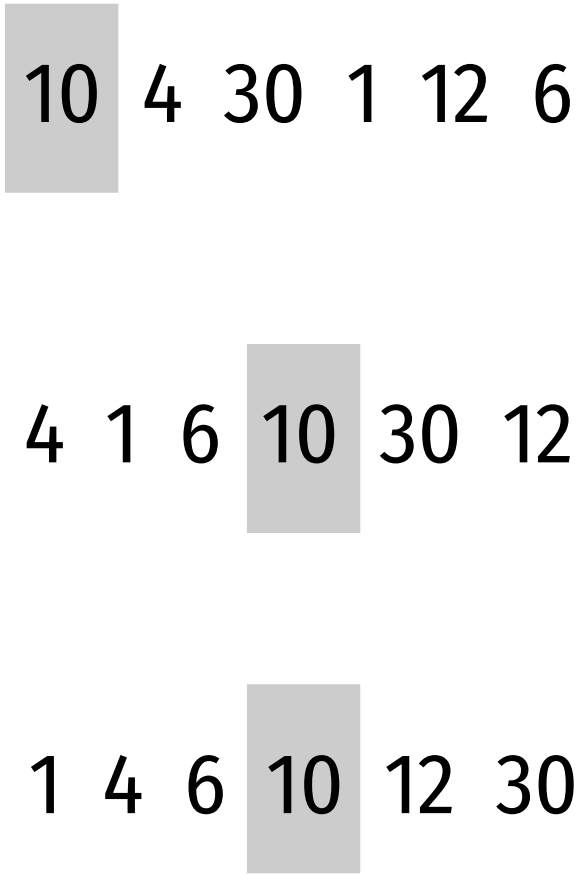
```
const pole = [ 15, 8, 23, 49, 3, 12 ];
let min = pole[0];

for ( let i in pole ) {
    if ( pole[i] < min ) {
        min = pole[i];
    }
}

console.log(min);
```

Koncové volání (tail call)

- **posledním příkazem** funkce **je volání funkce** (nebo sebe sama – koncová rekurze)
- výsledek se jen převezme, **není v žádném výrazu**
- lze implementovat efektivněji – není třeba ukládat na zásobník, volání funkce se prostě přepíše novým
- implementace Scheme musí podporovat
  - experiment: hledání nejmenšího shora dolů v DrRacket cca o 30 % rychlejší než zdola nahoru

Třídění rozdělováním (1)

- quicksort
- vybereme pivota (první prvek)
- rozdělíme na 3 části:
  - menší než pivot
  - pivot
  - větší než pivot
- k setřídění 1. a 3. části voláme rekurzivně



Třídění rozdělováním (2)

```
(define (quicksort L)
   (cond
      [(empty? L) '()]
      [(empty? (cdr L)) L]
      [else (append
               (quicksort (mensi (car L) (cdr L)))
               (list (car L))
               (quicksort (vetsi (car L) (cdr L))))]
))
```

Třídění rozdělováním (3)

```
(define (mensi N L)
   (cond
      [(empty? L) '()]
      [(>= (car L) N) (mensi N (cdr L))]
      [else (cons (car L)
                  (mensi N (cdr L)))]
   ))
(define (vetsi N L)
   ; totéž s podmínkou (< (car L) N)
   )
```

Třídění rozdělováním (4)

- přirozeně rekurzivní algoritmus
- průměrně nejrychlejší – $O(n \cdot \log_2 n)$

  předchozí jsou řádu $O(n^2)$
- problémy:
  - pro (skoro) uspořádaný seznam degeneruje na $O(n^2)$; obvykle se pivot volí jako prostřední hodnota z prvního, prostředního a posledního prvku, ve Scheme těžko
  - seznam prochází v každém kroku dvakrát (mensi a vetsi)

Typy rekurze

- **strukturální rekurze**
  - odvozena z datových struktur
  - typicky ošetření koncových stavů + jednoduchá metoda kombinace dílčích výsledků k dosažení celkového
  - např. insertsort, počítání délky seznamu apod.
- **generativní rekurze**
  - rekurze vychází z podstaty algoritmu
  - vztah určující výsledek bývá složitější
  - např. quicksort

Bonus: Bublinkové třídění (1)

- bubblesort
- projdeme zleva doprava a každou dvojici sousedních prvků ve špatném pořadí prohodíme
- největší prvek se ocitne na konci
- příště stačí projít o 1 méně

10 4 30 1 12 6

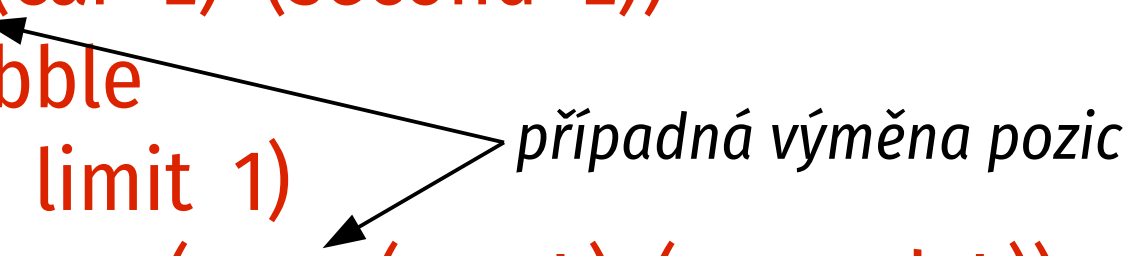
4 10 1 12 6 30

Bublinkové třídění (2)

```
(define (bubblesort L)
   (bbsort (length L) L))

(define (bbsort limit L)
   (if (<= limit 1) L
      (bbsort (- limit 1)
              (gobubble limit L))))
```

Bublinkové třídění (3)

```
(define (gobubble limit L)
   (if (<= limit 1) L
      (cons (min (car L) (second L))
            (gobubble
                  (- limit 1)
                  (cons (max (car L) (second L))
                        (cdr (cdr L))))))))
```



vytvořeno s podporou
projektu ESF