Paradigmata programování 1

Kombinatorika na seznamech, reprezentace stromů a množin

Vilém Vychodil

Katedra informatiky, PřF, UP Olomouc

Přednáška 10

Přednáška 10: Přehled

1. Reprezentace stromů pomocí seznamů
   - stromy jako seznamy seznamů
   - konstruktory a selektory pro stromy
   - průchod stromem do hloubky a do šířky
   - iterativní verze algoritmů využívající zásobník a frontu
2. Reprezentace množin
   - množiny reprezentované uspořádanými seznamy
   - intermezzo: slévání a třídění seznamů (algoritmus *merge sort*)
   - nalezení prvku, operace průniku a sjednocení
   - množiny reprezentované binárními vyhledávacími stromy
3. Kombinatorika na seznamech
   - výpočet prvků potenční množiny
   - permutace, variace, kombinace, ...

Opakování: `linearize`, `atoms` a jejich zobecnění

```
(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)))))))
```

Časová složitost: $O(n)$, kde n je (logický) počet párů (`atoms`)

Příklad (iterativní verze atoms)

Analýza:

- pokud je první prvek atom, inkrementujeme střadač a zkrátíme seznam
- pokud je první prvek seznam, modifikujeme seznam:
  místo prvního prvku připojíme na začátek seznamu prvky prvního prvku

Implementace:

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

Příklad (průběh výpočtu iterativního atoms)

```
(atoms '(a (((b) (c ((d))))) (e)))
(a (((b) (c ((d))))) (e))  0
((((b) (c ((d))))) (e))    1
(((b) (c ((d)))) (e))      1
((b) (c ((d))) (e))        1
(b (c ((d))) (e))          1
((c ((d))) (e))            2
(c ((d)) (e))              2
(((d)) (e))                3
((d) (e))                  3
(d (e))                    3
((e))                      4
(e)                        4
()                         5
```

5 ... analogicky lze udělat `linearize` a spol.

Reprezentace stromů pomocí seznamů

Co je strom? – mnoho (ekvivalentních) definic

- acyklický souvislý neorientovaný graf
- **uzly stromu** (nesou **hodnoty**) – někdy též **vrcholy**
  - různé typy uzlů: **kořen** (je jeden), **vnitřní uzly**, **vnější uzly** (**listy**)
- **hrany** – spojnice mezi uzly

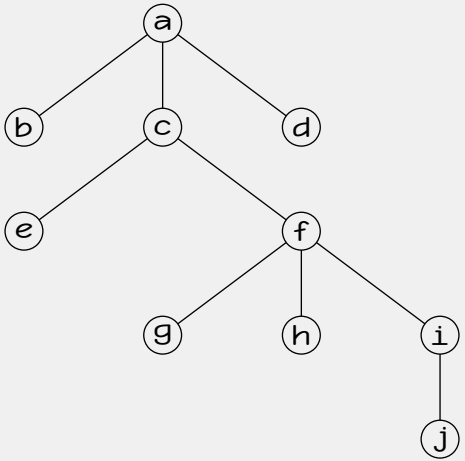
My budeme chápat stromy jako rekurzivní datové struktury:

Definice (strom, hodnota stromu, podstrom, list)

Každý tečkový pár T, kde `(car T)` je libovolný element a `(cdr T)` je *seznam stromů*, se nazývá **strom**. Je-li T strom, pak `(car T)` se nazývá **hodnota stromu** T a prvky seznamu `(cdr T)` se nazývají **podstromy** (**větve**) **stromu** T. Stromy, které nemají podstromy, se nazývají **listy**.

Příklad (strom a jeho grafická reprezentace)

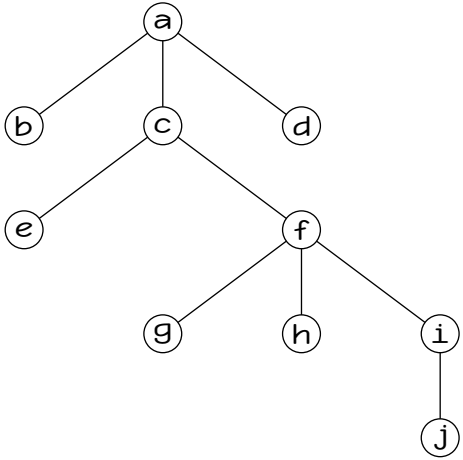
```
(a (b)
   (c (e)
      (f (g)
         (h)
         (i (j))))
   (d))
```

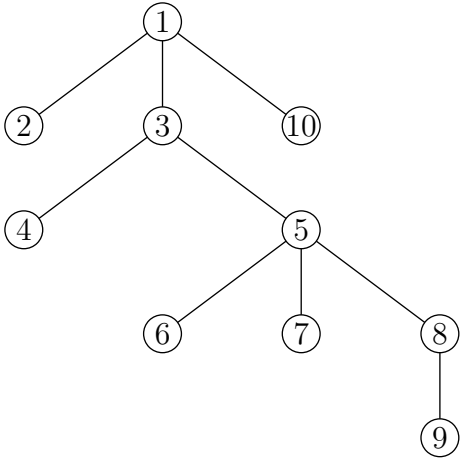


Vytvoření abstrakční bariéry pro stromy

```
(define make-tree
  (lambda (val . subtrees)
    (cons val subtrees)))

(define get-val car)

(define get-branches
  (lambda (x)
    (cdr x)))

(define tree?
  (lambda (x)
    (and (pair? x)
         (forall tree? (cdr x)))))
```

Prohledávání stromu do hloubky (DFS)





- **DFS** = **depth-first search**
- výsledek průchodu: `(a b c e f g h i j d)`

Příklad (DFS, rekurzivní verze)

Analýza:

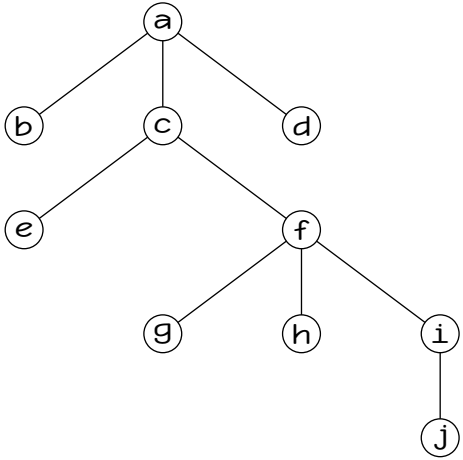
- nejprve zpracuj kořen a pak hloubkově prohledej podstromy zleva-doprava
- principiálně podobné *linearizaci seznamu*

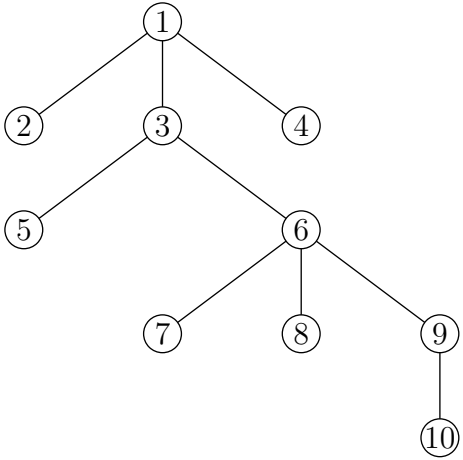
Implementace:

```
(define dfs
  (lambda (tree)
    (cons (get-val tree)
          (apply append
                 (map dfs (get-branches tree))))))
```

- časová složitost: $O(n)$, kde n je celkový počet podstromů (hodnot)
- prostorová složitost: $O(n)$ (strom degenerovaný na seznam)

Prohledávání stromu do šířky (BFS)





- **BFS** = **breadth-first search**
- výsledek průchodu: `(a b c d e f g h i j)`

Příklad (BFS, rekurzivní verze)

Analýza:

- je udržován seznam podstromů na jedné úrovni
- v každé fázi je seznam nahrazen seznamem podstromů na hlubší úrovni

Implementace:

```
(define bfs
  (lambda (tree)
    (let bfs ((trees (list tree)))
      (if (null? trees)
          '()
          (let ((vals (map get-val trees))
                (branches (map get-branches trees)))
          (append vals (bfs (apply append branches))))))))
```

- prostorová a časová složitost jako u DFS

Iterativní verze DFS a BFS

Iterativní DFS / BFS je založená na udržování:

- seznamu dosud nezpracovaných (pod)stromů
- střadač hodnot, které již byly vyhledány (navštíveny)

Jak aktualizovat střadač?

- každá navštívená hodnota je přidána do střadače (seznamu)
- přidáváme v opačném pořadí (technický detail)

Jak aktualizovat seznam nezpracovaných (pod)stromů?

- při DFS tak, aby byly nejprve zpracovány všechny podstromy nejlevějšího podstromu, pak druhého nejlevějšího, ...
- při BFS tak, aby byly zpracovány všechny podstromy nejlevějšího podstromu až potom, co jsou zpracovány vrcholy „na vyšších patrech“, ...

Příklad (průběh iterativního DFS)

```
[(a (b) (c (e) (f (g) (h) (i (j)))) (d))] ()
[(b) (c (e) (f (g) (h) (i (j)))) (d)]     (a)
[(c (e) (f (g) (h) (i (j)))) (d)]         (b a)
[(e) (f (g) (h) (i (j))) (d)]             (c b a)
[(f (g) (h) (i (j))) (d)]                 (e c b a)
[(g) (h) (i (j)) (d)]                     (f e c b a)
[(h) (i (j)) (d)]                         (g f e c b a)
[(i (j)) (d)]                             (h g f e c b a)
[(j) (d)]                                 (i h g f e c b a)
[(d)]                                     (j i h g f e c b a)
[]                                        (d j i h g f e c b a)
(a b c e f g h i j d)
```

- průběžné seznamy odebíráme ze předu a přidáváme ze předu (**zásobník**)
- seznam akumulovaných hodnot na konci převrátíme

Příklad (průběh iterativního BFS)

```
[(a (b) (c (e) (f (g) (h) (i (j)))) (d))] ()
[(b) (c (e) (f (g) (h) (i (j)))) (d)]     (a)
[(c (e) (f (g) (h) (i (j)))) (d)]         (b a)
[(d) (e) (f (g) (h) (i (j)))]             (c b a)
[(e) (f (g) (h) (i (j)))]                 (d c b a)
[(f (g) (h) (i (j)))]                     (e d c b a)
[(g) (h) (i (j))]                         (f e d c b a)
[(h) (i (j))]                             (g f e d c b a)
[(i (j))]                                 (h g f e d c b a)
[(j)]                                     (i h g f e d c b a)
[]                                        (j i h g f e d c b a)
(a b c d e f g h i j)
```

- průběžné seznamy odebíráme ze předu a přidáváme na konec (**fronta**)
- seznam akumulovaných hodnot na konci převrátíme

Příklad (obecné řešení prohledávání stromu)

```
(define tree-search
  (lambda (tree f)
    (let iter ((trees (list tree))
               (accum '()))
      (if (null? trees)
          (reverse accum)
          (let ((first (car trees))
                (rest (cdr trees)))
            (iter (f append (get-branches first) rest)
                  (cons (get-val first) accum)))))))

(define apply-fxy (lambda (f x y) (f x y)))
(define apply-fyx (lambda (f x y) (f y x)))
(define dfs (lambda (tree) (tree-search s apply-fxy)))
(define bfs (lambda (tree) (tree-search s apply-fyx)))
```

Kde najít stromy?

Extensible Markup Language (XML), podle Wikipedie:

- is a set of rules for encoding documents in machine-readable form,
- emphasize simplicity, generality, and usability over the Internet, ...

Pozor na XML maniaky (**!!**)

```
<cd>
  <title>Stop</title>
  <artist>Sam Brown</artist>
  <country>UK</country>
  <company>A and M</company>
  <price currenty="USD">8.90</price>
  <year>1988</year>
</cd>
```

```
(cd
  (title "Stop")
  (artist "Sam Brown")
  (country "UK")
  (company "A and M")
  ((price currency "USD") 8.90)
  (year 1988))
```

Reprezentace množin

Už jsme ukázali:

- reprezentace množin pomocí seznamů bez duplicit (Přednáška 6)
- neefektivní, většina množinových operací (průnik, sjednocení) složitost $O(n^2)$

Nová reprezentace:

- prvky množin $A = \{a_1, a_2, \ldots, a_n\}$ jsou z univerza U
- na U je definováno lineární uspořádání $\leq$
- reprezentace množiny: **uspořádaný** (**setříděný**) **seznam hodnot**

Výhody a nevýhody nové reprezentace:

- výhoda: sjednocení a průnik lze implementovat se složitostí v $O(n)$
- nevýhoda: technická, potřeba kromě množin předávat i uspořádání $\leq$
- naše příklady: pro jednoduchost uvažujeme seznamy čísel, nepředáváme $\leq$

Příklad (test přítomnosti prvku v množině)

Test založený na porovnání hodnot s prvním prvkem:

```
(define in?
   (lambda (x set)
      (cond ((or (null? set) (< x (car set))) #f)
            ((= x (car set)) #t)
            (else (in? x (cdr set))))))
```

Příklady použití:

```
(in? 3 '())          ⟹  #f
(in? 3 '(1 2))       ⟹  #f
(in? 3 '(1 2 3 4))   ⟹  #t
(in? 3 '(1 2 4))     ⟹  #f
```

Příklad (sjednocení a průnik sléváním)

seznam A	*seznam* B	*prvky zařazené do* $A \cup B$
(2 4 6 7)	(1 2 3 4 5)	1
(2 4 6 7)	(2 3 4 5)	2
(4 6 7)	(3 4 5)	3
(4 6 7)	(4 5)	4
(6 7)	(5)	5
(6 7)	()	6 7, *výsledek* $A \cup B$: (1 2 3 4 5 6 7)

seznam A	*seznam* B	*prvky zařazené do* $A \cap B$
(2 4 6 7)	(1 2 3 4 5)	
(2 4 6 7)	(2 3 4 5)	2
(4 6 7)	(3 4 5)	
(4 6 7)	(4 5)	4
(6 7)	(5)	
(6 7)	()	*výsledek* $A \cap B$: (2 4)

Příklad (implementace sjednocení množin sléváním)

```
(define union
  (lambda (set-A set-B)
    (cond ((null? set-A) set-B)
          ((null? set-B) set-A)
          ((= (car set-A) (car set-B))
           (cons (car set-A)
                 (union (cdr set-A) (cdr set-B))))
          ((<= (car set-A) (car set-B))
           (cons (car set-A)
                 (union (cdr set-A) set-B)))
          (else (cons (car set-B)
                      (union set-A (cdr set-B)))))))
```

Příklad (implementace průniku množin sléváním)

```
(define intersection
  (lambda (set-A set-B)
    (cond ((or (null? set-B) (null? set-A)) '())
          ((= (car set-A) (car set-B))
           (cons (car set-A)
                 (intersection (cdr set-A) (cdr set-B))))
          ((<= (car set-A) (car set-B))
           (intersection (cdr set-A) set-B))
          (else (intersection set-A (cdr set-B))))))
```

Intermezzo: Třídění seznamů sléváním

Motivace: Jak setřídit seznam (čísel)?

- Jak na základě daného seznamu zkonstruovat nový, obsahující stejné prvky (například čísla) ale tak, aby byly uspořádané vzhledem k danému lineárnímu uspořádání $\leq$.

Jak třídit seznamy?

- jiný problém než třídění pole (vnitřní, asociativní třídění)
  - k prvkům pole se přistupuje sekvenčně
  - operace přístupu k prvku je výrazně dražší než porovnání hodnot
  - nepoužívají se klasické algoritmy pro třídění polí (quicksort)

Metody pro (asociativní) třídění spojových seznamů

- vnější třídění (k datovým položkám nelze přistupovat v konstantním čase)
- typický zástupce: *merge sort*

Příklad (slévání dvou uspořádaných seznamů)

```
(define merge
  (lambda (s1 s2 . pred?)
    (let ((<= (if (null? pred?) <= (car pred?))))
      (let merge ((s1 s1)
                  (s2 s2))
        (cond ((null? s1) s2)
              ((null? s2) s1)
              ((<= (car s1) (car s2))
               (cons (car s1)
                     (merge (cdr s1) s2)))
              (else (cons (car s2)
                          (merge s1 (cdr s2)))))))))
```

Příklad (efektivní rozdělení dvou seznamů na dva)

```
(define split
  (lambda (l)
    (let iter ((l l)
               (1st '())
               (2nd '()))
      (if (null? l)
          (cons 1st 2nd)
          (iter (cdr l) 2nd (cons (car l) 1st))))))
```

```
(split '(a b))         ⟼  ((a) b)
(split '(a b c))       ⟼  ((b) c a)
(split '(a b c d))     ⟼  ((c a) d b)
(split '(a b c d e))   ⟼  ((d b) e c a)
```

Příklad (procedura **mergesort**, John von Neumenn, 1945)

```
(define mergesort
  (lambda (l . pred?)
    (let ((<= (if (null? pred?) <= (car pred?))))
      (if (or (null? l) (null? (cdr l)))
          l
          (let ((l (split l)))
            (merge (mergesort (car l) <=)
                   (mergesort (cdr l) <=)
                   <=))))))
```

```
(mergesort '(2 7 4 3 4))      ⟹  (2 3 4 4 7)
(mergesort '(2 7 4 3 4) >=)   ⟹  (7 4 4 3 2)
```

- časová složitost: $O(n \log_2 n)$, kde n je délka seznamu

Reprezentace množin pomocí binárních vyhledávacích stromů

Binární vyhledávací strom (BST) – angl. *binary search tree*

- každý uzel má (nejvýše) dva podstromy (levý a pravý)
- vzhledem k danému uspořádání $\leq$, platí:
  - všechny hodnoty v levém podstr. (pokud existuje) jsou $\leq$ než hodnota v kořenu
  - všechny hodnoty v pravém podstr. (pokud existuje) jsou $\geq$ než hodnota v kořenu
- levý i pravý podstrom (pokud existují) jsou opět binární vyhledávací stromy

Naše reprezentace: opět pomocí seznamů

- analogicky jako u obecných stromů
- fakt, že podstrom není přítomen indikujeme prázdným seznamem

Proč je zajímavé?

- pokud je BST strom vyvážený, vyhledávání v něm je efektivní
- pokud je z něj „liána“, máme „smůlu“

Vytvoření abstrakční bariéry pro BST

Základní konstruktory:

```
(define make-tree
  (lambda (value left right)
    (list value left right)))

(define make-leaf
  (lambda (value)
    (make-tree value '() '())))
```

Selektory pro hodnoty a podstromy:

```
(define tree-value car)
(define tree-left cadr)
(define tree-right caddr)
```

Vyhledávání prvků v BST

Test přítomnosti prvku v BST:

```
(define in?
  (lambda (x tree)
    (cond ((null? tree) #f)
          ((= x (tree-value tree)) #t)
          ((< x (tree-value tree)) (in? x (tree-left tree)))
          (else (in? x (tree-right tree))))))
```

Poznámky:

- pokud je BST vyvážený, vyhledání zabere nejvýše $\log_2 n$ kroků
- časová složitost v nejhorším případě: $O(n)$
- vyvažování stromů, vyvážené stromy, . . . více kurs *Algoritmická matematika II*

Hlavní konstruktor pro BST

Konstruuj BST přidáním elementu do existujícího BST:

```
(define cons-tree
  (lambda (x tree)
    (cond ((null? tree) (make-leaf x))
          ((= x (tree-value tree)) tree)
          ((< x (tree-value tree))
           (make-tree (tree-value tree)
                      (cons-tree x (tree-left tree))
                      (tree-right tree)))
          (else
           (make-tree (tree-value tree)
                      (tree-left tree)
                      (cons-tree x (tree-right tree)))))))
```

A můžeme použít `foldr`, . . . (**!!**)

Kombinatorika na seznamech

Cílem je: pro daný seznam L (bez duplicit) vygenerovat:

- seznam všech podseznamů (podmnožin) L,
- seznam všech permutací seznamu L,
- seznam všech k-prvkových podseznamů (podmnožin) L,
- seznam všech k-tic skládajících se z prvků z L

$\vdots$

Příklad (potenční množina, permutace, . . .)

```
(power-set '(a b c))
  ⟹  ((a b c) (a b) (a c) (a) (b c) (b) (c) ())

(permutations '(a b c))
  ⟹  ((a b c) (a c b) (b c a) (b a c) (c a b) (c b a))
```

Příklad (potenční množina)

Analýza:

$$2^A = \begin{cases} \{\emptyset\} & \text{pokud } A = \emptyset, \\ 2^{\{a_2,\dots\}} \cup \bigcup \{\{a_1\} \cup B | B \in 2^{\{a_2,\dots\}}\} & \text{pokud } A = \{a_1, a_2, \dots\} \neq \emptyset. \end{cases}$$

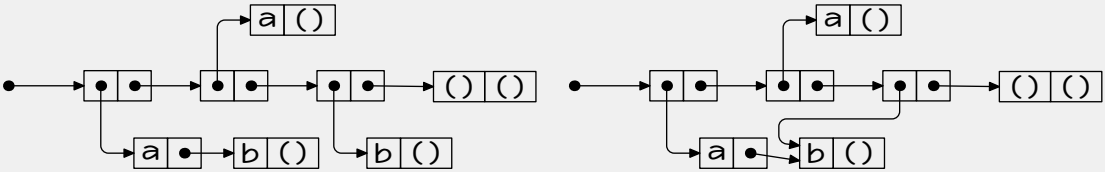
- problém nalezení 2^A se redukuje na problém nalezení 2^B, kde B vznikne z A odebráním jednoho prvku

```
(define power-set
  (lambda (set)
    (if (null? set)
        '(())
        (append (map (lambda (x)
                       (cons (car set) x))
                     (power-set (cdr set)))
                (power-set (cdr set))))))
```

Příklad (potenční množina, stejný princip, lepší implementace)

```
(define power-set
  (lambda (set)
    (if (null? set)
        '(())
        (let ((power-rest (power-set (cdr set))))
          (append (map (lambda (x)
                         (cons (car set) x))
                       power-rest)
                  power-rest)))))
```

Výsledek `(power-set '(a b))`:



Příklad (kombinace)

```
(define combination
  (lambda (k set)
    (cond ((= k 0) '(()))
          ((null? set) '())
          ((= k 1) (map list set))
          (else (%combination k set)))))

(define %combination
  (lambda (k set)
    (append (map (lambda (x)
                   (cons (car set) x))
                 (combination (- k 1) (cdr set)))
            (combination k (cdr set)))))
```

Příklad (permutace)

```
(define permutation
  (lambda (l)
    (if (null? l) '(()) (%permutation l (cdr l)))))

(define %permutation
  (lambda (l rest)
    (if (null? l)
        '()
        (append (map (lambda (x)
                       (cons (car l) x))
                     (permutation rest))
                (let ((rest (cdr (%append l rest))))
                  (%permutation (cdr l) rest))))))

(define %append (lambda (l r) (append r (list (car l)))))
```