KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 2 ◇ poznámky k přednášce

7. Líné vyhodnocování pro úsporu místa

verze z 4. dubna 2020

V této přednášce se budeme zabývat líným vyhodnocováním z pohledu úspory místa. Ukážeme, jak lze v některých úlohách ušetřit velké množství paměti nahrazením datových struktur funkcemi. Jako datové struktury budeme používat jednorozměrná a dvourozměrná pole.

1 Práce s poli v Lispu

S jedním druhem pole jsme se už setkali: textový řetězec je jednorozměrné pole znaků. Takové pole lze vytvořit pomocí funkce `make-array`:

```
(make-array 10 :element-type 'character :initial-element #\A)
"AAAAAAAAAA"
```

Prvním (povinným) argumentem funkce je zde počet prvků pole. Další, nepovinný, argument `:element-type` určuje, jakého typu budou prvky pole. Když použijeme symbol `character`, říkáme, že prvky budou znaky, takže půjde o jednorozměrné pole znaků, čili textový řetězec. Argument `:initial-element` obsahuje znak, na který budou všechny prvky řetězce inicializovány.

Obecné jednorozměrné pole bez inicializovaných prvků může vypadat třeba takto:

```
CL-USER 2 > (make-array 20)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL)
```

Mohlo by to vypadat, že nové pole má inicializované složky, ale obecně to není pravda a nelze na to spoléhat:

```
CL-USER 15 > (make-array 10 :element-type 'number)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)

CL-USER 16 > (make-array 10 :element-type 'double-float)
#(551.0D0 14.0D0 25.0D0 4.0583D-320 368.0D0 4.0583D-320 551.0D0
4.0583D-320 14.0D0 4.0583D-320)
```

```
CL-USER 17 > (make-array 10 :element-type 'bit)
#*1111111100
```

Proto budeme složky nově vytvořených polí vždy inicializovat. (`number`, `double-float`, `bit` jsou příklady datových typů v Lispu, ukazuji je tady jen pro ilustraci, nebudeme se jimi zabývat.)

Jak vidíme, různé typy polí Lisp může tisknout různou syntaxí. Tyto rozdíly nás nemusejí zajímat.

Jednorozměrným polím také říkáme *vektory*.

K vytvoření vícerozměrného pole použijeme jako první argument seznam čísel určujících velikost jednotlivých dimenzí (stručně těmto číslům říkáme dimenze).

```
CL-USER 3 > (make-array '(5 10))
#2A((NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL))
```

U vícerozměrných polí je už syntax jednotná: začíná znaky `#`*n*`A`, kde *n* je číslo určující *hodnost pole* (*rank*, tak se v Lispu říká počtu dimenzí). Kdybychom si předchozí výsledek uspořádali, dostali bychom tabulku s pěti řádky a desíti sloupci:

```
#2A((NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
    (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
    (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
    (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
    (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL))
```

Funkce `make-array` má další zajímavé nepovinné argumenty. Zájemci si o nich mohou přečíst ve standardu.

Uložme si výsledek posledního vyhodnocení do proměnné:

```
CL-USER 4 > (setf arr *)
#2A((NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL))
```

Ukážeme na příkladech a bez podrobného vysvětlování několik funkcí, které s poli pracují:

```
CL-USER 5 > (array-dimensions arr)
(5 10)

CL-USER 6 > (array-dimension arr 0)
5

CL-USER 7 > (array-dimension arr 1)
10

CL-USER 8 > (array-rank arr)
2
```

Ke čtení a nastavování (pomocí `setf`) jednotlivých prvků pole slouží operátor `aref`:

```
CL-USER 43 > (setf arr2 (make-array '(3 4)))
#2A((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))

CL-USER 44 > (setf (aref arr2 1 2) t)
T

CL-USER 45 > arr2
#2A((NIL NIL NIL NIL) (NIL NIL T NIL) (NIL NIL NIL NIL))

CL-USER 46 > (aref arr2 1 2)
T

CL-USER 47 > (apply #'aref arr2 (list 1 2))
T
```

```
CL-USER 55 > (setf str (make-array 4 :element-type 'character
                                     :initial-element #\0))
"0000"

CL-USER 56 > (setf (aref str 0) #\A
                   (aref str 2) #\o
                   (aref str 3) #\j
                   (aref str 1) #\h)
#\h

CL-USER 57 > str
"Ahoj"
```

2 Cykly

K průchodu polem bývá nejlepší používat cykly, je to někdy pohodlnější než použití rekurze. Se způsobem, jak lze v Lispu cykly psát, jsme se už setkali, ale bez podrobnějšího popisu. Ukážu tedy dva nejjednodušší způsoby, jak lze v Lispu cykly napsat. Téma spadá do oblasti imperativního programování.

Jde o makra `dotimes` a `dolist`. První slouží k iteraci přes všechna čísla větší nebo rovna nule a menší než zadaná hodnota:

```
CL-USER 1 > (dotimes (x (+ 2 3))
              (print x))

0
1
2
3
4
NIL
```

Výraz vyhodnotil podvýraz `(+ 2 3)` a pak vytiskl všechna čísla od `0` menší než získaná hodnota `5`. Jako výsledek vrátil `nil`. (V této podobě makro `dotimes` vždy vrací `nil`.)

Makro `dolist` vytváří cyklus, v němž je daná proměnná navázána postupně na všechny prvky daného seznamu:

```
CL-USER 2 > (dolist (x '(2 -1 3))
              (print x))

2
-1
3
NIL
```

V principu je možné procházet seznam i pomocí makra `dotimes`:

```
(dotimes (x (length list))
  (print (nth x list)))
```

Tento způsob je ovšem velmi nevhodný. Kdo neví proč, měl by si zopakovat, co je to seznam a co tedy uvedený kód přesně dělá.

Poznámka: Makra `dotimes` a `dolist` vždy vytvářejí nové prostředí s novou vazbou řídicí proměnné cyklu (v našich příkladech symbolu `n`). Ve standardu ovšem není řečeno, jestli nové prostředí vznikne jednou a pak se v něm s každou iterací změní hodnota vazby symbolu, nebo jestli prostředí vzniká v každé iteraci znovu.

3 Posloupnosti

Jak jsem řekl na minulé přednášce, s jedním typem líného vyhodnocování jsme se setkali už minulý semestr (konkrétně v 10. přednášce, můžete se k ní vrátit a zopakovat si to), když jsme mluvili o *posloupnostech*. Těmi jsme mysleli matematické číselné posloupnosti a reprezentovali jsme je funkcemi, které pro zadaný index počítaly příslušný člen posloupnosti.

Například posloupnost přirozených čísel jsme reprezentovali funkcí vrácenou výrazem

```
(lambda (n)
  n)
```

a třeba posloupnost sudých čísel:

```
(lambda (n)
  (* 2 n))
```

Abychom takto reprezentované posloupnosti mohli chápat jako datové struktury, napsali jsme si i selektor:

```
(defun mem (seq index)
  (funcall seq index))
```

Teď, když umíme pracovat s makry, můžeme pohodlně napsat i konstruktor posloupností. Bude to makro `seq`, které bude požadovat jako argument jednak symbol pro index a jednak kód, kterým se z dané hodnoty indexu vypočte příslušný člen posloupnosti:

```
(defmacro seq (ind expr)
  `(lambda (,ind) ,expr))
```

Posloupnost přirozených a posloupnost sudých přirozených čísel teď můžeme vytvořit těmito výrazy:

```
(seq n n)
(seq n (* 2 n))
```

Kdyby byla posloupnost datovou strukturou, která své hodnoty obsahuje, musela by být nekonečná. My jsme ji ale zadali jako funkci, která každý člen posloupnosti zjistí (vypočítá), až to bude potřeba.

Ještě připomeňme, že jsme také programovali funkce, které pracovaly s posloupnostmi jako s hodnotami. S využitím makra `seq` můžeme například napsat funkci na součet dvou posloupností takto:

```
(defun seq-+ (seq1 seq2)
  (seq n (+ (mem seq1 n) (mem seq2 n))))
```

Sečíst dvě nekonečné posloupnosti v konečném čase samozřejmě nejde. Nám se to přesto povedlo, jelikož jsme nesčítali příslušné členy obou posloupností, ale pouze napsali funkci, která to v případě potřeby udělá. S posloupnostmi jsme dělali i jiné operace (některé na přednášce, další na zkouškových písemkách) — aniž bychom potřebovali vypočítat hodnotu jediného členu.

Na písemce v minulém semestru byla úloha napsat funkci `seq-map`, která podobně jako funkce `mapcar` pro seznamy aplikuje danou funkci na všechny prvky posloupnosti. Takto může vypadat řešení:

```
(defun seq-map (fun seq)
  (seq n (funcall fun (mem seq n))))
```

Takto bychom mohli z posloupností a a b vytvořit posloupnost obsahující odmocniny ze součtů druhých mocnin jejich prvků, tedy posloupnost s n-tým členem

$$c_n = \sqrt{a_n^2 + b_n^2}\,:$$

```
(seq-map #'sqrt (seq-+ (seq-map (lambda (x) (expt x 2))
                                a)
                       (seq-map (lambda (x) (expt x 2))
                                b)))
```

Jiná možnost:

```
(seq n (sqrt (+ (expt (mem a n) 2)
                (expt (mem a n) 2))))
```

Její nevýhody si také ukážeme za chvíli na bitmapách.

V obou případech jsme definovali posloupnost, aniž bychom počítali její členy. Můžeme říci, že jsme postupnými kroky vytvořili *výpočet* (reprezentovaný funkcí), který je třeba vykonat, chceme-li získat členy posloupnosti.

Poté, co vytvoříme požadovanou posloupnost jakožto výpočet, můžeme chtít získat jejích prvních n členů, např. uložených v jednorozměrném poli. To uděláme selektorem `seq-members`, který projde v cyklu zadaný počet prvních členů posloupnosti a uloží je do pole:

```
(defun seq-members (seq count)
  (let ((result (make-array count)))
    (dotimes (i count)
      (setf (aref result i) (mem seq i)))
    result))
```

A test:

```
CL-USER 40 > (seq-map #'sqrt
                      (seq-+ (seq-map (lambda (x) (expt x 2))
                                      (seq n n))
                             (seq-map (lambda (x) (expt x 2))
                                      (seq n 2))))
#<Closure 1 subfunction of SEQ-MAP 4060000D7C>

CL-USER 41 > (seq-members * 20)
#(2.0 2.236068 2.828427 3.6055513 4.472136 5.3851647 6.3245554
7.28011 8.246211 9.219544 10.198039 11.18034 12.165525 13.152946
14.142136 15.132746 16.124516 17.117243 18.110772 19.104973)
```

Obvyklý způsob práce s posloupnostmi, tedy bez líného vyhodnocování, by vypadal jinak. Nepoužívali bychom líné vyhodnocování, ale posloupnosti bychom reprezentovali jednorozměrnými poli. Posloupnosti by tím pádem nemohly být nekonečné, ale to by nemuselo vadit.

Nebudu tady tento přístup rozebírat, to udělám v dalších odstavcích v příkladu o bitmapách. Jen vás nechám, abyste si sami uvědomili, že způsob bez líného vyhodnocování by znamenal velkou spotřebu paměti. Například zrovna u první varianty posledního příkladu (posloupnost $c_n = \sqrt{a_n^2 + b_n^2}$), než bychom došli k výsledku, museli bychom vytvořit tři pomocné posloupnosti (dvě funkcí `seq-map`, jednu funkcí `seq-+`). Pokud by každá představovala nějaké velké jednorozměrné pole hodnot, byla by spotřeba paměti znatelná. Druhá varianta má zase jiné nevýhody. Všechno rozebereme podrobněji za chvíli.

4 Práce s bitmapami poprvé

Černobílou bitmapu můžeme reprezentovat dvourozměrným polem s hodnotami složek `nil` nebo `t`. Hodnota `nil` znamená, že příslušný pixel má černou barvu, hodnota `t` bílou.

Pro jednoduchost budeme předpokládat, že všechny bitmapy mají stejné rozměry (v realitě by toto nastavení neobstálo):

```
(defvar *bm-height* 250)
(defvar *bm-width* 250)
```

Bitmapa bude datová struktura s konstruktory `bm-black-bitmap` a `bm-bitmap-from-array`, selektory `bm-bit` a `bm-bits` a mutátorem `(setf bm-bit)`:

```
(defun bm-black-bitmap ()
  (make-array (list *bm-width* *bm-height*)
              :initial-element nil))

(defun bm-bitmap-from-array (array)
  array)

(defun bm-bit (bm x y)
  (aref bm x y))

(defun bm-bits (bm)
  bm)

(defun (setf bm-bit) (value bm x y)
  (setf (aref bm x y) value))
```

Jak vidíme, to, že bitmapu reprezentujeme polem, před uživatelem skrýváme. To nám umožní v budoucnu reprezentaci snadno změnit.

Naprogramujeme několik jednoduchých operací s bitmapami, nejprve jejich „míchání“. V našem případě, kdy máme jen dvě hodnoty každého pixelu, půjde čistě o logické operace na jednotlivých bitech.

Toto jsou operace negace, konjunkce a disjunkce bitmap po jednotlivých pixelech. Funkce na negaci akceptuje jednu bitmapu, zbylé dvě dvě. Všechny funkce nejprve vytvoří novou bitmapu a pak ve dvojitém cyklu projdou jednotlivé pixely a pro každý provedou operaci na odpovídajících pixelech vstupních bitmap:

```
(defun bm-not (bm)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (not (bm-bit bm x y)))))
    result))

(defun bm-and-2 (bm1 bm2)
  (let ((result (bm-black-bitmap)))
```

```
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (and (bm-bit bm1 x y) (bm-bit bm2 x y)))))
    result))

(defun bm-or-2 (bm1 bm2)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (or (bm-bit bm1 x y) (bm-bit bm2 x y)))))
    result))
```

Vidíme, že funkce jsou velmi podobné. Bylo by dobré napsat jednu obecnou, kterou by tyto tři funkce (a třeba v budoucnu i další) využily. Nebudeme to ale dělat, abychom neodbíhali od tématu.

Bitmapu můžeme chápat jako množinu, a to množinu bílých pixelů. Hodnota uložená v poli (`t` nebo `nil`) může být chápána jako logická hodnota „pixel leží v množině“. Pokud bitmapu chápeme takto, pak uvedené tři funkce realizují operace doplňku, průniku a sjednocení množin.

Bude se nám hodit zobecnit funkce `bm-and-2` a `bm-or-2` na víc než dvě bitmapy. Tak, aby bylo například možné zjistit sjednocení čtyř bitmap takto:

```
(bm-or bm1 bm2 bm3 bm4)
```

Na to by se hodila funkce `foldr`, kterou známe z 9. přednášky z minulého semestru:

```
(defun foldr (fun list init)
  (if (null list)
      init
    (funcall fun (car list) (foldr fun (cdr list) init))))
```

nebo `foldl`, která tehdy byla za úlohu.

Jejich roli hraje v Lispu vestavěná funkce `reduce`. Místo

```
(foldl fun list init)
```

můžeme v našem případě napsat

```
(reduce fun list :initial-value init)
```

a místo

```
(foldr fun list init)
```

lze napsat

```
(reduce fun list :initial-value init :from-end t)
```

V našem případě je jedno, kterou variantu zvolíme, protože dělat průnik nebo sjednocení množin zleva doprava nebo zprava doleva vede ke stejnému výsledku (průnik i sjednocení množin je komutativní a asociativní).

Funkce pro obecný počet bitmap tedy budou takové:

```
(defun bm-and (bm1 &rest bitmaps)
  (reduce #'bm-and-2 bitmaps :initial-value bm1))

(defun bm-or (bm1 &rest bitmaps)
  (reduce #'bm-or-2 bitmaps :initial-value bm1))
```

Další základní funkce pro bitmapu je posun:

```
(defun bm-shift (bm dx dy)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (if (and (< -1 (- x dx) *bm-width*)
                       (< -1 (- y dy) *bm-height*))
                  (bm-bit bm (- x dx) (- y dy))
                nil))))
    result))
```

Funkce posune hodnoty v bitmapě ve směru osy x o `dx` a ve směru osy y o `dy` (v počítačové grafice ovšem často bývá počátek souřadnic vlevo nahoře a osa y směřuje dolů; tak je tomu i v příkladech k této přednášce). Odkryté pixely (tedy ty, které neodpovídají žádným pixelům původní bitmapy) nastaví na `nil`.

Podrobněji: pixel o souřadnicích (x, y) posunuté bitmapy se nastaví na hodnotu pixelu o souřadnicích $(x-dx, y-dy)$ původní bitmapy. Pokud by ovšem souřadnice $(x - dx, y - dy)$ ležely mimo původní bitmapu, nastaví se na `nil`.

Všimněte si, že funkce `bm-and`, `bm-or`, `bm-not` a `bm-shift` používají ve svém těle imperativní prostředky (nastavují v cyklu všechny pixely bitmapy), ale navenek fungují čistě funkcionálně: návratovou hodnotu vytvoří a vypočítají z hodnot svých argumentů. Nemají žádný vedlejší efekt a ani nijak nemodifikují své argumenty (nejsou *destruktivní*).

To je velká výhoda těchto funkcí, která se projeví i v tom, jak hezky a jednoduše lze s jejich pomocí napsat další funkce, například:

```
(defun bm-diff (bm1 bm2)
  (bm-and bm1 (bm-not bm2)))

(defun bm-top-edge (bm)
  (bm-diff bm (bm-shift bm 0 5)))

(defun bm-edges (bm)
  (bm-or (bm-diff bm (bm-shift bm 0 5))
         (bm-diff bm (bm-shift bm 0 -5))
         (bm-diff bm (bm-shift bm 5 0))
         (bm-diff bm (bm-shift bm -5 0))))
```

Význam a použití těchto funkcí jsem graficky demonstroval na přednášce.

Jejich zdrojový kód je pro programátora srozumitelný bez dalšího vysvětlování. Nepoužívá složité a nepřehledné cykly, je v něm napsáno přesně to, co měl autor na mysli, bez zbytečného balastu kolem. Tedy: rozdíl množin je roven průniku první množiny s doplňkem druhé; horní okraj obrázku lze zjistit tak, že od původního obrázku odečtu obrázek mírně posunutý dolů; okraj obrázku získám sjednocením horního, levého, dolního a pravého okraje. (V reálném kódu bychom ve funkcích místo hodnoty `5` použili parametr.)

Shrňme hlavní výhody našeho způsobu reprezentace bitmap a práce s nimi:

1. S bitmapami pracujeme pohodlně čistě funkcionálním stylem, nemusíme se trápit s vedlejším efektem.
2. Kromě základních funkcí, které používají iteraci, jsou ostatní funkce napsány přehledně a čitelně.

Naprogramované funkce jsou k dispozici v kódu k této přednášce. Bitmapy si také můžete prohlížet pomocí jednoduchého prohlížeče, který je rovněž k dispozici (dokumentace je na začátku jeho zdrojového souboru).

5 Nevýhoda

Uvedený způsob reprezentace bitmap a práce s nimi má ovšem nevýhody, které jej činí nepoužitelným pro náročnější aplikace. Problém je v enormních nárocích na

paměť, které také znamenají výrazné zpomalení výpočtů. Během práce s bitmapami totiž vzniká mnoho pomocných bitmap, které zabírají hodně místa v paměti.

Kolik místa zabírá jedna bitmapa? To můžeme snadno zjistit:

```
CL-USER 3 > (time (progn (bm-black-bitmap) nil))
Timing the evaluation of (PROGN (BM-BLACK-BITMAP) NIL)

User time = 0.000
System time = 0.000
Elapsed time = 0.001
Allocation = 500072 bytes
0 Page faults
NIL
```

Na mém počítači je to tedy zhruba 500000 bajtů (vychází to pokaždé trochu jinak), což odpovídá osmi bajtům na jeden prvek pole: $250 \cdot 250 \cdot 8 = 500000$.

Poznámka: 8 B proto, že mám 64 bitové LispWorks. U vás to pravděpodobně vyjde 4 B. To je samozřejmě i tak zbytečně mnoho. Alokaci na jednu bitmapu by šlo optimalizovat, a to dokonce až tak, že by každý prvek pole zabíral jen jeden bit (volbou `:element-type 'bit`; což by ale přineslo jiné potíže). To ale není účelem této přednášky — zabýváme se především principy, nikoli technickými problémy.

Při práci s bitmapami jich ovšem vzniká mnohem více. Podívejme se například na funkci `bm-top-edge`. Funkce `bm-shift`, kterou používá, vrací jako výsledek novou bitmapu. Další volaná funkce, `bm-diff`, volá ještě funkci `bm-not`, která vytváří další bitmapu, a sama pak ještě jednu novou bitmapu vrací. Funkce `bm-top-edge` tedy celkově vytváří tři nové bitmapy, z nichž dvě okamžitě zase zahodí a třetí vrátí jako výsledek. Funkce `bm-edges` vytváří 15 bitmap, z nichž 14 hned zahodí.

Jde o problém, na který ve funkcionálním programování můžeme narazit: pohodlnost a bezpečnost programu je vykoupena vysokými nároky na paměť. To se děje už v jednoduchých situacích: pokud chceme třeba vypočítat skalární součin dvou vektorů reprezentovaných seznamy, můžeme napsat elegantně

```
(apply #'+ (mapcar #'* vec1 vec2))
```

ale v některých případech (při velkém počtu velkých vektorů) může být problém, že funkce `mapcar` vytváří nový seznam, který vlastně vůbec nepotřebujeme, protože se nepoužije na nic jiného, než na argument funkce `apply`. V mnoha případech se tímto problémem zabývat nemusíme, protože se vůbec neprojeví, ale někdy je to jinak. (Případ se skalárním součinem a jemu podobné by ovšem mohl inteligentně vyřešit kompilátor, což by bylo nejlepší.)

Stojí za to se zamyslet, jak skalární součin napsat tak, aby zbytečně vytvářený seznam nevznikal. Šlo by to například pomocí cyklu:

```
(let ((result 0))
  (dolist ((x vec1))
    (incf result (* x (pop vec2))))
  result)
```

(Schválně jsem tady použil makra `incf` a `pop` abyste si je osvěžili.)

Doufám, že uznáte, že tato varianta je delší a méně přehledná než varianta původní.

Druhou možností by bylo použít *destruktivní* variantu funkce `mapcar`. Je to funkce `map-into`, která nevytváří nový seznam, ale výsledky zapisuje do zadaného již existujícího seznamu:

```
(apply #'+ (map-into vec1 #'* vec1 vec2))
```

O destruktivních funkcích jsme už mluvili. Jejich nevýhody jsou jasné: po jejich aplikaci už nemůžeme dále pracovat s hodnotami, které byly jejich argumenty.

Porovnání:

```
CL-USER 77 > (setf vec1 (list 1 0 -1 3)
                   vec2 (list 0 2 2 1))
(0 2 2 1)

CL-USER 78 > (apply #'+ (mapcar #'* vec1 vec2))
1

CL-USER 79 > vec1
(1 0 -1 3)

CL-USER 80 > vec2
(0 2 2 1)

CL-USER 81 > (apply #'+ (map-into vec1 #'* vec1 vec2))
1

CL-USER 82 > vec1
(0 0 -2 3)
```

Podobně se můžeme pokusit optimalizovat funkce na práci s bitmapami. Takto by mohla například vypadat funkce `bm-top-edge`, která by nepoužívala pomocnou bitmapu:

```
(defun bm-top-edge (bm)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (if (< -1 (- y 5))
                  (if (bm-bit bm x (- y 5))
                      nil
                    (bm-bit bm x y))
                (bm-bit bm x y)))))
    result))
```

Tento způsob má samozřejmě hromadu nevýhod. Hlavní nevýhodou asi je, že kdykoli bychom potřebovali napsat další funkci na práci s bitmapami, museli bychom se s ní trápit stejně nebo více než jako před chvilkou já s touto funkcí. Jak tímto stylem napsat například funkci `bm-edges`, nechci ani pomyslet. Elegance a jednoduchost předchozího řešení, kdy bylo možné složitější funkce psát jednoduše a čitelně pomocí už napsaných, je pryč. Nemluvím ani o tom, že při tomto způsobu psaní je v podstatě nemožné nedělat chyby, což je nejkritičtější problém tvorby softwaru.

Další, trochu lepší možností by mohlo být napsat funkce *destruktivně*, aby nevytvářely nové bitmapy, ale nové hodnoty ukládaly do jedné z bitmap, které jsou jejich argumentem.

Destruktivní verze funkce `bm-and-2`:

```
(defun bm-and-2 (bm1 bm2)
  (dotimes (y *bm-height*)
    (dotimes (x *bm-width*)
      (setf (bm-bit bm1 x y)
            (and (bm-bit bm1 x y) (bm-bit bm2 x y)))))
  bm1))
```

Vidíme, že funkce přepisuje první ze zadaných bitmap. Je tedy opravdu destruktivní, ale zato nezatěžuje paměť alokací nové bitmapy.

Destruktivní verze funkcí `bm-not` a `bm-or-2` bychom napsali stejně. Destruktivní verzi funkce `bm-shift` lze také napsat, ale je třeba dát pozor, aby si funkce nepřepisovala pixely, které bude ještě potřebovat. Zdrojový kód se tedy zkomplikuje. Už to nebudu psát, zkuste si to představit.

Funkci `bm-diff` můžeme nechat, jak je. Všimněte si ale, že destruuje oba své argumenty!

Co s funkcí `bm-top-edge`? Její původní verze je tato:

```
(defun bm-top-edge (bm)
  (bm-diff bm (bm-shift bm 0 5)))
```

Tu ovšem nemůžeme použít. Proč? Protože destruktivní funkce `bm-shift` modifikuje bitmapu v parametru `bm`. Ta by se pak chybně použila i jako první argument funkce `bm-diff` a výsledkem volání funkce `bm-top-edge` by byla vždy prázdná bitmapa.

Bitmapu `bm` tedy musíme nejdřív duplikovat. Napíšeme si na to funkci:

```
(defun bm-copy (bm)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (bm-bit bm x y))))
    result))
```

Destruktivní verze funkce `bm-top-edge` bude následující:

```
(defun bm-top-edge (bm)
  (bm-diff bm (bm-shift (bm-copy bm) 0 5)))
```

Alokaci jedné pomocné bitmapy jsme se tedy nevyhnuli, ale stále je to o jednu méně, než u původní funkce.

U funkce `bm-edges` si musíme dát pozor, které bitmapy kopírovat (v Lispu se argumenty funkcí vyhodnocují zásadně zleva doprava):

```
(defun bm-edges (bm)
  (bm-or (bm-diff (bm-copy bm) (bm-shift (bm-copy bm) 0 5))
         (bm-diff (bm-copy bm) (bm-shift (bm-copy bm) 0 -5))
         (bm-diff (bm-copy bm) (bm-shift (bm-copy bm) 5 0))
         (bm-diff bm (bm-shift (bm-copy bm) -5 0))))
```

Takové řešení nás ovšem sotva uspokojí.

Shrňme si naše dosavadní úvahy: Původní elegantní a jednoduchá implementace práce s bitmapami má nevýhodu velkého nároku na paměť, který způsobuje i zpomalení programu. Dvě navržená řešení mají nevýhodu v tom, že značně znepřehledňují zdrojový kód, komplikují tvorbu nových funkcí a (co je nejhorší) zvyšují riziko chybovosti . . . a ta lepší stále vytváří pomocné bitmapy.

6 Reprezentace bitmap funkcemi

Uvedený problém se dá vyřešit použitím pokročilejších technik funkcionálního programování. Jednoduše řečeno, bitmapa nemusí obsahovat dvojrozměrné pole s jednotlivými bity, ale pouze informaci o výpočtu, po jehož spuštění se bity vypočítají, a to až v momentě, kdy budou opravdu potřeba. Jde tedy o *odložení výpočtu* na pozdější dobu, neboli použití techniky líného vyhodnocování.

Bitmapu nebudeme reprezentovat dvourozměrným polem, ale *funkcí*, která vrací hodnotu pixelu na místě daném dvojicí argumentů.

Toto je konstruktor černé bitmapy:

```
(defun lbm-black-bitmap ()
  (lambda (x y)
    (declare (ignore x y))
    nil))
```

(Řádku s deklarací si nevšímejte; nemá vliv na funkčnost. Jen potlačuje warning o nepoužitých parametrech `x` a `y`. Pokud ho vynecháte, bude funkce fungovat stejně, ale při kompilaci dostanete warningy.)

Funkci by také šlo elegantně napsat pomocí funkce `constantly`:

```
(defun lbm-black-bitmap ()
  (constantly nil))
```

Konstruktor, který vytváří bitmapu z dvojrozměrného pole:

```
(defun lbm-bitmap-from-array (arr)
  (lambda (x y)
    (aref arr x y)))
```

Selektor, který vrací hodnotu bitu zadaného pixelu:

```
(defun lbm-bit (lbm x y)
  (funcall lbm x y))
```

Selektor vracející celou bitmapu ve dvojrozměrném poli:

```
(defun lbm-bits (lbm)
  (let ((result (make-array (list *lbm-width* *lbm-height*))))
    (dotimes (y *lbm-width*)
```

```
      (dotimes (x *lbm-height*)
        (setf (aref result x y)
              (lbm-bit lbm x y))))
    result))
```

Tento selektor je jediná funkce, která opravdu vytváří celou bitmapu jakožto dvourozměrné pole. Dokud skutečnou fyzickou podobu bitmapy nepotřebujeme, pamatujeme si pouze výpočty, které ji eventuelně vytvoří. Tak používáme techniku líného vyhodnocování. (Mimochodem, jistě jste pochopili, že písmeno `l` v předponě `lbm` znamená „lazy“.)

Tato reprezentace bitmap nemá žádný mutátor.

K pochopení následujících funkcí na práci s bitmapami je třeba mít stále na paměti, že bitmapy reprezentujeme funkcemi.

```
(defun lbm-not (lbm)
  (lambda (x y)
    (not (lbm-bit lbm x y))))

(defun lbm-and-2 (lbm1 lbm2)
  (lambda (x y)
    (and (lbm-bit lbm1 x y)
         (lbm-bit lbm2 x y))))

(defun lbm-or-2 (lbm1 lbm2)
  (lambda (x y)
    (or (lbm-bit lbm1 x y)
        (lbm-bit lbm2 x y))))

(defun lbm-and (lbm1 &rest bitmaps)
  (reduce #'lbm-and-2 bitmaps :initial-value lbm1))

(defun lbm-or (lbm1 &rest bitmaps)
  (reduce #'lbm-or-2 bitmaps :initial-value lbm1))

(defun lbm-shift (lbm dx dy)
  (lambda (x y)
    (let ((old-x (- x dx))
          (old-y (- y dy)))
      (if (and (< -1 old-x *lbm-width*)
               (< -1 old-y *lbm-height*))
          (lbm-bit lbm old-x old-y)
        nil))))
```

Další funkce už můžeme psát stejně jako dříve:

```
(defun lbm-diff (lbm1 lbm2)
  (lbm-and lbm1 (lbm-not lbm2)))

(defun lbm-top-edge (lbm)
  (lbm-diff lbm (lbm-shift lbm 0 5)))

(defun lbm-edges (lbm)
  (lbm-or (lbm-diff lbm (lbm-shift lbm 0 5))
          (lbm-diff lbm (lbm-shift lbm 0 -5))
          (lbm-diff lbm (lbm-shift lbm 5 0))
          (lbm-diff lbm (lbm-shift lbm -5 0))))
```

Žádná z operací na bitmapách, které jsme teď napsali, nevytváří dvourozměrné pole. Vytváří pouze (operátorem `lambda`) lexikální uzávěr, který zabírá mnohem méně místa.

Teprve až je bitmapa hotová a chceme ji zobrazit, stačí dvourozměrné pole vytvořit funkcí `lbm-bits`. Tedy pouze jednou a úplně nakonec.

Tím jsme konečně dosáhli výsledku, který se nám v předchozích pokusech nedařil: zdrojový kód operací s bitmapami je krátký, jednoduchý a čitelný (to se týká hlavně odvozených funkcí, jako jsou `lbm-diff`, `lbm-top-edge` a `lbm-edges`), nehrozí v něm tak velké nebezpečí chyb, a současně funkce nejsou náročné na paměť.

Otázky a úlohy na cvičení

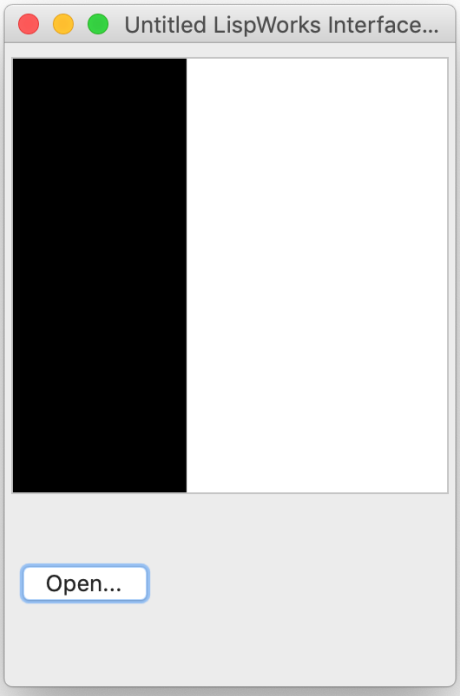
1. Na konci kapitolky o cyklech je poznámka o tom, že není stanoveno, jestli se v makru `dotimes` a `dolist` vytvoří jedno prostředí s vazbou řídicí proměnné cyklu a hodnota vazby se s každou iterací mění, nebo zda se pokaždé vytváří nová vazba. Jak experimentálně ověřit, která z možností platí?

2. Napište funkce `bm-xor` a `lbm-xor`, které spojí dvě bitmapy operací `xor` provedenou na jejich pixelech. V případě funkce `bm-xor` zkuste napsat jak variantu s vnořenými cykly, tak variantu elegantní, používající už napsané funkce.

3. Destruktivní verzi funkce `bm-top-edge` na straně 15 lze změnit tak, aby funkce sama nebyla destruktivní. Zkuste tuto změnu navrhnout.

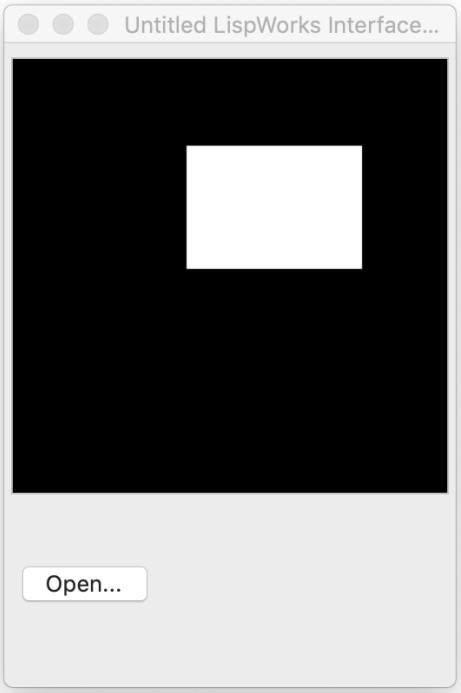
4. Pomocí výsledku předchozí úlohy upravte destruktivní verzi funkce `bm-edges` tak, aby se výrazně omezil počet kopírování vstupní bitmapy.

5. Napište konstruktory bitmap `bm-left-half-plane`, `bm-right-half-plane`, `bm-upper-half-plane` a `bm-lower-half-plane`, které zobrazí levou, pravou, horní resp. dolní polorovinu (polorovina je bílá, pozadí černé). Okraj poloroviny bude vždy dán souřadnicí, která bude parametrem příslušné funkce. Takto by se měla zobrazit bitmapa vzniklá vyhodnocením výrazu `(bm-right-half-plane 100)`:



6. Totéž udělejte pro líné bitmapy (funkce s předponou „`lbm-`“).

7. Jak pro klasické, tak pro líné bitmapy napište konstruktor obdélníka `bm-rectangle` (`lbm-rectangle`), který bude zadán souřadnicí levého, horního, pravého a dolního okraje. Obdélník implementujte jako průnik polorovin. Takto bude vypadat obdélník zkonstruovaný výrazem `(lbm-rectangle 100 50 200 120)`:



8. Pomocí makra `time` porovnejte paměťové nároky naprogramovaných funkcí.