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 1◇ poznámky k přednášce

5. Páry a seznamy

verze z 11. listopadu 2019

1 Datové struktury a datová abstrakce

Funkce, které jsme dosud programovali, pracovaly s jednoduchými daty: čísly. V programech je ale potřeba používat i data složená. Hlavním důvodem je opět potřeba abstrakce, tentokrát **datové abstrakce**.

Uveďme si příklad (vychází ze starší úlohy na cvičení). Při psaní jakékoliv funkce, která pracuje s body v rovině, musíme zatím body zadávat pomocí dvou parametrů — jejich první a druhé souřadnice:

```
(defun point-distance (A-x A-y B-x B-y)
  (sqrt (+ (expt (- A-x B-x) 2)
           (expt (- A-y B-y) 2))))
```

(Funkce `expt` je funkce Common Lispu, která odpovídá funkci `power`, již jsme programovali na minulých přednáškách.)

Pokud chceme vypočítat vzdálenost dvou bodů, musíme je tedy zadat po souřadnicích:

```
CL-USER 3 > (point-distance 2 -1 5 3)
5.0
```

Použití funkce by bylo jistě jednodušší, kdyby umožňovala místo souřadnic bodů zadávat přímo body. Kdyby například proměnné `A` a `B` obsahovaly dva body (ať už vytvořené jakkoli), napsali bychom prostě

```
CL-USER 5 > (point-distance A B)
5.0
```

Toto a další vlastnosti bodů, které ještě uvedeme, znamená, že chceme, aby body byly **hodnoty**. Hodnoty

1. mohou být hodnotami vazeb symbolů,
2. dají se na ně aplikovat funkce,

3. mohou být výsledky vyhodnocení výrazů.

Aniž si zatím řekneme, jak přesně by body byly implementovány, ukážeme si, jak by se s nimi dalo pracovat.

Nový bod bychom vytvořili pomocí funkce `point`:

```
CL-USER 7 > (setf A (point 2 -1))
CL-USER 8 > (setf B (point 5 3))
```

(Zatím nebudu ukazovat výsledek vyhodnocení těchto výrazů.)

Jednotlivé souřadnice bodů bychom zjišťovali pomocí funkcí `point-x` a `point-y`:

```
CL-USER 9 > (point-x A)
2

CL-USER 10 > (point-y A)
-1
```

Funkci `point` říkáme *konstruktor* bodu, funkcím `point-x` a `point-y` *selektory*.

Pomocí uvedených funkcí můžeme přepsat funkci `point-distance` tak, aby pracovala s našimi body:

```
(defun point-distance (A B)
  (sqrt (+ (expt (- (point-x A) (point-x B)) 2)
           (expt (- (point-y A) (point-y B)) 2))))
```

a otestovat ji:

```
CL-USER 11 > (point-distance A B)
5.0

CL-USER 12 > (point-distance (point 2 5) A)
6.0
```

Díky konstruktoru `point` můžeme psát funkce, které vracejí nové body. Například následující funkce vrací bod ve středu úsečky dané koncovými body:

```
(defun segment-center (pt1 pt2)
  (point (/ (+ (point-x pt1) (point-x pt2)) 2)
         (/ (+ (point-y pt1) (point-y pt2)) 2)))
```

Test:

```
CL-USER 13 > (point-x (segment-center A B))
7/2
```

Všimněme si, že k tomu, abychom s body mohli pracovat, nepotřebujeme vědět, jak jsou konstruktor `point` a selektory `point-x` a `point-y` napsané. To je důležitý moment, který je dán tím, že používáme tzv. **datovou abstrakci**: pracujeme s nějakými daty, aniž bychom věděli, jak jsou implementována.

Bod, se kterým by se pracovalo tak, jak jsme uvedli, je příkladem **abstraktní datové struktury**

Datová struktura

Datová struktura je hodnota, které se skládá z více dalších hodnot. Datovou strukturu vytváříme pomocí funkcí zvaných *konstruktory* a hodnoty uvnitř datové struktury získáváme pomocí *selektorů* (zvaných také *přístupové funkce*).

Při práci s datovými strukturami se někdy používají ještě *mutátory*, což jsou funkce, pomocí nichž lze měnit hodnoty uložené v datové struktuře. V tomto semestru mutátory používat nebudeme — jednou vytvořenou datovou strukturu už nebudeme měnit. Věnujeme se totiž tzv. *funkcionálnímu programování*, kde jsou takové postupy nepřípustné. Podobně také nepoužíváme operátor `setf` — s jedinou výjimkou, a to za účelem experimentování v Listeneru.

Abstraktní datová struktura

Datovou strukturu nazýváme *abstraktní*, pokud nevíme, jak je implementována, a k práci s ní nepoužíváme nic jiného než konstruktory a selektory.

To, že body používáme jako abstraktní datové struktury, znamená, že se nezabýváme tím, *co* to bod je, ale stačí nám, že víme, *jak* se s ním pracuje. To nám

1. zjednodušuje práci — nemusíme se starat o to, jak jsou body implementovány, neboli jaká je jejich *datová reprezentace*,
2. případná budoucí změna datové reprezentace bodů neovlivní způsob, jak s nimi pracujeme; stačí změnit konstruktory a selektory (ukážeme za chvíli),
3. zvyšuje čitelnost kódu (ukážeme za chvíli).

Samozřejmě je nutné, aby někdo vhodnou datovou reprezentaci bodů navrhl a funkce `point`, `point-x` a `point-y` naprogramoval. Tomu se budeme věnovat dále.

2 Pár jako nejjednodušší datová struktura

Tečkový pár

Tečkový pár (stručně *pár*) je datová struktura, která se skládá ze dvou složek, nazývaných (z historických důvodů) *car* a *cdr* (výslovnost: *kar* a *kudr*). Tečkové páry se zapisují do kulatých závorek, složky jsou odděleny tečkou.

K vytvoření nového páru slouží funkce `cons`:

```
CL-USER 1 > (cons 1 2)
(1 . 2)
```

Aplikace funkce `cons` na dva argumenty vrátí jako výsledek tečkový pár se složkou *car* rovnou prvnímu a složkou *cdr* druhému argumentu. Funkce `cons` je tedy konstruktorem tečkových párů.

Podle této funkce se v Lispu také často tečkovým párům říká *cons* (jako podstatné jméno).

Ke zjištění složek párů slouží funkce `car` a `cdr`:

```
CL-USER 2 > (car (cons 3 4))
3

CL-USER 3 > (cdr (cons 3 4))
4
```

Jsou to tedy selektory tečkových párů.

Predikát `consp` zjišťuje, zda je daná hodnota pár:

```
CL-USER 4 > (consp 1)
NIL

CL-USER 5 > (consp t)
NIL

CL-USER 6 > (consp (cons 5 6))
T
```

3 Reprezentace bodů pomocí párů

Tečkové páry se hodí jako reprezentace bodů:

```
(defun point (x y)
  (cons x y))

(defun point-x (pt)
  (car pt))

(defun point-y (pt)
  (cdr pt))
```

Když takto definujeme konstruktor a selektory bodů, budou obě naše funkce, které pracují s body (tedy `point-distance` a `segment-center`, ale samozřejmě i libovolné další) fungovat podle očekávání.

V budoucnu se můžeme ale rozhodnout reprezentaci bodů změnit. Pokud body používáme jako abstraktní datovou strukturu, bude stačit vhodně změnit konstruktor a selektory:

```
(defun point (x y)
  (cons y x))

(defun point-x (pt)
  (cdr pt))

(defun point-y (pt)
  (car pt))
```

4 Reprezentace zlomků

Jako další příklad si ukážeme, jak lze tečkové páry použít k reprezentaci zlomků. Zlomek (*fraction*) se skládá z čitatele (*numerator*) a jmenovatele (*denominator*). Ty uložíme do složek *car* a *cdr* tečkového páru. Konstruktor a selektory mohou tedy vypadat takto:

```
(defun fraction (n d)
  (cons n d))

(defun numer (frac)
  (car frac))

(defun denom (frac)
  (cdr frac))
```

Napíšeme několik funkcí na základní operace se zlomky. Všechny používají k práci se zlomky jen konstruktor a selektory (pracují se zlomky jako s abstraktní datovou strukturou):

```
(defun frac-+ (x y)
  (fraction (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(defun frac-* (x y)
  (fraction (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

Porovnávání zlomků: zlomky nemusí být zkrácené, takže nestačí porovnat čitatel a jmenovatel jednoho s čitatelem a jmenovatelem druhého. Můžeme si ale všimnout, že $\frac{a}{b} = \frac{c}{d}$ je totéž jako $ad = cb$.

```
(defun frac-equal-p (x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Vraťme se teď znovu k výhodám datové abstrakce: datová abstrakce

1. zjednodušuje práci (nemusíme se starat o implementační detaily),
2. umožňuje v budoucnu v případě potřeby snadno přejít k jiné reprezentaci,
3. zvyšuje čitelnost kódu.

Následující varianta funkce `frac-*` nepoužívá datovou abstrakci, ale jinak dělá přesně totéž co funkce původní:

```
(defun frac-* (x y)
  (cons (* (car x) (cdr y))
        (* (car y) (cdr x))))
```

Můžeme vidět, že funkce postrádá všechny tři výhody datové abstrakce:

1. Abychom ji mohli napsat, museli jsme vědět, že zlomky jsou reprezentovány tečkovými páry. V původní verzi jsme to vědět nemuseli. (Původní verzi jsme mohli dokonce klidně napsat ještě před tím, než jsme se o konkrétní reprezentaci zlomků rozhodli! Tak jsme to udělali na začátku s funkcemi pracujícími s body.)

2. Pokud bychom se v budoucnu rozhodli reprezentovat zlomky jinak, museli bychom funkci přepsat. V původní verzi ne.

3. Zdrojový kód funkce není dobře čitelný. Není například hned jasné, co znamená `(car x)`. Vidíme sice, že jde o složku *car* páru `x`, ale nevidíme, jaký má význam. V původní verzi uvedené `(numer x)` nám jasně říká, že jde o čitatel zlomku.

Abychom měli zlomky vždy v základním tvaru (zkrácené), upravíme funkci `fraction`, aby před vytvořením zlomku zadaný čitatel a jmenovatel zkrátila:

```
(defun fraction (n d)
  (let ((div (gcd n d)))
    (cons (/ n div) (/ d div))))
```

(Funkce `gcd` počítá největší společný dělitel zadaných dvou čísel. V Lispu je k dispozici jako vestavěná funkce, umíme ji ale taky sami naprogramovat.)

Funkce `frac-equal-p` se pak zjednoduší:

```
(defun frac-equal-p (x y)
  (and (= (numer x) (numer y))
       (= (denom x) (denom y))))
```

5 Páry jako základ složitějších datových struktur

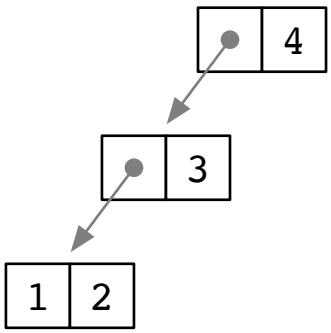
Páry mohou ve svých složkách *car* a *cdr* obsahovat i jiná data než čísla:

```
CL-USER 8 > (cons nil t)
(NIL . T)
```

Mohou dokonce obsahovat i jiné páry:

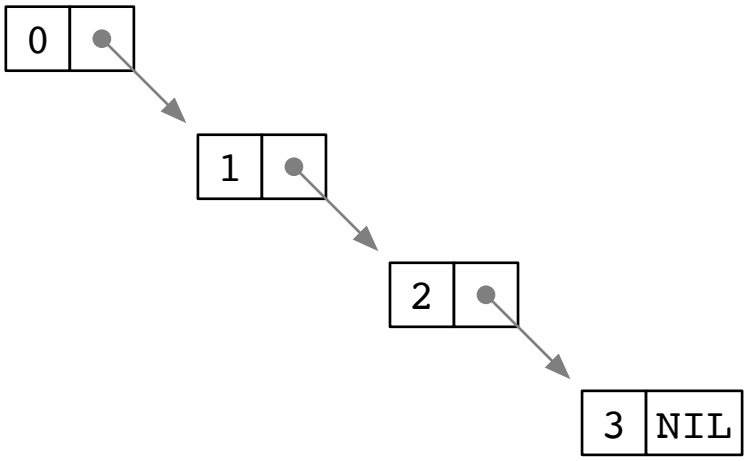
```
CL-USER 9 > (cons (cons 1 2) 3)
((1 . 2) . 3)

CL-USER 10 > (cons (cons (cons 1 2) 3) 4)
(((1 . 2) . 3) . 4)
```

Můžeme si představit, že to, co do složek páru ukládáme, může být jen informace o tom, kde se obsah složky nachází, tedy adresa. Tu si můžeme představit třeba jako šipku. Poslední uvedený pár si tedy můžeme představit takto:



Takovému znázornění struktury párů říkáme *krabičkové znázornění*.

Zásadní roli hrají páry uspořádané následujícím způsobem:



Jsou to páry, v jejichž *cdr* je vždy uložen jiný pár nebo symbol `NIL`. Strukturu z posledního obrázku můžeme vytvořit vyhodnocením výrazu

```
(cons 0 (cons 1 (cons 2 (cons 3 ()))))
```

Takovým strukturám se říká čisté seznamy. Přesně je čistý seznam definován takto:

Čistý seznam, jeho struktura a prvky

Čistý seznam délky n (stručně, ale trochu nepřesně *seznam*, anglicky *proper list*) je pro $n = 0$ symbol `nil` a pro $n > 0$ tečkový pár, jehož *cdr* je čistý seznam délky $n - 1$. Symbolu `nil` se říká také *prázdný seznam*. Je možné ho značit takto: `()`.

Páry tvořící strukturu seznamu jsou páry dosažitelné ze seznamu několikanásobnou (počínaje nulanásobnou) aplikací funkce `cdr`.

Hodnoty uložené v *car*-složkách těchto párů se nazývají *prvky seznamu*.

Seznamy se v Lispu zapisují tak, jak jsme zvyklí. Proto výše uvedený seznam Lisp vytiskne takto:

```
CL-USER 11 > (cons 0 (cons 1 (cons 2 (cons 3 nil))))
(0 1 2 3)
```

Jde ovšem jen o jeden z několika možných zápisů. Další je například `(0 . (1 . (2 . (3 .())))))`.

Podle definice seznamu je *cdr* daného neprázdného seznamu vždy seznam:

```
CL-USER 12 > (cdr (cons 0 (cons 1 (cons 2 (cons 3 nil)))))
(1 2 3)
```

Uvedený seznam lze tedy zapsat například i takto: `(0 . (1 2 3))` nebo takto: `(0 . (1 . (2 3)))`.

Seznam `(0 1 2 3)` je čistý seznam délky 4, jeho prvky jsou `0`, `1`, `2`, `3`.

V Lispu je k dispozici funkce `length`, která zjišťuje délku zadaného čistého seznamu. Mohla by být napsána takto (používám jiný název, aby nedošlo ke konfliktu):

```
(defun my-length (list)
  (if (eql list nil)
      0
    (+ (my-length (cdr list)) 1)))
```

Test:

```
CL-USER 13 > (my-length nil)
0

CL-USER 14 > (my-length (cons 0 (cons 1 (cons 2 (cons 3 ()))))
4
```

Tato funkce vytváří seznam dané délky s daným prvkem (v Lispu je podobná funkce `make-list`):

```
(defun my-make-list (length elem)
  (if (= length 0)
      ()
    (cons elem (my-make-list (- length 1) elem))))
```

Test:

```
CL-USER 17 > (my-make-list 30 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

CL-USER 18 > (my-make-list 30 t)
(T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T)

CL-USER 19 > (my-make-list 10 ())
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)

CL-USER 20 > (my-make-list 5 (my-make-list 3 1))
((1 1 1) (1 1 1) (1 1 1) (1 1 1) (1 1 1))
```

Otázky a úkoly na cvičení

1. Napište predikát `right-triangle-p`, který (podobně jako predikát z druhého cvičení) zjistí, zda zadaný trojúhelník je pravoúhlý. Predikát bude akceptovat jako argumenty vrcholy trojúhelníka jako body.

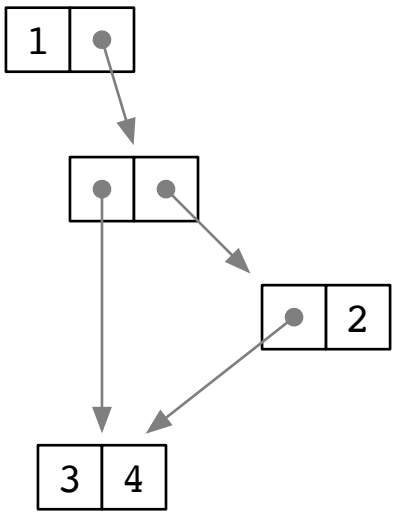
2. Napište funkci `op-vertex`, která k bodům A a B najde bod C tak, že bod B je středem úsečky s vrcholy A a C.

3. Napište funkce na rozdíl a podíl zlomků.

4. Vylepšete funkce pro práci se zlomky tak, aby správně pracovaly i se zápornými hodnotami. (Ani zdaleka není nutné upravovat všechny.)

5. Navrhněte abstraktní datovou strukturu reprezentující uzavřené intervaly reálných čísel. Konstruktor s názvem `interval` bude mít dva argumenty: dolní a horní konec nového intervalu. Selektory se budou jmenovat `lower-bound` a `upper-bound`. Dále napište predikát `number-in-interval-p`, který zjistí, zda je dané číslo prvkem daného intervalu a funkci `interval-intersection`, která vrátí interval, jenž je průnikem zadaných dvou intervalů, nebo `nil` pokud je jejich průnik prázdný.

6. Napište výraz, jehož vyhodnocením vznikne struktura znázorněná krabičkovým znázorněním na tomto obrázku:



7. Napište predikát `proper-list-p`, který zjistí, zda jeho argument je čistý seznam.

8. Napište iterativní verzi funkce `my-make-list`.

9. Napište funkci `make-ar-seq-list`, která vytvoří seznam členů aritmetické posloupnosti se zadaným prvním členem, diferencí a počtem členů:

```
> (make-ar-seq-list 10 2 6)
(10 12 14 16 18 20)
```

10. Napište iterativní verzi funkce `make-ar-seq-list`.

11. Napište funkci `make-geom-seq-list` která vytvoří seznam členů geometrické posloupnosti s daným prvním členem, kvocientem a počtem členů:

```
> (make-geom-seq-list 5 2 10)
(5 10 20 40 80 160 320 640 1280 2560)
```

12. Napište iterativní verzi funkce `make-geom-seq-list`.