KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 10: Kombinatorika na seznamech, reprezentace stromů a množin

Učební materiál k přednášce 7. prosince 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2006

Lekce 10: Kombinatorika na seznamech, reprezentace stromů a množin

Obsah lekce: Tato lekce obsahuje pokročilejší příklady sloužící k procvičení práce s hierarchickými daty. V první části lekce se budeme zabývat reprezentací stromů pomocí seznamů a jejich prohledáváním do hloubky a do šířky. Dále ukážeme reprezentaci množin pomocí uspořádaných seznamů a pomocí binárních vyhledávacích stromů. Závěr lekce je věnován kombinatorice na seznamech.

Klíčová slova: kombinatorika na seznamech, reprezentace množin, stromy.

10.1 Reprezentace n-árních stromů

První sada příkladů se týká n-árních stromů (dále v této sekci budeme používat pojmenování *strom*). Nejdříve popíšeme reprezentaci těchto struktur, poté nadefinujeme příslušný konstruktor a selektory. V závěru sekce pak budeme realizovat algoritmy pro průchod stromem do hloubky a do šířky.

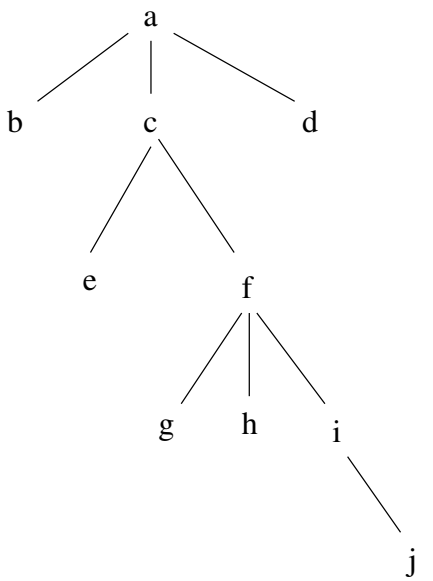
Za strom budeme považovat jakýkoli seznam, který, pokud je neprázdný, má za prvky ocasu další stromy. Hlavu seznamu přitom nazýváme *hodnota* a prvky ocasu nazýváme *podstromy* nebo též *větve stromu*. *Prvky stromu* rozumíme hodnotu stromu a prvky jeho větví. Stromy, které nemají podstromy nebo jsou všechny jejich podstromy prázdné, nazýváme *listy*.

Například strom nakreslený na obrázku 10.1 je reprezentován seznamem ve tvaru

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

Větve tohoto stromu jsou reprezentovány seznamy `(b)`, `(c (e) (f (g) (h) (i (j))))` a `(d)`. Hodnota stromu je `a`. Prvky stromu jsou `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i` a `j`.

Obrázek 10.1. Příklad n-árního stromu



Následuje definice konstruktoru a selektorů pro uvažovanou reprezentaci stromů:

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

(define get-val car)

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

```
    (if (null? x) '() (cdr x))))
```

Dále uvádíme predikát `tree?` zjišťující zda je jeho argument reprezentace stromu. Predikát tedy zjistí, jestli se jedná o seznam a v případě že ano, zjistí jestli jsou jeho prvky (vyjma prvního) též reprezentace stromu. Implementace je velmi přímočará, využíváme v ní predikátu `forall`, který jsme zavedli v lekci 6.

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

Další procedury, které naimplementujeme v této sekci, jsou selektory vracející konkrétní větve. Přesněji selektor `left-branch` vracející k-tou větev zleva a selektor `right-branch` vracející k-tou větev zprava. Přitom budeme chtít, aby pro k, které bude záporné, nebo větší nebo rovno počtu větví vracel selektor prázdný strom (prázdný seznam). Následuje definice selektorů:

```
(define left-branch
  (lambda (tree k)
    (if (or (< k 0) (null? tree))
        '()
        (let iter ((branches (cdr tree))
                   (k k))
          (cond ((null? branches) '())
                ((= k 0) (car branches))
                (else (iter (cdr branches) (- k 1))))))))
```

Procedura `left-branch` nejprve ověří, zda nebylo zadáno záporné číslo a zda zadaný strom není prázdný. Pokud ano, vrací prázdný strom. V opačném případě pomocí iterativní procedury `iter`, postupně „odjímá“ podstromy, dokud nedojde na konec seznamu podstromů – pak vrací prázdný strom – nebo dokud nenajde požadovaný podstrom. Proceduru `right-branch` pak lze vytvořit jednoduše pomocí procedury `left-branch`. Viz její následující definici:

```
(define right-branch
  (lambda (tree k)
    (left-branch tree (- (length tree) k 2))))
```

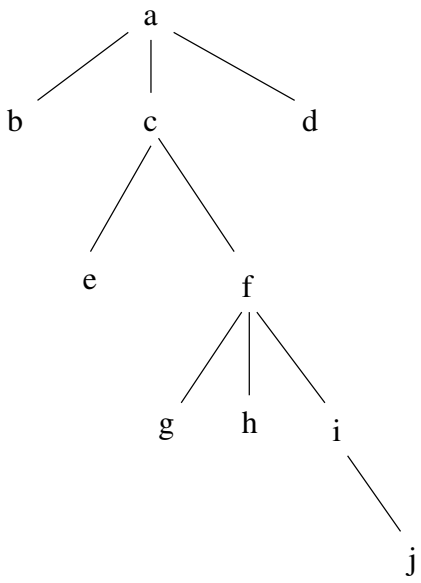
Nyní budeme realizovat algoritmy průchodu stromu do hloubky a průchodu stromu do hloubky. Průchodem stromu přitom myslíme proceduru, která postupně „zpracuje“ všechny prvky stromu. Pod pojmem „zpracování prvků“ budeme mít na mysli konstrukci seznamu prvků v podstromech v tom pořadí, v jakém je navštěvujeme. Dále uvedené algoritmy se liší právě v pořadí, v jakém prvky stromu zpracovávají. Při průchodu stromu do hloubky je strom zpracováván „po větvích“, při průchodu stromu do šířky je strom zpracováván „po patrech“. Viz ilustraci na obrázku 10.2. Procedura `dfs` uvedená v programu 10.1 pro průchod stromu do hloubky vrací prázdný seznam pro prázdný strom. Jinak vezme větve stromu a projde je do hloubky (aplikací sebe sama), výsledné seznamy spojí a přidá k nim hodnotu stromu.

Procedura `bfs` uvedená v programu 10.2 pro průchod stromu do šířky využívá pomocnou proceduru `aux`, která si ve svém argumentu pamatuje seznam stromů určených k průchodu. Pokud je tento seznam prázdný, vrací prázdný seznam. Jinak vezme hodnoty za všech stromů v seznamu. V dalším kroku je pak tento seznam tvořen podstromy těchto stromů. Tím jakoby „sestoupíme o jedno patro“.

10.2 Reprezentace množin pomocí uspořádaných seznamů

V sekci 6.5 jsme ukázali, jak se dají reprezentovat konečné množiny pomocí seznamů neobsahujících vícenásobné výskyty prvků (duplicity). V této části předvedeme jinou možnost reprezentace množin. Za množiny budeme považovat seznamy bez vícenásobných výskytů prvků, které jsou navíc uspořádané. Tedy například seznam `(5 8 2 4 9)`, který byl v sekci 6.5 platnou reprezentací množiny, není uspořádaný (podle uspořádání daným predikátem `<=`), a proto není reprezentací množiny pro tuto sekci. Platnou reprezentací této množiny by pak byl seznam `(2 4 5 8 9)`.

Obrázek 10.2. Ukázka průchodu do šířky (vlevo) a do hloubky (vpravo).



Program 10.1. Prohledávání stromu do hloubky.

```
(define dfs
  (lambda (tree)
    (if (null? tree) '()
        (cons (get-val tree)
              (apply append (map dfs (get-branches tree)))))))
```

Toto zpřísnění reprezentace nám umožní psát efektivnější procedury, než jsou ty ze sekce 6.5. Na druhou stranu ale musíme mít dané nějaké rozumné lineární uspořádání (rozumné ve smyslu složitosti – ověření, zda-li je jeden prvek menší nebo roven než druhý musí mít přijatelnou časovou a prostorovou složitost). Také kód procedur pro tuto reprezentaci bude mnohem méně přehledný, což může vést ke vzniku chyb při jejich implementaci. Proto je nutné provádět důsledné testování.

U následujících procedur (`in?`, `cons-set`, `union` a `inter`) budeme uvažovat jen číselné množiny a uspořádání dané predikátem `<=`. U každé z těchto procedur nejdříve naznačíme, jak pracují, uvedeme jejich definici s popisem kódu a příklady aplikace.

První procedura – predikát `in?` – bude využívat uspořádání seznamu k rychlejšímu zjištění nepřítomnosti daného prvku v množině. Tato procedura bude postupně procházet přes prvky seznamu dokud nenarazí na prvek, který je shodný s hledaným prvkem, dokud neprojde všechny prvky, nebo dokud nenarazí na prvek, který je větší než ten hledaný. Právě poslední případ zastavení je přínosem zpřísnění reprezentace množiny. Viz následující definici predikátu `in?`:

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

V této rekurzívní proceduře zastavujeme rekurzi při zjištění, že je seznam prázdný, nebo že je první prvek seznamu větší než hledaný prvek. V tom případě je jisté, že hledaný prvek v množině není, a tak je vrácena nepravda `#f`. Dále zastavujeme rekurzi, pokud je první prvek seznamu shodný s hledaným prvek, a tehdy je vrácena pravda `#t`. Zastavení rekurze je v kódu zahrnuto v prvních dvou větvích speciální formy `cond`.

Program 10.2. Prohledávání stromu do šířky.

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

V případě, že není splněna ani jedna z těchto limitních podmínek, je predikát `in` rekurzivně aplikován na seznam bez prvního prvku. Následují příklady aplikace:

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

Druhou procedurou je konstruktor množiny `cons-set`. Ten má zatřídit zadaný element do dané množiny, pokud už tam tento element není. Ta bude procházet seznam reprezentující množinu podobným způsobem jako predikát `in?`. Přínosem zpřísnění reprezentace je opětně to, že můžeme dříve zjistit nepřítomnost prvku. Implementace této procedury se bude jen málo lišit od implementace predikátu `in?`. Viz definici:

```
(define cons-set
   (lambda (x set)
      (cond ((null? set) (list x))
            ((= x (car set)) set)
            ((<= (car set) x) (cons (car set) (cons-set x (cdr set))))
            (else (cons x set)))))
```

V těle této procedury zastavujeme rekurzi v případě, že je množina prázdná a v tom případě vracíme jednoprvkový seznam obsahující přidaný prvek. Dále zastavujeme rekurzi při zjištění, že je první prvek seznamu shodný s přidávaným elementem a tehdy vracíme původní seznam bez jakékoli změny, protože je v něm přidávaný prvek už obsažen. Poslední limitní podmínkou je skutečnost, že první prvek seznamu je větší, než přidávaný element. Pokud je tato podmínka splněna, znamená to, že jsme našli místo, kam má být prvek přidán. Pokud není splněna žádná z limitních podmínek, pak rozkládáme problém na přidání prvku do seznamu (konstruktorem `cons`), a přidávání prvku do menší množiny (tedy množiny reprezentované strukturálně jednodušším seznamem). Následují příklady použití procedury `cons-set`.

```
(cons-set 3 '())          ⟹  ()
(cons-set 3 '(1 2))       ⟹  (1 2 3)
(cons-set 3 '(1 2 3 4))   ⟹  (1 2 3 4)
(cons-set 3 '(1 2 4))     ⟹  (1 2 3 4)
```

Další dvě procedury uvedené v této sekci představují množinové operace sjednocení a průniku. Protože se jedná o složitější procedury, ukážeme nejdříve to, jak pracují na konkrétním případě a pak až se zaměříme na konkrétní implementaci.

U operace sjednocení množin zpracováváme současně dva seznamy reprezentující množiny. S těmito seznamy provádíme slévání, podobně jako tomu bylo u procedury `merge` v sekci 8.5. Jediným rozdílem je to, že sledujeme i možnost, že by při průběžném zpracování seznamů nastala skutečnost, že tyto seznamy mají stejný první prvek. V tom případě zařadíme tento prvek do výsledného seznamu a odebereme jej *z obou* seznamů. Tím zamezíme vzniku duplicit ve výsledném seznamu, které by nastaly při běžném slévání. Viz příklad:

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 ∪ B: (1 2 3 4 5 6 7)
```

Následuje definice procedury `union`:

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

Limitní podmínkou je test na prázdnost alespoň jednoho ze seznamů. Pokud je tato podmínka splněna, vracíme druhý ze seznamů (ten, který není prázdný). To je vyjádřeno prvními dvěma větvemi speciální formy `cond`. Jinak rozlišujeme tyto možnosti:

- Oba seznamy začínají stejným prvkem. Pak tento společný prvek přidáme na začátek sjednocení množin bez tohoto prvku. To je zahrnuto v třetí větvi formy `cond`.
- Jeden seznam začíná menším prvkem než druhý seznam. V tom případě sjednotíme seznamy bez tohoto nejmenšího prvku, a tento prvek přidáme do výsledku. Tato možnost je vyjádřena posledními dvěma větvemi formy `cond`.

Viz příklady aplikace:

```
(union '() '())                   ⟹  ()
(union '() '(1 2 3 4 5))          ⟹  (1 2 3 4 5)
(union '(2 4 6 7) '())            ⟹  (2 4 6 7)
(union '(2 4 6 7) '(1 2 3 4 5))   ⟹  (1 2 3 4 5 6 7)
```

Podobně pracuje i procedura `inter` na výpočet průniku množin. Prvky do výsledné množiny ale zařazujeme jen v případě, že se vyskytují v obou seznamech. Takové se nám při průběžném zpracování dostanou na začátek seznamu. Viz následující příklad:

```
seznam A      seznam B        prvky zařazené do A ∩ 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 ∩ B: (2 4)
```

Následuje implementace procedury `inter`:

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

Procedura `inter` zastavuje rekurzi v případě, že alespoň jeden ze seznamů je prázdný. V takovém případě je výsledkem prázdná množina. Pokud oba seznamy začínají stejným elementem, přidáme tento společný element do průniku množin bez tohoto elementu. Jinak pouze „odjímáme“ nejmenší prvky ze seznamů.

Viz příklady aplikace:

```
(inter '() '())                 ⟼  ()
(inter '() '(1 2 3 4 5))        ⟼  ()
(inter '(2 4 6 7) '())          ⟼  ()
(inter '(2 4 6 7) '(1 2 3 4 5)) ⟼  (2 4)
```

10.3 Reprezentace množin pomocí binárních vyhledávacích stromů

V této sekci ukážeme další možnost, jak reprezentovat množiny. Jedná se o reprezentaci pomocí binárních vyhledávacích stromů. Binárním vyhledávacím stromem vzhledem k uspořádání $\leq$ rozumíme:

- prázdný strom
- strom, který má právě dva podstromy. Tyto podstromy nazýváme *pravý podstrom* a *levý podstrom*. Jsou-li tyto podstromy neprázdné, musí platit, že hodnota levého podstromu je menší vzhledem k uspořádání $\leq$ než hodnota stromu a hodnota pravého podstromu je větší vzhledem k uspořádání $\leq$ než hodnota stromu.

Binární vyhledávací stromy budeme reprezentovat stejně jako n-ární stromy uvedené v sekci 10.1. Tomu budou odpovídat i konstruktory a selektory pro binární strom.

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

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

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

Pomocí těchto struktur budeme reprezentovat množiny. Napíšeme predikát `in?` zjišťující, zda je zadaný prvek v množině, a konstruktor `cons-tree` přidávající prvek do množiny. Při implementaci těchto procedur budeme uvažovat stromy reprezentující množiny čísel a uspořádání dané predikátem `<=`.

Procedura `in?` zastavuje rekurzi v případě, že zkoumaný strom je prázdný – pak vrací nepravdu – nebo v případě, že hodnota stromu je shodná s hledaným prvkem – pak vrací pravdu. Pokud ani jedna z těchto limitních podmínek podmínek není splněna, pokračuje hledáním prvku v levém nebo v pravém podstromu, to podle porovnání hledaného prvku a hodnoty stromu.

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

Procedura konstrukce množiny reprezentované stromem také zastavuje v případě, že je strom prázdný. Tehdy vrací jednoprvkový strom vytvořený procedurou `make-leaf`. Zastavuje také, pokud je hodnota stromu rovna přidávanému prvku. V tom případě je prvek už ve stromu přítomen a ten se jeho přidáním nezmění, a je tedy vrácen původní strom. Jinak je vytvořen nový strom do jehož levého nebo pravého podstromu (v závislosti na porovnání vkládaného prvku a hodnoty stromu) je vložen přidávaný prvek (použitím procedury `cons-tree`).

Program 10.3. Výpočet potenční množiny.

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

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

10.4 Kombinatorika na seznamech

V této poslední sekci předvedeme několik příkladů na výpočet kombinatorických úloh nad seznamy. Budeme například hledat všechny podmnožiny množiny reprezentované seznamem bez vícenásobných výskytů prvků, všechny permutace, všechny k-prvkové kombinace nebo k-prvkové kombinace prvků zadané množiny.

Výpočet potenční množiny 2^A (množiny všech podmnožin množiny A) vytvoříme podle tohoto rekurzivního přepisu:

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

Pokud zadaná množina $A = \{a_1, a_2, \dots\}$ není prázdná, vypočteme nejdříve potenční množinu její podmnožiny $\{a_2, \dots\}$. Do výsledné množiny pak zařadíme všechny prvky této potenční množiny „jakoby dvakrát“. Jednou to budou přímo tyto množiny, podruhé tyto množiny s přidaným prvkem a_1. Program 10.3 je přímým přepisem právě uvedeného popisu. Všimněte si, že se v programu 10.3 vyskytuje dvakrát výraz `(power-set (cdr set))`. Výpočet můžeme zefektivnit pomocí lokální vazby tak, jak je uvedeno v programu 10.4. Důsledkem, že potenční množinu podmnožiny počítáme jen jednou, bude nejen kratší čas výpočtu, ale také struktura výsledného seznamu bude vypadat jinak (viz obrázek 10.3, vlevo je výsledek pro původní proceduru, vpravo je výsledek pro novou proceduru). Z čistě funkcionálního pohledu na seznamy, který jsme doposud uplatňovali, je to však principiálně jedno.

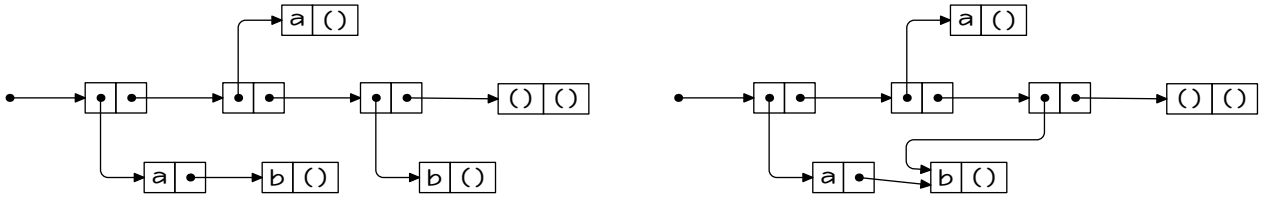
Dále se budeme zabývat výpočtem všech permutací n prvků. Budeme to provádět tak, že postupně projdeme všech n prvků seznamu. Pro každý vytvoříme seznam permutací majících na začátku právě tento prvek. Ty vytvoříme tak, ze najdeme (rekurzí) permutace zbývajících prvků a na jejich začátek přidáme uvažovaný prvek. Tyto seznamy spojíme. Následující schéma zachycuje hledaní všech permutací prvků seznamu `(1 2 3)`:

uvazovany prvek: 1 2 3

Program 10.4. Efektivnější výpočet potenční množiny.

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

Obrázek 10.3. Výsledek aplikace staré a vylepšené verze `power-set` pro argument `(a b)`.




```
zbyvajici prvky:                (2 3)          (3 1)          (1 2)
permutace zbyvajicich prvku:    (2 3)   (3 2)  (3 1) (1 3)    (1 2) (2 1)
permutace zacinajici uv. pr.:   (1 2 3) (1 3 2) (2 3 1) (2 1 3) (3 1 2) (3 2 1)
```

Program 10.5 obsahuje definici právě popsané procedury. Jednotlivé prvky zadaného seznamu `l` jsou procházeny pomocnou procedurou `perm`. Tato procedura si v argumentu `p-set` pamatuje seznam zbylých prvků (to jest prvků, které jsou různé od právě zpracovávaného). Z prvků tohoto seznamu vytvoří permutace (pomocí rekurzivního volání procedury `permutation`), připojí na jejich začátek aktuální prvek, a seznam takto vytvořených permutací připojí k permutacím začínajícím ostatními prvky (procedura `perm` pokračuje zpracováním dalšího prvku).

Můžeme také generovat přímo konkrétní permutace, aniž bychom museli vytvářet všechny. A to přes jejich index. K tomu použijeme takzvanou *faktoradickou soustavu čísel*. Faktoradická číselná soustava je soustava o proměnlivém základu založeném na faktoriálu:

```
zaklad:                    7    6    5   4   3   2   1   0
hodnota pozice:            7!   6!   5!  4!  3!  2!  1!  0!
v dekadicke soustave: 5040  720  120  24  6   2   1   1
```

Prvních šest čísel faktoradické soustavy je ukázáno v prostředním sloupci tabulky 10.4.

Vztah faktoradických čísel a permutací n prvků je velice jednoduchý. My teď popíšeme, jak z faktoradického čísla získat permutaci:

Z n prvků vytvoříme tzv. seznam kandidátů. Číslice, která je v n-ciferné reprezentaci (případně doplněné počátečními nulami) faktoradického čísla nejvíce vlevo označuje index prvku v seznamu kandidátů, který dosadíme na nejlevější neobsazené pozici v n-tici permutace. Vybraný prvek odebereme ze seznamu kandidátů a z reprezentace faktoradického čísla odebereme nejlevější číslici. Zbytek pozic n-tice permutace vyplníme jako permutaci zbývajících kandidátů podle zkráceného faktoradického čísla.

Příklad 10.1. Předveďme popsaný postup na následujícím příkladě

$2_2 0_1 0_0$	$\{a, b, c\}$	$[c, \_, \_]$
$0_1 0_0$	$\{a, b\}$	$[c, a, \_]$
0_0	$\{b\}$	$[c, a, b]$

Program 10.5. Výpočet všech permutací prvků množiny.

```
(define permutation
  (lambda (l)
    (if (null? l)
        '(())
        (let perm
            ((base l)
             (p-set (cdr l)))
          (if (null? base)
              '()
              (append
               (map (lambda (x)
                      (cons (car base) x))
                    (permutation p-set))
               (perm (cdr base)
                     (cdr (append p-set (list (car base))))))))))))
```

Obrázek 10.4. Faktoradická čísla a permutace.

0	$0_2 0_1 0_0$	(0,1,2)
1	$0_2 1_1 0_0$	(0,2,1)
2	$1_2 0_1 0_0$	(1,0,2)
3	$1_2 1_1 0_0$	(1,2,0)
4	$2_2 0_1 0_0$	(2,0,1)
5	$2_2 1_1 0_0$	(2,1,0)

Permutace prvků a, b, c odpovídající faktoradickému číslu $2_2 0_1 0_0$ je permutace $[c, a, b]$. Další příklady jsou v tabulce 10.4.

Při implementaci budeme využívat procedur, které jsme naprogramovali dříve. Konkrétně se jedná o proceduru `fak` (viz programy 7.12, 8.3, 8.5 a 8.6 na stranách 185, 204, 206 a 210 a proceduru `remove` z programu6.3 na straně 146, respektive její efektivní implementaci).

Vyjdeme z jednoduché procedury na převod čísla `dec` v dekadické soustavě na číslo v `bas`-adické soustavě:

```
(define prevod
  (lambda (dec bas)
    (let iter ((dec dec)
               (res '()))
      (if (= dec 0) res
          (iter (quotient dec bas) (cons (modulo dec bas) res))))))
```

Nyní uděláme dvě úpravy (viz program 10.6):

1. na místo pevně zadaného základu bude procedura přijímat proceduru `basf`, která pro index (pozici, počítanou zleva) vrací její základ. To proto, abychom mohli převádět čísla z dekadické soustavy na soustavy s proměnlivým základem.

2. procedura bude vracet přesně `l` čísel (výsledek bude ořezán, nebo budou doplněny počáteční nuly). Číslo `l` bude předáváno proceduře jako další parametr.

Program 10.6 obsahuje definici procedury pro převod `prevod` s právě popsanými úpravami a definici procedury `index->perm` realizující výše popsané hledání permutace náležející k danému indexu.

Viz příklady aplikace:

Program 10.6. Výpočet permutace prvků množiny pomocí faktoradických čísel.

```
(define prevod
  (lambda (dec basf l)
    (let iter ((dec dec)
               (res '())
               (i 1))
      (if (> i l) res
          (let ((bas (basf i)))
            (iter (quotient dec bas)
                  (cons (modulo dec bas) res)
                  (+ i 1)))))))

(define index->perm
  (lambda (i set)
    (let perm ((set set)
               (fnum (prevod i n! (length set))))
      (if (null? set) '()
          (cons (list-ref set (car fnum))
                (perm (remove set (car fnum))
                      (cdr fnum)))))))
```

```
(index->perm 0 '(1 2 3))  ⟼  (1 2 3)
(index->perm 1 '(1 2 3))  ⟼  (1 3 2)
(index->perm 2 '(1 2 3))  ⟼  (2 1 3)
(index->perm 3 '(1 2 3))  ⟼  (2 3 1)
(index->perm 4 '(1 2 3))  ⟼  (3 1 2)
(index->perm 5 '(1 2 3))  ⟼  (3 2 1)
```

Procedura `combination` přijímá dva argumenty – číslo k a množinu S – a vrací seznam všech k-prvkových kombinací prvků množiny S (to jest seznam všech různých k-tic, jejichž složky jsou vzájemně různé prvky množiny S). Pracuje tak, že prochází seznam reprezentující množinu S a pro každý prvek vytvoří dva seznamy kombinací:

1. kombinace obsahující tento prvek. Ty najde tak, že vytvoří $k-1$-prvkové kombinace doposud neuvažovaných prvků a pak do nich přidá právě zpracovávaný prvek.

2. kombinace neobsahující tento prvek. Hledají se tedy k-prvkové kombinace doposud neuvažovaných prvků.

Tyto seznamy se pak spojí dohromady. Viz program 10.7

Malou úpravou kódu z programu 10.7 dostaneme proceduru, která vrací seznam kombinací s opakováním. Program 10.8 se liší od definice procedury `combination-dup` jen v tom detailu, že při rekurzivním volání procedury, jehož výsledkem má být zbytek kombinace, neodebíráme už dosazený prvek z množiny kombinovaných prvků.

Shrnutí

Tato lekce obsahuje pokročilejší příklady sloužící k procvičení práce s hierarchickými daty. V první části lekce se budeme zabývat reprezentací stromů pomocí seznamů a jejich prohledáváním do hloubky a do šířky. Dále ukážeme reprezentaci množin pomocí uspořádaných seznamů a pomocí binárních vyhledávacích stromů. Závěr lekce je věnován kombinatorice na seznamech.

Program 10.7. Výpočet všech kombinací prvků množiny.

```
(define combination
  (lambda (k set)
    (cond ((null? set) '())
          ((= k 0) '(()))
          ((= k 1) (map list set))
          (else (append (map (lambda (x)
                               (cons (car set) x))
                             (combination (- k 1) (cdr set)))
                        (combination k (cdr set)))))))
```

Program 10.8. Výpočet všech kombinací s opakováním.

```
(define combination-dup
  (lambda (k set)
    (cond ((null? set) '())
          ((= k 0) '(()))
          (else (append (map (lambda (x)
                               (cons (car set) x))
                             (combination-dup (- k 1) set))
                        (combination-dup k (cdr set)))))))
```

Pojmy k zapamatování

-
-
-
-

Kontrolní otázky

1.
2.
3.

Cvičení

1.
2.
3.
4.

Úkoly k textu

1.
2.
3.
4.

Řešení ke cvičením

1.
2.
3.