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 16. března 2020

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 stejné jako u proměnných, na jejichž místě ovšem může být obecný výraz:

```
(setf place1 expr1 ... placen exprn)
```

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

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 jednodušších míst (což jsou všechna, která jsme zatím uvedli) při nastavování hodnoty makro `setf` nejprve **vyhodnotí argumenty místa**.

Tedy například při vyhodnocování výrazu

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

se vyhodnotí výrazy `(+ 1 1)`, `(cdr list)`. Funkce `nth` se ovšem nezavolá. Místo toho se provede aktualizace seznamu: prvek s indexem `2` jeho *cdr* se nastaví na `4` (hodnotu `4` samozřejmě získáme vyhodnocením drzhého argumentu `setf`-výrazu).

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*.

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!"
```

(Jak lze nastavovat prvky polí si řekneme časem.)

Dále není povoleno modifikovat seznamy uložené v `&rest`-parametrech funkcí.

3 Makra na práci s místy

Jak už víme z minula, v Common Lispu existují kromě makra `setf` další makra, která jsou z něj odvozená. Teď můžeme i upřesnit, že jde o makra, která pracují s místy. Proto je můžeme použít i takto:

```
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)

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)

CL-USER 17 > (setf list (list 1 2 3 1))
(1 2 3 1)

CL-USER 18 > (incf (fourth list) 3)
4

CL-USER 19 > list
(1 2 3 4)
```

Makra `push` a `pop`

S těmito makry jsme se ještě nesetkali. Jsou určena k práci se seznamy jako se zásobníky. Na místě, které je argumentem makra, musí být uložen seznam (tj. vyhodnocením místa dostaneme seznam). 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).
```

V jedné z úloh z minulé přednášky jste měli napsat vlastní verzi makra `incf`. Jedno z možných řešení je toto:

```
(defmacro my-incf (sym &optional (inc 1))
  `(setf ,sym (+ ,sym ,inc)))
```

Makro funguje dobře, když ho používáme jen s proměnnými, jak jsme to dělali na minulé přednášce. Při použití s obecnými místy zjistíme, že ale vytváří problém vícenásobného vyhodnocení:

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

CL-USER 2 > (incf (second (print list)))

(1 2 3)
3

CL-USER 3 > list
(1 3 3)

CL-USER 4 > (my-incf (second (print list)))

(1 3 3)
(1 3 3)
4

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

(Na řádku 4 byl výraz `(print list)` vyhodnocen dvakrát.) Řešením tohoto problému se nebudeme zabývat.

4 Definice míst

V Lispu je možné definovat vlastní místa. Lze to udělat několika způsoby. Zde si pro zajímavost ukážeme jeden jednoduchý, který použijeme i v příštím semestru. 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) 4)
4

CL-USER 8 > list
```

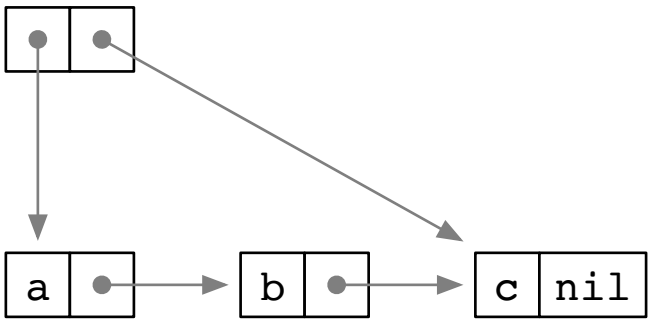
```
(1 2 4)

CL-USER 9 > (decf (last-element list))
3

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

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`.

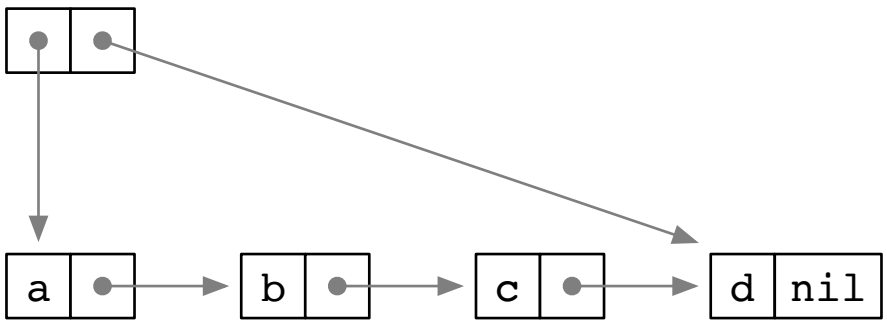
5 Příklad: fronta

Seznam můžeme použít k implementaci fronty, tj. struktury typu FIFO, které lze na konec přidávat nové hodnoty a ze začátku odebírat. Abychom s frontou mohli pracovat jako s hodnotou, použijeme navíc jeden pár, který bude frontu reprezentovat a který bude ve svých složkách obsahovat její začátek a konec.

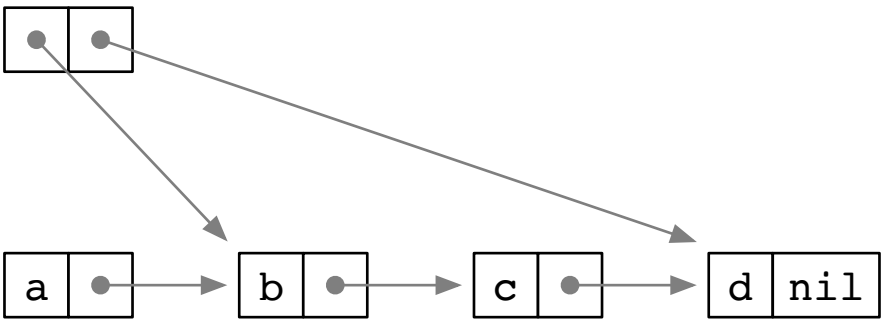
Takto vypadá fronta s hodnotami `a`, `b`, `c`:



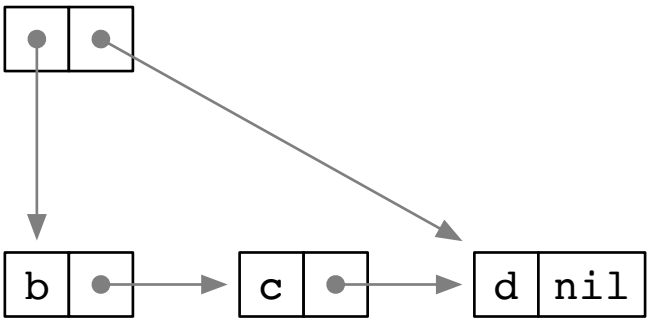
Po přidání symbolu `d` na konec:



Po vypuštění symbolu `a` ze začátku:



Pár obsahující tento symbol už není použit, proto se o něj nemusíme starat (bude vymazán garbage collectorem):



Při všech těchto operacích zůstává pár, který frontu reprezentuje (je zakreslený vždy nahoře), stejný, jen se mění obsah jeho složek *car* a *cdr*.

Zvláštním typem fronty je *prázdná fronta*. Ta neobsahuje žádný prvek, složka *car* páru, který ji reprezentuje je `nil`. (Složka *cdr* pak nemá význam.)

Konstruktor fronty vytvoří novou prázdnou frontu:

```
(defun make-queue ()
  (cons nil nil))
```

Pomocné funkce (neprozradíme uživateli):

```
(defun queue-front (q)
  (car q))

(defun (setf queue-front) (front q)
  (setf (car q) front))

(defun queue-rear (q)
  (cdr q))

(defun (setf queue-rear) (rear q)
  (setf (cdr q) rear))
```

Test prázdnosti fronty:

```
(defun empty-queue-p (q)
  (null (queue-front q)))
```

Mutátory na odebrání prvku zepředu a přidání prvku dozadu (oba vrací prvek):

```
(defun dequeue (q)
  (when (empty-queue-p q)
    (error "Queue is empty"))
  (prog1 (car (queue-front q))
    (setf (queue-front q) (cdr (queue-front q)))))

(defun enqueue (q el)
  (let ((new (cons el nil)))
    (if (empty-queue-p q)
        (setf (queue-front q) new)
      (setf (cdr (queue-rear q)) new))
    (setf (queue-rear q) new)
    el))
```

6 Nebezpečí destruktivních funkcí

Takto by mohla vypadat funkce na spojení dvou seznamů (obecnější funkci `append` známe z Lispu):

```
(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 aplikace 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.

7 Kruhové seznamy

Co dostaneme vyhodnocením těchto výrazů?

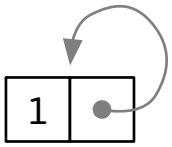
```
> (setf result (list 1))
> (setf (cdr result) result)
```

Na vytvoření datové struktury, kterou vrátí druhý výraz, můžeme napsat i 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.

Strukturu vrácenou vyhodnocením druhého ze shora uvedených výrazů vytvoříme výrazem `(cl 1)`. Krabičkovým znázorněním ji můžeme nakreslit takto:



Funkce pracuje správně a má opravdu požadovaný efekt. Přesto pokus o její aplikaci v Listeneru (a samozřejmě už pokus o manuální vyhodnocení dvou výrazů na začátku) 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 seznamu vráceného vyhodnocením výrazu `(cl 1)`, dostali bychom toto:

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

Jak víme, v Lispu je k dispozici speciální (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 ...)
```

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 7 > '(1 . #1=(2 3 . #1#))
(1 . #1=(2 3 . #1#))
```

(Za apostrofem je napsán výraz, který reader přečte jako kruhový seznam; díky apostrofu se vrátí jako výsledek.)

Vzhledem k existenci kruhových seznamů musíme vylepšit naše předchozí definice. Při té příležitosti a pro úplnost také zavedeme pojem tečkového seznamu. V definici použijeme funkci `nthcdr`.

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*, tj. není `nil`.

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

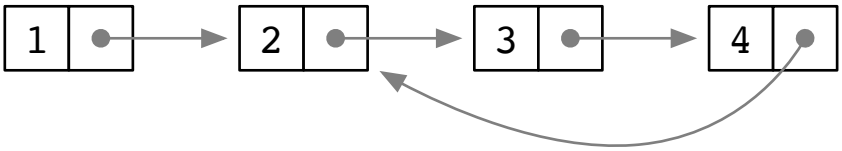
Definice čistého seznamu je tady tedy jiná než definice z prvního semestru, kde jsme definovali *čistý seznam délky n* (je s ní ale ekvivalentní).

Čistý seznam můžeme vymezit také tak, že řekneme, že množina všech čistých seznamů je 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. Pokud bychom pracovali se seznamem, který může být i tečkový nebo kruhový, vždy na to upozorníme.

U kruhového seznamu nemusí být ani jedno z čísel m a n z definice rovno nule. Toto je příklad takového kruhového seznamu:



Mohl by vzniknout například vyhodnocením výrazu

```
(let ((circle (list 1 2 3 4)))
  (setf (cdddr circle) (cdr circle))
  circle)
```

Na co se tento kruhový seznam vytiskne (při hodnotě proměnné `*print-circle*` různé od `nil`)?

Mnoho funkcí Lispu, které pracují se seznamy, předpokládá, že jde o čisté seznamy. Jsou to například funkce `find`, `append` (ta to předpokládá o všech svých argumentech kromě posledního), `length` atd. Pokud si je zkusíte sami naprogramovat, pochopíte, že v případě tečkových seznamů může jejich použití vést k chybě a v případě kruhových k zacyklení.

V Lispu je také funkce `list-length`, která vrací délku seznamu, ale umí se vypořádat i s možností, že je seznam kruhový (a pak místo délky vrací `nil`). Jak to asi dělá? Jeden ze způsobů, jak zjistit, zda je daný seznam kruhový, teď ukážu.

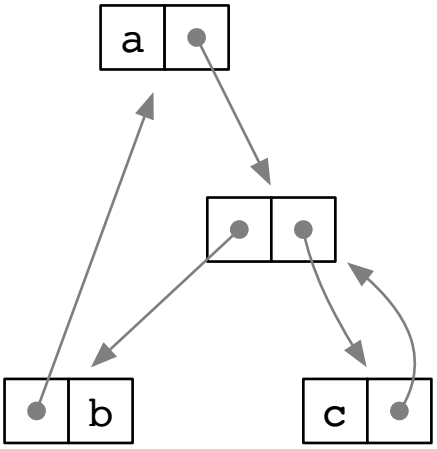
Algoritmus využívá techniku **pomalého a rychlého běžce**. Každý z běžců startuje na vstupním seznamu (tj. na jeho prvním páru). Pomalý běžec se pak v každém kroku přesouvá na *cdr* páru, na kterém zrovna je, rychlý běžec na *cdr* z *cdr* páru, na kterém je. Rychlý běžec tedy běží dvakrát rychleji a na konec seznamu dorazí jako první. Pokud je seznam ovšem kruhový, dřív nebo později seznam oběhne a pomalého běžce dožene zezadu.

Princip si můžete vyzkoušet na kruhovém seznamu nakresleném na předchozím obrázku. Takto by mohl vypadat predikát, který jej implementuje (predikát funguje na čisté a kruhové seznamy, u tečkových vede k chybě):

```
(defun circlep (list)
  (labels ((cp (slow fast)
             (cond ((null fast) nil)
                   ((eql slow fast) t)
                   (t (cp (cdr slow) (cddr fast))))))
    (cp (cdr list) (cddr list))))
```

8 Obecné kruhové struktury

Struktura tečkových párů (jak jsme o nich mluvili minulý semestr) může být také kruhová v tom smyslu, že po několikanásobné aplikaci funkcí `car` a `cdr` na pár, který v ní leží, dostaneme tentýž pár. Toto je například kruhová struktura:



Chceme-li u takto obecné struktury zjistit, zda je kruhová, musíme ji procházet a někde si pamatovat již navštívené páry. Tak to dělá následující predikát:

```
(defun circular-structure-p (s)
  (let ((visited '()))
    (labels ((cs (node)
               (cond ((find node visited) t)
                     ((consp node)
                      (push node visited)
                      (or (cs (car node)) (cs (cdr node))))
                     (t nil))))
      (cs s))))
```

Otázky a úlohy 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. Jak jste se učili v jiném předmětu, struktura (čistého) seznamu je výhodná z hlediska přidávání a odebírání prvků. Vyzkoušejte si to napsáním funkcí `add-val` a `delete-cons`, které destruktivně upraví seznam přidáním páru na dané místo a odebráním daného páru.

   Funkce `add-val` přijímá jako argumenty hodnotu, seznam a pár. Daný seznam modifikuje tak, že do něj přidá pár s touto hodnotou a do seznamu jej umístí před pár, který je třetím argumentem funkce. Přidání na konec seznamu se realizuje uvedením symbolu `nil` místo tohoto páru. Funkce upravený seznam vrátí:

```
> (setf list (list 1 2 4))
(1 2 4)

> (add-val 3 list (cddr list))
(1 2 3 4)

> (eql * list)
T
```

   Funkce `delete-cons` přijímá jako argument pár a seznam. Ze seznamu pár odebere a seznam vrátí jako výsledek (s proměnnou `list` vzniklou předchozím testem):

```
> (delete-cons (cddr list) list)
(1 2 4)

> (eql * list)
T
```

4. Na konci 6. přednášky minulého semestru jste viděli příklad třídění seznamu algoritmem *merge sort*. Upravte jej tak, aby nevytvářel ani jeden nový pár, ale aby místo toho destruktivně modifikoval původní seznam. Budete muset napsat svou destruktivní verzi lispové funkce `ldiff` a upravit funkci `merge-lists`.

5. Nastavte proměnnou `list` vyhodnocením výrazu `(setf list (list 3 2 1))` a pak vyhodnoťte `(merge-sort list)` se svou novou verzí funkce `merge-sort`. Co pak bude v proměnné `list`? Odhadněte a pak vyzkoušejte. Porovnejte s verzí funkce z minulého semestru.

6. 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
```

   Pozor, seznam uložený v `&rest`-parametru funkce není povoleno modifikovat. Vytvořte si nejdřív jeho kopii (např. funkcí `copy-list`).

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

8. Pomocí notace `#1= ...` atd. zapište strukturu z obrázku na začátku části 8 jako výraz.

9. Napište výraz, který při vyhodnocení zkonstruuje (tj. vytvoří zcela novou) strukturu z obrázku na začátku části 8.

10. Řekli jsme, že množina všech čistých seznamů je nejmenší možná množina hodnot, pro kterou platí:

    (a) Symbol `nil` v této množině leží.

    (b) Jestliže *cdr* páru v množině leží, leží v ní i pár.

Vysvětlete, proč kruhové ani tečkové seznamy v této množině neleží a proč naopak každý čistý seznam v ní leží.

11. Napište funkci `circ-find`, která bude pracovat stejně jako funkce `find`, ale poradí si i s kruhovými seznamy. Nepovinné parametry funkce `find` nemusíte programovat. O některé z nich (např. `:test` a `:key`) se můžete pokusit potom.

12. Napište funkci `struct-find`, která dělá totéž pro obecnou kruhovou strukturu, jak jsme o nich mluvili v části 8. Pokud například v proměnné `struc` máme strukturu z obrázku, tak

```
> (struct-find 'c struc)
C

> (struct-find 'd struc)
NIL
```

13. *Obousměrný seznam* se skládá z buněk, které na rozdíl od klasického seznamu obsahují (odkaz na) následující i předchozí buňku. Každá buňka, je tedy datovou strukturou obsahující tři hodnoty, a to hodnotu buňky (selektor `val`), předchozí prvek (`prev`) a následující prvek (`next`). V případě, že buňka nemá předchozí nebo následující buňku, je příslušná hodnota `nil`. Napište konstruktor pro takovou buňku s názvem `bidir-cons` a dále uvedené selektory. Definujte je jako místa, aby bylo možné každé buňce hodnoty i upravovat.

14. Obousměrný seznam budeme reprezentovat párem, který v *car* obsahuje první a v *cdr* poslední buňku seznamu. V případě prázdného obousměrného seznamu obsahuje v *car* symbol `nil`. (*cdr* pak nemá význam, stejně jako v našem příkladu fronty.)

    Napište funkci `list-to-bidir-list`, která k zadanému seznamu vytvoří obousměrný seznam se stejnými prvky ve stejném pořadí. Pak napište funkci `bidir-list`, která vytvoří nový obousměrný seznam podobně, jako funkce `list` vytváří nový klasický seznam.