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. Funkcionální grafika

verze z 19. května 2019

1 Pole

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

```
CL-USER 16 > (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 17 > (make-array 20)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL)
```

Pro tuto přednášku budou důležitá pole s prvky typu `bit`. V Lispu je to typ, který obahuje dvě hodnoty: číslo `0` a číslo `1`.

```
CL-USER 18 > (make-array 20 :element-type 'bit)
#*00001110110001000100
```

Jak vidíme, každý z typů polí Lisp tiskne trochu jinou syntaxí. Tyto rozdíly nás nyní nemusejí zajímat. Dále si všimněme, že jelikož jsme pole nenechali inicializovat, jsou jeho prvky náhodné, protože LispWorks při alokaci bloku paměti v ní nechal hodnoty, které tam už byly.

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

```
CL-USER 19 > (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))
```

Nás budou v této přednášce zajímat dvourozměrná pole s prvky typu `bit`:

```
CL-USER 20 > (make-array '(7 9) :element-type 'bit)
#2A((0 0 0 0 0 0 1 0 1) (1 0 1 1 0 1 0 1 0) (0 0 0 1 0 1 0 1 0) (0
0 1 0 0 0 1 1 0) (0 1 1 1 0 0 0 0 0) (0 0 0 0 0 0 0 0 0) (0 0 0 0 0
0 0 0 0))
```

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 21 > (setf arr *)
#2A((0 0 0 0 0 0 1 0 1) (1 0 1 1 0 1 0 1 0) (0 0 0 1 0 1 0 1 0) (0
0 1 0 0 0 1 1 0) (0 1 1 1 0 0 0 0 0) (0 0 0 0 0 0 0 0 0) (0 0 0 0 0
0 0 0 0))
```

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

```
CL-USER 22 > (array-dimensions arr)
(7 9)

CL-USER 23 > (array-dimension arr 0)
7

CL-USER 24 > (array-dimension arr 1)
```

```
9

CL-USER 25 > (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 48 > (setf (apply #'aref arr2 (list 1 2)) 100)
100

CL-USER 49 > arr2
#2A((NIL NIL NIL NIL) (NIL NIL 100 NIL) (NIL NIL NIL NIL))
```

```
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

Následující téma spadá do oblasti **imperativního programování**. S cykly jste se seznámili v jiných předmětech, tady jen stručně popíšu dva nejjednodušší způsoby, jak lze v Lispu cykly napsat. (Jejich použití je někdy pohodlnější než použití rekurze.)

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.

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 Práce s bitmapami poprvé

Černobílou bitmapu můžeme reprezentovat dvourozměrným polem s hodnotami typu `bit`. Hodnota `0` znamená, že příslušný pixel má černou barvu, hodnota `1` bílou.

Pro jednoduchost předpokládejme, že všechny bitmapy mají stejné rozměry (v realitě by toto rozhodnutí 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*)
              :element-type 'bit
              :initial-element 0))

(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))
```

Zaměříme se nejprve na operace s bitmapami, konkrétně jejich míchání. V našem jednoduchém případě, kdy máme jen dvě hodnoty každého pixelu, půjde čistě o logické operace na jednotlivých bitech.

Ve zdrojovém kódu jsou napsané dvě verze logických operací. Druhá vznikla z první **abstrakcí**, kdy jsme si všimli, že všechny operace jsme napsali stejným způsobem a vytvořili jsme tedy pro ně jednu funkci, ve které je konkrétní operace dána parametrem. (Podrobnosti ve zdrojovém kódu.)

Obecná funkce vypadá takto:

```
(defun bm-op (fun bm1 &rest bitmaps)
  (let ((result (bm-black-bitmap)))
    (dotimes (y *bm-height*)
```

```
      (dotimes (x *bm-width*)
        (setf (bm-bit result x y)
              (apply fun (mapcar (lambda (bm)
                                   (bm-bit bm x y))
                                 (cons bm1 bitmaps))))))
    result))
```

Toto je zdrojový kód pro konjunkci, disjunkci a negaci (pokud chápeme bitmapu jako množinu bílých pixelů, pak jde o průnik, sjednocení a doplněk):

```
(defun bm-and (bm1 &rest bitmaps)
  (apply #'bm-op #'min bm1 bitmaps))

(defun bm-or (bm1 &rest bitmaps)
  (apply #'bm-op #'max bm1 bitmaps))

(defun bm-not (bm)
  (bm-op (lambda (b) (- 1 b)) bm))
```

Na přednášce jsem ukazoval trochu jinou implementaci, ale tahle je snad srozumitelnější. Konjunkci logických hodnot vyjádřeních čísly `0` a `1` lze totiž vypočítat funkcí `min`, disjunkci funkcí `max` a negaci funkcí vrácenou výrazem

```
(lambda (b)
  (- 1 b))
```

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))
                0))))
    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 `0` (na přednášce jsem je nastavoval na `1`, to ale nebyl dobrý nápad).

Všimněte si, že funkce `bm-op` (na které jsou založeny funkce `bm-and`, `bm-or` a `bm-not`) i funkce `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))))
```

Zdrojový kód těchto funkcí 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. (V reálném kódu bychom 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.

4 Nevýhody

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.

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 devět bitmap, z nichž osm hned zahodí.

Jedna bitmapa přitom zabírá více než $(250 \cdot 250)/8 = 7812.5$ bajtů.

Na přednášce jsem ukazoval nějaké experimenty pomocí makra `time`. Tady je ukazovat nebudu, protože LispWorks zdá se v některých případech alokuje mnohem víc paměti, než je nutné, a já nevím přesně proč. Spokojíme se tedy s teoretickým popisem problému.

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)) 1)
                      0
                    (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 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-op`:

```
(defun bm-op (fun bm1 &rest bitmaps)
  (dotimes (y *bm-height*)
    (dotimes (x *bm-width*)
      (setf (bm-bit bm1 x y)
            (apply fun (mapcar (lambda (bm)
                                 (bm-bit bm x y))
                               (cons bm1 bitmaps))))))
  bm1)
```

Vidíme, že funkce přepisuje svůj první argument (uložený v parametru `bm1`). Je tedy opravdu destruktivní, ale zato nezatěžuje paměť alokací nové bitmapy.

Destruktivní verze funkcí `bm-not`, `bm-and` a `bm-or` bychom nechali jako dříve pomocí funkce `bm-op`. 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.

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 (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:

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

5 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 tzv. **líného vyhodnocování**. O ní ještě bude podrobněji řeč na dalších přednáškách.

Bitmapu nebudeme reprezentovat 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))
    0))
```

(Jak už jsme si říkali, řádek s deklarací nemá vliv na funkčnost. Zde jen potlačuje warning o nepoužitých parametrech `x` a `y`.) Funkci by také šlo elegantně napsat pomocí funkce `constantly`:

```
(defun lbm-black-bitmap ()
  (constantly 0))
```

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*)
                            :element-type 'bit)))
    (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. (Funkce `reduce` je obdobou procedury `foldr` z minulého semestru. Používáme ji na vytvoření operace s libovolným počtem argumentů z operace dvou argumentů.)

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

(defun lbm-and-2 (lbm1 lbm2)
```

```
  (lambda (x y)
    (min (lbm-bit lbm1 x y)
         (lbm-bit lbm2 x y))))

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

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

(defun lbm-or (lbm1 &rest more-lbms)
  (reduce #'lbm-or-2 more-lbms :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)
        0))))
```

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 napsali, nevytváří dvourozměrné pole. Vytváří pouze (operátorem `lambda`) lexikální uzávěr, který zabírá mnohem méně paměťového prostoru.

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.

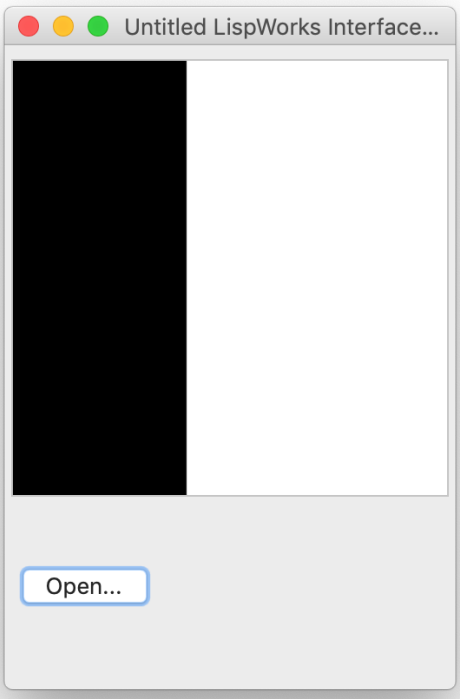
Otázky a úkoly na cvičení

1. Na konci kapitoly o cyklech je poznámka o tom, že není stanoveno, jestli se v makru `dotimes` a `dolist` vytvoří jedno prostředí s vazbou řícicí 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. Destruktivní verzi funkce `bm-top-edge` na straně 11 lze změnit tak, aby funkce sama nebyla destruktivní. Zkuste tuto změnu navrhnout.

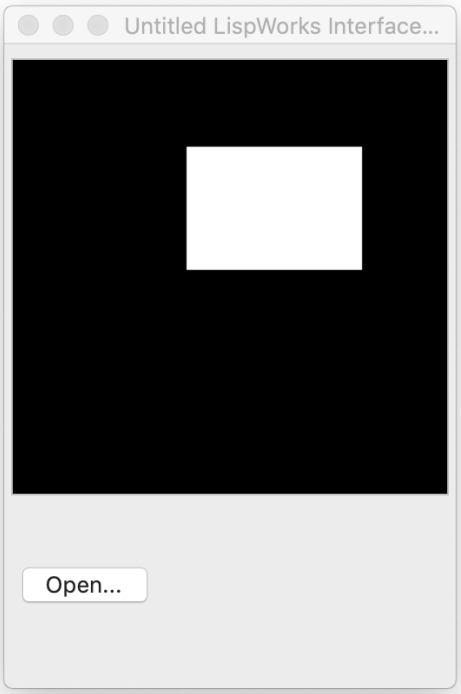
3. Pomocí výsledku předchozího příkladu upravte destruktivní verzi funkce `bm-edges` tak, aby se výrazně omezil počet kopírování vstupní bitmapy.

4. 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)`:



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

6. 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)`:



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