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

5. Mutace v datových strukturách

verze z 19. května 2019

1 Mutace v párech a seznamech

Základní datovou strukturou, o které jsme mluvili minulý semestr, je tečkový pár. Známe její konstruktor `cons` a selektory `car` a `cdr`. K mutacím v tečkových párech lze použít makro `setf`:

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

CL-USER 4 > (setf (car pair) 2)
2

CL-USER 5 > pair
(2 . 2)
```

Důležité je si uvědomit, že v proměnné `pair` je stále uložen týž pár, jen je změněn obsah jedné jeho složky. To lze demonstrovat tímto testem:

```
CL-USER 6 > (setf pair2 pair)
(2 . 2)

CL-USER 7 > (setf (cdr pair) 1)
1

CL-USER 8 > pair
(2 . 1)

CL-USER 9 > pair2
(2 . 1)
```

Tím, že jsme (na řádku 7) změnili obsah proměnné `pair`, se změnil i obsah proměnné `pair2`. Je to tím, že obě proměnné obsahují tutéž hodnotu:

```
CL-USER 10 > (eql pair pair2)
T
```

Srovnejte to s tímto pokusem:

```
CL-USER 11 > (setf pair (cons 1 2))
(1 . 2)

CL-USER 12 > (setf pair2 (cons 1 2))
(1 . 2)

CL-USER 13 > (eql pair pair2)
NIL
```

Páry uložené v proměnných `pair` a `pair2` jsou různé, i když se vytisknou stejně. Každý z nich je totiž vytvořen zvláštním voláním konstruktoru `cons`. Mutace jednoho páru tak nemá vliv na druhý:

```
CL-USER 14 > (setf (car pair) 3)
3

CL-USER 15 > pair
(3 . 2)

CL-USER 16 > pair2
(1 . 2)
```

Jak víme, **seznamy** jsou vytvářeny z párů. Konstruktor seznamů (hlavní je funkce `list`) a selektory (například funkce `first`, `second` atd., a třeba `nth`) jsou definovány pomocí funkcí `car` a `cdr`.

K **mutaci** seznamů lze opět použít makro `setf` společně se jmény uvedených funkcí. Například:

```
CL-USER 6 > (setf list (list 1 4 3))
(1 4 3)

CL-USER 7 > (setf (second list) 2)
2

CL-USER 8 > list
(1 2 3)

CL-USER 9 > (setf (cddr (cdr list)) (list 3 5))
```

```
(3 5)

CL-USER 10 > list
(1 2 3 3 5)

CL-USER 11 > (setf (nth (+ 1 1) (cdr list)) 4)
4

CL-USER 12 > list
(1 2 3 4 5)
```

2 Místa

Makro `setf` tedy můžeme používat nejen k aktualizaci hodnoty proměnné, ale obecně dalších hodnot uložených například v datových strukturách. Obecné použití makra je

```
(setf place expression)
```

kde ***place*** je výraz určující jaká hodnota se má aktualizovat a ***expression*** je výraz, jehož hodnota se k aktualizaci použije. Po aktualizaci musí výraz ***place*** vrátit tutéž hodnotu, na jakou se při aplikaci `setf`-výrazu vyhodnotil výraz ***expression***.

Výrazu ***place*** se říká *místo*. S místy lze pracovat obdobně jako s proměnnými: lze na ně ukládat hodnoty a potom je zase číst. (Proměnné jsou základním příkladem míst.) Samozřejmě ne všechny výrazy mohou sloužit jako místa.

V příkladech v předchozí části jsme se (kromě proměnných) setkali s následujícími místy: `(car pair)`, `(cdr pair)`, `(second list)`, `(cddr (cdr list))`, `(nth (+ 1 1) (cdr list))`. U všech těchto míst makro `setf` postupuje tak, že nejprve vyhodnotí jejich argumenty. Tedy například při vyhodnocování výrazu

```
(setf (nth (+ 1 1) (cdr list)) 4)
```

se vyhodnotí výrazy `(+ 1 1)`, `(cdr list)` (a samozřejmě také výraz `4`). Funkce `nth` se ovšem nezavolá. Místo toho se provede aktualizace seznamu.

O symbolech `car`, `cdr`, `second`, `nth` (a samozřejmě mnoha dalších) říkáme, že *určují místo*. Obecně se toto říká o symbolech ***f***, které jsou jednak jménem funkce (nebo i makra) a pro které výraz označující jejich aplikaci (tedy výraz `(`***f arg1 ... argn***`)`, který lze bez chyby vyhodnotit) je místo. Ve standardu se také takovým funkcím říká *accessor*.

Makro `setf` ve své obecné podobě akceptuje libovolný sudý počet argumentů. Slouží to k nastavení více míst současně.

Například

```
(setf x 1 y 2)
```

je totéž jako

```
(progn (setf x 1)
       (setf y 2))
```

Důležitá poznámka

Je zakázáno modifikovat páry a seznamy vzniklé kvotováním ve zdrojovém kódu. Toto je tedy zakázáno:

```
> (setf a '(1 2 3))
(1 2 3)

> (setf (third a) 4)                                    nesprávně
```

Páry a seznamy lze tedy modifikovat, pouze když vznikly za běhu programu aplikací některého konstruktoru, tedy například funkcí `cons` nebo `list`.

Stejné omezení se týká všech tzv. **literálů**, tedy hodnot napsaných přímo ve zdrojovém kódu, tedy například řetězců zapsaných přímo s uvozovkami:

```
"Nemodifikovat!"
```

3 Makra na práci s místy

V Common Lispu existuje kromě makra `setf` několik užitečných maker, která s místy pracují. Pro zajímavost některá z nich uvedu.

Makro `psetf` („paralelní `setf`“)

Makro pracuje podobně jako makro `setf`, ale nastavované hodnoty počítá všechny před tím, než jsou nastaveny. Příklad:

```
CL-USER 3 > (setf x 1 y 2 z 3)
3

CL-USER 4 > (psetf x (+ x y)
                   y (- x y)
```

```
                    z (* x y))
NIL

CL-USER 5 > (list x y z)
(3 -1 2)
```

Makro ovšem pracuje s místy, takže je možné udělat například i toto:

```
CL-USER 11 > (setf c (cons 1 2))
(1 . 2)

CL-USER 12 > (psetf (car c) (cdr c) (cdr c) (car c))
NIL

CL-USER 13 > c
(2 . 1)
```

K podobným účelům je ovšem ještě vhodnější makro `rotatef`.

Makro `rotatef`

```
CL-USER 14 > (setf l (list 1 2 3))
(1 2 3)

CL-USER 15 > (rotatef (first l) (second l) (third l))
NIL

CL-USER 16 > l
(2 3 1)
```

Všimněte si, že makro `rotatef` používá místa jak ke čtení hodnoty, tak k zápisu. To dělá i následující makro `incf`.

Makro `incf`

Toto makro zjistí hodnotu daného místa (musí být číselná), inkrementuje ji a výsledek uloží zpět na dané místo. Také jej vrátí jako svůj výsledek:

```
CL-USER 17 > (setf x 5)
5

CL-USER 18 > (incf x)
6
```

```
CL-USER 19 > x
6
```

Hodnota přírůstku je defaultně 1, ale lze ji změnit nepovinným parametrem:

```
CL-USER 20 > (setf list (list 1 2 3 1))
(1 2 3 1)

CL-USER 21 > (incf (fourth list) 3)
4

CL-USER 22 > list
(1 2 3 4)
```

Makro `decf`

funguje stejně jako makro `incf`, jen s tím rozdílem, že místo inkrementace dekrementuje.

Makra `push` a `pop`

Tato makra pracují se seznamy jako se zásobníky. Makro `push` přidá k seznamu na zadaném místě daný prvek a místo aktualizuje. Makro `pop` vrátí první prvek seznamu a zadané místo aktualizuje na zbytek seznamu:

```
CL-USER 36 > (setf stack (list 1 2 3 4 5))
(1 2 3 4 5)

CL-USER 37 > (push 0 stack)
(0 1 2 3 4 5)

CL-USER 38 > stack
(0 1 2 3 4 5)

CL-USER 39 > (pop stack)
0

CL-USER 40 > stack
(1 2 3 4 5)

CL-USER 41 > (pop stack)
1

CL-USER 42 > stack
(2 3 4 5)
```

Následující pokusy vedou k chybám, protože jsme jako argumenty maker nepoužili místa (chybová hlášení jsou zkrácena):

```
CL-USER 43 > (push 1 (list 2 3))

Error: Undefined operator (SETF LIST) in form ((SETF LIST)
#:|Store-Var-3469| #:G3470 #:G3471).

CL-USER 44 : 1 > :a

CL-USER 45 > (pop (list 1 2 3))

Error: Undefined operator (SETF LIST) in form ((SETF LIST)
#:|Store-Var-3473| #:G3474 #:G3475 #:G3476).
```

4 Definice míst

V Lispu je možné definovat vlastní místa. Lze to udělat několika způsoby. Zde si pro zajímavost ukážeme jeden. Je-li *f* funkce (nebo i makro), můžeme definovat funkci (nebo makro) s názvem `(defun f)`. Ta musí mít stejný λ-seznam jako *f* s jedním povinným parametrem navíc, který je nutné uvést na první místo λ-seznamu. Funkce není určena k použití uživatelem, ale bude implicitně použita k aktualizaci místa. Povinný parametr navíc ponese nastavovanou hodnotu. Tuto hodnotu musí funkce také vracet jako výsledek.

Ukážeme si to na příkladě.

Příklad

Funkce `last-element` vrací poslední prvek daného seznamu. Používá k tomu funkci `last`, která v Lispu už je a vrací poslední pár (nikoli prvek).

```
(defun last-element (list)
  (car (last list)))
```

Test:

```
CL-USER 5 > (last-element (list 1 2 3 4))
4
```

Aby symbol `last-element` určoval místo, definujeme funkci `(setf last-element)`:

```
(defun (setf last-element) (value list)
  (setf (car (last list)) value))
```

Na této funkci je zvláštní, že jejím názvem není symbol, ale seznam. To nás ovšem nemusí trápit, budeme ji používat pouze nepřímo. Funkce má o jeden parametr víc než funkce `last-element`. Je to parametr `value`, který bude při (implicitní) aplikaci funkce navázán na nastavovanou hodnotu. Tu také funkce vrátí jako výsledek (to zařídí použité makro `setf`).

Test:

```
CL-USER 6 > (setf list (list 1 2 2))
(1 2 2)

CL-USER 7 > (setf (last-element list) 3)
3

CL-USER 8 > list
(1 2 3)
```

Poznámka. Makro `decf` v posledním příkladě muselo vyzvednout hodnotu z místa `(last-element list)` a novou hodnotu na ně uložit. Zavolalo tedy nejprve naši funkci `last-element` a pak funkci `(setf last-element)`. To tedy vedlo ke dvojímu zavolání funkce `last`. Jinými slovy, konec seznamu se hledal dvakrát. To by někdy mohlo být příliš neefektivní (tady ne). Náprava by vyžadovala větší znalosti o místech a složitější definici místa daného symbolem `last-element`.

5 Kruhové struktury

Podívejme se na tuto funkci:

```
(defun cl (elem)
  (let ((result (list elem)))
    (setf (cdr result) result)
    result))
```

Funkce vrací seznam, jehož prvním prvkem je zadaný prvek `elem` a zbytkem (*cdr*) je opět tentýž seznam. Takový seznam jsme v prvním semestru nemohli vyrobit, protože bez použití mutace to nejde.

Funkce pracuje správně a má opravdu požadovaný efekt. Přesto pokus o její aplikaci v Listeneru skončí chybou. Bude to proto, že chybou skončí pokus o vytisknutí výsledku, protože povede k nekonečnému cyklu.

Pokud bychom chtěli vytisknout aspoň prvních několik prvků např. seznamu vráceného vyhodnocením výrazu `(cl 1)`, dostali bychom toto:

```
(1 1 1 1 1 1 1 1 1 1 1 ...
```

V Lispu je k dispozici globální proměnná `*print-length*`, s jejíž pomocí lze omezit počet tisknutých prvků daného seznamu. Pokud má hodnotu `nil`, nemá žádný efekt a Lisp se pokouší tisknout seznamy celé (i s rizikem zacyklení). Jinak má mít číselnou hodnotu; ta pak udává maximální počet tisknutých prvků:

```
CL-USER 1 > (setf *print-length* 12)
12

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

Jiná možnost, jak se vyhnout zacyklení při tisku je použít proměnnou `*print-circle*`. To má výhodu, že máme kompletní informaci o struktuře a obsahu seznamu, což je rozdíl proti předchozímu případu, kdy z vytisknutého seznamu s dvanácti jedničkami nevíme, zda na třináctém místě není třeba dvojka.

Proměnná `*print-circle*` má výchozí hodnotu `nil`. Pokud hodnotu změníme na něco jiného (bude jí tedy *Pravda* v rámci zobecněné logiky), Lisp bude dávat při tisku pozor, zda se nesnaží tisknout pár, který už jednou tiskl. Pokud ano, zavede pro něj značku a místo páru vytiskne ji:

```
CL-USER 3 > (setf *print-circle* t)
T

CL-USER 4 > (cl 2)
#1=(2 . #1#)

CL-USER 5 > (list (cl 3) (cl 4))
(#1=(3 . #1#) #2=(4 . #2#))

CL-USER 6 > (cl *)
#3=((#1=(3 . #1#) #2=(4 . #2#)) . #3#)
```

Takto zapsané kruhové struktury se dají zadávat i přímo v programu:

```
CL-USER 12 > '(1 . #1=(2 3 . #1#))
(1 . #1=(2 3 . #1#))
```

Vzhledem k existenci kruhových seznamů musíme vylepšit naše předchozí definice. Použijeme k tomu funkci `nthcdr` (podrobnosti o ní si zjistěte ve standardu).

Seznam je buď symbol `nil`, nebo pár.

Tečkový seznam je seznam *`list`* takový, že existuje číslo *`n`*, pro které `(nthcdr` *`n list`*`)` není seznam.

Kruhový seznam je seznam *`list`* takový, že existují dvě různá čísla *m* a *`n`*, pro která `(nthcdr` *`m list`*`)` ani `(nthcdr` *`n list`*`)` není `nil`, a současně `(eql (nthcdr` *`m list`*`) (nthcdr` *`n list`*`))` je *Pravda*.

Čistý seznam je seznam, který není ani tečkový, ani kruhový.

Čistý seznam můžeme vymezit také tak, že řekneme, že je to nejmenší možná množina hodnot, pro kterou platí:

1. Symbol `nil` v této množině leží.
2. Jestliže *cdr* páru v množině leží, leží v ní i pár.

Obvykle (a je to tak i v jiných předmětech) pokud řekneme *seznam*, myslíme tím pouze čistý seznam. (Například funkce `list` vytváří pouze čisté seznamy, funkce `find` prohledává bez chyby jen čisté seznamy atd.) Pokud bychom pracovali se seznamem, který může být i tečkový nebo kruhový, vždy na to upozorníme.

Ještě jedna důležitá poznámka. Takto by mohla vypadat funkce na spojení dvou seznamů (v Lispu je funkce na spojení libovolného počtu seznamů, jmenuje se, jak víme, `append`):

```
(defun append-2 (list1 list2)
  (if list1
      (cons (car list1)
            (append-2 (cdr list1) list2))
    list2))
```

Mohli bychom si říci, že si ušetříme kopírování prvního seznamu a přímo ho použijeme (funkce v Lispu je `nconc`):

```
(defun append-2-dest (list1 list2)
  (if list1
      (progn (setf (cdr (last list1)) list2)
        list1)
    list2))
```

Zdánlivě funguje vše, jak má:

```
CL-USER 1 > (setf list1 (list 1 2 3)
                  list2 (list 4 5 6))
(4 5 6)

CL-USER 2 > (append-2 list1 list2)
(1 2 3 4 5 6)

CL-USER 3 > (append-2-dest list1 list2)
(1 2 3 4 5 6)
```

Obsah proměnné `list1` se nám ale po posledním vyhodnocení změnil, což je něco, co by nás mohlo nepříjemně překvapit:

```
CL-USER 4 > list1
(1 2 3 4 5 6)
```

Způsobila to funkce `append-2-dest`, která seznam v proměnné modifikovala — funkce je *destruktivní*.

Fatální dopad může mít pokus spojit funkcí `append-2-dest` seznam se sebou samým:

```
CL-USER 5 > (append-2 list2 list2)
(4 5 6 4 5 6)

CL-USER 6 > list2
(4 5 6)

CL-USER 7 > (append-2-dest list2 list2)
#1=(4 5 6 . #1#)

CL-USER 8 > list2
#1=(4 5 6 . #1#)
```

V Common Lispu se o některých funkcích říká, že jsou *destruktivní*. Znamená to, že během volání mohou modifikovat některý svůj argument. Příkladem destruktivní funkce může být naše funkce `append-2-dest`. Pokud si programátor není jistý, co přesně dělá, měl by se takovým funkcím vyhnout.

Otázky a úkoly na cvičení

1. Funkce `test1` a `test2` jsou definovány takto:

```
(defun test1 (a)
  (setf a 2))

(defun test2 (a)
  (test1 a)
  a)
```

Co bude hodnotou výrazu `(test2 1)`?

2. Funkce `test3` a `test4` jsou definovány takto:

```
(defun test3 (a)
  (setf (car a) 2))

(defun test4 (a)
  (test3 a)
  a)
```

Co bude hodnotou výrazu `(test4 (list 1))`?

3. Definujte funkci `circlist`, která pracuje stejně jako funkce `list`, ale vytvoří kruhový seznam podle následujícího testu:

```
CL-USER 1 > (setf *print-circle* t)
T

CL-USER 2 > (setf list (circlist 1 2 3 4))
#1=(1 2 3 4 . #1#)

CL-USER 2 > (sixth list)
2
```

4. Nakreslete jako přihrádky páry, kterými je tvořený kruhový seznam `list` v předchozím příkladě.

5. Napište funkci `circularp`, která pro libovolnou strukturu tvořenou páry a jinými hodnotami zjistí, zda je kruhová. Postupujte tak, že budete strukturu procházet a do pomocného seznamu si ukládat všechny páry, na které narazíte. Jakmile narazíte na pár, který v seznamu už je, výsledek je pozitivní. (Hledat v seznamu můžete pomocí funkce `find`.)