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

4. Mutace proměnných

verze z 19. května 2019

V této přednášce vybočíme z čistě funkcionálního programování a ukážeme si, jak lze v Lispu nastavovat hodnoty proměnných a složek datových struktur.

1 Změna hodnoty vazby symbolu makrem `setf`

Víme, že symboly v Lispu mají **vazby**. Ty se vytvářejí především pomocí speciálního operátoru `let` a při aplikaci funkce. Při vytvoření vazby je vždy rovnou nastavena její hodnota, která se při používání čistě funkcionálního stylu později už nedá měnit. (Vůči tomuto pravidlu jsme už učinili jednu nebo dvě výjimky, jinak jsme ho ale dodržovali.)

Pokud chceme změnit hodnotu aktuální vazby symbolu, můžeme k tomu použít makro `setf`:

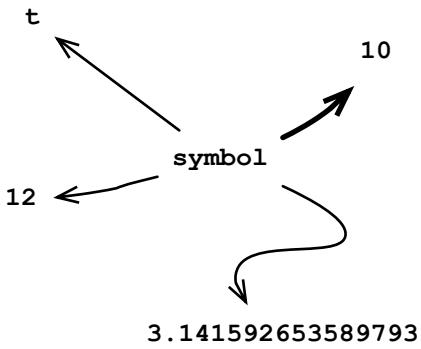
```
CL-USER 1 > (let ((x 1))
              (setf x 2)
              x)
2
```

Výraz

```
(setf symbol expression)
```

1. vyhodnotí výraz ***`expression`*** v aktuálním prostředí,
2. hodnotu aktuální vazby symbolu ***`symbol`*** změní na výsledek vyhodnocení.
3. Výsledek také vrátí jako svou hodnotu.

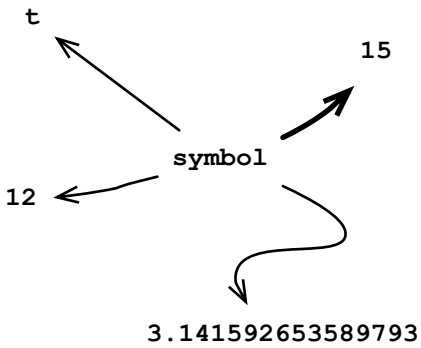
V minulém semestru jsme na jedné přednášce viděli takovýto obrázek:



Šipky znázorňují vazby symbolu `symbol`, tučná šipka je vazba aktuální. Pokud bychom v prostředí s touto vazbou vyhodnotili výraz

```
(setf symbol (+ symbol 5))
```

dopadl by výsledek takto:



Je to proto, že v aktuálním prostředí má symbol nejprve hodnotu `10`, takže výraz `(+ symbol 5)` se vyhodnotí na `15`. Poté se aktuální hodnota symbolu změní. Jeho vyhodnocením (stále ve stejném prostředí) bychom tedy dostali číslo `15`.

Z toho by měl být jasný základní rozdíl mezi makrem `setf` a (například) speciálním operátorem `let`. Zatímco speciální operátor `let` **vytváří novou vazbu symbolu**, makro `setf` **mění hodnotu aktuální vazby symbolu**.

Poznámky k makru `setf`

- Už jsme se setkali (a dále ještě setkáme) s jiným použitím makra `setf` než na nastavení hodnoty proměnné. Například výraz `(setf (car a) 1)` nastaví hodnotu páru uloženého v proměnné `a` na `1`.
- Kromě experimentování v Listeneru bychom nikdy neměli nastavovat hodnotu proměnné, která nemá vazbu (mimo Listener vede pokus o kompilaci takového výrazu k warningu). Makro `setf` ovšem v takové situaci vazbu vytvoří.

Demonstrace ke druhé poznámce. Pokud například napíšeme

```
(defun setf-test ()
  (setf a 1))
```

a pak se definici pokusíme zkompilovat, dostaneme warning

```
;;;*** Warning in SETF-TEST: A assumed special in SETQ
```

2 Použití v lexikálních uzávěrech

Víme, že při vyhodnocování `let`-výrazu i při aplikaci funkce vzniká **pokaždé nové prostředí**. Následky tohoto faktu se plně projeví při modifikaci proměnných. Uvažme tento příklad:

```
(defun two-functions ()
  (let ((x 0))
    (list (lambda ()
            x)
          (lambda (y)
            (setf x y)))))
```

Funkce `two-functions` vrací seznam o dvou prvcích. Každý z nich je nově vytvořená funkce.

První z těchto funkcí vrací hodnotu proměnné `x` v jistém prostředí, druhá tuto hodnotu nastavuje. Funkci můžeme vyzkoušet:

```
CL-USER 3 > (setf list1 (two-functions))
(#<Closure 1 subfunction of TWO-FUNCTIONS Closed 200FDC4A>
#<Closure 2 subfunction of TWO-FUNCTIONS 200FDC32>)

CL-USER 4 > (funcall (first list1))
0

CL-USER 5 > (funcall (second list1) 1)
1

CL-USER 6 > (funcall (first list1))
1
```

Při aplikaci funkce `two-functions` vznikne nové prostředí s vazbou symbolu `x` na hodnotu `0`:

symbol	hodnota
x	0

V tomto prostředí pak funkce vyhodnotí dva λ-výrazy. Jejich výsledkem budou tedy funkce (lexikální uzávěry), které si budou pamatovat prostředí svého vzniku, tedy prostředí, v němž je symbol `x` navázán na nulu. Funkce vrátí ve dvouprvkovém seznamu.

Při zavolání první funkce v seznamu, tedy např. při vyhodnocování výrazu

```
(funcall (first list1))
```

vznikne nové prostředí, jehož předkem je prostředí na předchozím obrázku. Toto prostředí samo neobsahuje žádnou vazbu, protože funkce `(first list1)` nemá žádný parametr:

symbol	hodnota
x	0

↑

symbol	hodnota

V něm se zjistí hodnota symbolu `x` a vrátí jako výsledek.

Při vyhodnocování výrazu

```
(funcall (second list1) 1)
```

se zavolá druhá funkce seznamu. Ta si také pamatuje prostředí svého vzniku, tedy původní prostředí. Její tělo tedy bude vyhodnocováno v tomto prostředí:

symbol	hodnota
x	0

↑

symbol	hodnota
y	1

V těle funkce je výraz `(setf x y)`. Ten nejprve vyhodnotí symbol `y` (samozřejmě v aktuálním prostředí, tedy v prostředí na obrázku) a dostane hodnotu `1`. Potom najde aktuální vazbu symbolu `x` a její hodnotu změní. Po této operaci tedy bude aktuální prostředí vypadat takto:

symbol	hodnota
x	1

symbol	hodnota
y	1

Proto další volání první funkce, tedy vyhodnocení výrazu

```
(funcall (first list1))
```

bude mít jako výsledek číslo `1`.

Všimněme si důležité skutečnosti. Při aplikaci funkce `two-functions` vznikne nové prostředí (je to prostředí s vazbou symbolu `x`), které **nepřestane existovat** ani poté, co aplikace funkce proběhla.

Druhá důležitá skutečnost: Pokud je funkce `two-functions` aplikována podruhé, vytvoří **další prostředí**, nezávislé na prostředí předchozím.

Po dvojí aplikaci funkce tedy budeme mít dvě různá prostředí:

symbol	hodnota
x	0

symbol	hodnota
x	0

Proto můžeme hodnotu vazby symbolu `x` v jednom z nich změnit, aniž by to ovlivnilo druhé prostředí; třeba takto:

symbol	hodnota
x	10

symbol	hodnota
x	20

Toho dosáhneme následujícím vyhodnocením:

```
CL-USER 25 > (setf list1 (two-functions))
(#<Closure 1 subfunction of TWO-FUNCTIONS 20094B52> #<Closure 2
subfunction of TWO-FUNCTIONS Closed 20094B3A>)

CL-USER 26 > (funcall (second list1) 10)
10

CL-USER 27 >
(setf list2 (two-functions))
(#<Closure 1 subfunction of TWO-FUNCTIONS 200AE322> #<Closure 2
subfunction of TWO-FUNCTIONS Closed 200AE30A>)

CL-USER 28 > (funcall (second list2) 20)
20
```

Jak si můžeme potom ověřit:

```
CL-USER 29 > (funcall (first list1))
10

CL-USER 30 > (funcall (first list2))
20
```

3 Počet aplikací funkce

Jako příklad použití mutace u lexikálních uzávěrů si ukážeme funkci, která umí počítat počet svých aplikací.

Bude to funkce `fact` na výpočet faktoriálu, ke které ovšem bude přidružena funkce `fact-cc` (jako *call count*), která bude vracet počet dosud provedených aplikací funkce `fact`:

```
CL-USER 32 > (fact-cc)
0

CL-USER 33 > (fact 5)
120

CL-USER 34 > (fact-cc)
6

CL-USER 35 > (fact 10)
3628800

CL-USER 36 > (fact-cc)
17
```

Je jasné, že počet aplikací funkce je nutné si někde pamatovat, nejspíš v nějaké proměnné. Při naší implementaci se vyhneme tomu, aby proměnná byla globální (globální proměnné jsou nebezpečné a je dobré je nepoužívat, pokud to není nezbytně nutné). Místo toho vytvoříme nové prostředí, ve kterém obě funkce (tj. funkce `fact` a `fact-cc`) vzniknou. V tomto prostředí bude definována proměnná nesoucí počet volání.

Využijeme toho, že i **funkce vytvořené makrem `defun` si pamatují prostředí svého vzniku**. (Makro `defun` totiž ve skutečnosti expanduje na `lambda`-výraz.)

Definice bude vypadat takto:

```
(let ((count 0))

  (defun fact (n)
    (setf count (+ count 1))
    (if (= n 0)
        1
      (* n (fact (- n 1)))))

  (defun fact-cc ()
    count)

)
```

(Tady jsem kvůli přehlednosti vyjímečně porušil konvence psaní zdrojového textu a napsal jednu závorklu na zvláštní řádek.)

Zajímavé je, že proměnná `count` je pro uživatele nepřístupná. Není žádná možnost, jak změnit její hodnotu. (Můžeme jedině proměnnou a obě funkce znovu definovat opětovným vyhodnocením definice.)

4 Reprezentace datových struktur s mutátory

Z minulého semestru víme, že k práci s abstraktními datovými strukturami potřebujeme konstruktor a selektory. Pokud chceme, aby byly datové struktury i mutovatelné (tj. aby bylo možné měnit jejich položky), potřebujeme definovat i *mutátory*. Tohle všechno lze udělat pomocí lexikálních uzávěrů způsobem ukázaným v předchozí části.

Ukážeme si, jak takto reprezentovat páry s mutovatelnými složkami *car* a *cdr*.

Pár budeme reprezentovat jako funkci (uzávěr), jejíž prostředí bude obsahovat dvě vazby s informací o složkách *car* a *cdr*. Funkci samotnou bude možné volat s argumentem, který bude určovat, jakou operaci s párem chceme provést. Bude mít i druhý, nepovinný parametr, který bude případně obsahovat požadovanou novou hodnotu jedné ze složek páru.

Funkce by měla pracovat takto:

```
CL-USER 52 > (setf mc (my-cons 1 2))
#<Closure 1 subfunction of MY-CONS 2009484A>

CL-USER 53 > (funcall mc 'car)
1

CL-USER 54 > (funcall mc 'cdr)
2
```

```
CL-USER 55 > (funcall mc 'set-cdr 3)
3

CL-USER 56 > (funcall mc 'cdr)
3
```

Funkci bychom tedy mohli definovat takto:

```
(defun my-cons (x y)
  (lambda (what &optional val)
    (cond ((eql what 'car) x)
          ((eql what 'cdr) y)
          ((eql what 'set-car) (setf x val))
          ((eql what 'set-cdr) (setf y val)))))
```

Druhá, jednodušší možnost používá makro `case`. Informace o něm si můžete zjistit v dokumentaci:

```
(defun my-cons (x y)
  (lambda (what &optional val)
    (case what
      (car x)
      (cdr y)
      (set-car (setf x val))
      (set-cdr (setf y val)))))
```

Nyní ještě funkce `my-car`, `my-cdr`, `my-set-car` a `my-set-cdr`, které zjednodušují práci s našimi páry:

```
(defun my-car (c)
  (funcall c 'car))

(defun my-cdr (c)
  (funcall c 'cdr))

(defun my-set-car (c val)
  (funcall c 'set-car val))

(defun my-set-cdr (c val)
  (funcall c 'set-cdr val))
```

Test:

```
CL-USER 65 > (setf c (my-cons 1 2))
#<Closure 1 subfunction of MY-CONS Closed 21B404DA>

CL-USER 66 > (my-car c)
1

CL-USER 67 > (my-cdr c)
2

CL-USER 68 > (my-set-cdr c 3)
3

CL-USER 69 > (my-cdr c)
3
```

5 Memoizace

Další hezkou ukázkou použití mutace hodnot vazeb je *memoizace*. Hodí se na funkce, které s danými argumenty vracejí vždy stejné výsledky (jsou tedy **čistě funkcionální**). U takových funkcí je někdy učitečné si již vypočítané výsledky zapamatovat (také se hovoří o *kešování*).

Podívejme se na funkci na výpočet n-tého členu Fibonacciho posloupnosti:

```
(defun fib-1 (n)
  (if (<= n 1)
      1
    (+ (fib-1 (- n 1)) (fib-1 (- n 2)))))
```

Výhodou funkce je její čitelnost, nevýhodou vysoká neefektivita: rychle zjistíme, že kolem hodnoty argumentu 40 se čas výpočtu neúnosně prodlužuje. Někde kolem hodnoty 50 už výpočet trvá nepřijatelně dlouho a výraz `(fib-2 100)` například počítač, na kterém tento text píšu, nevyhodnotí ani za 100000 let.

Zhruba lze odhadnout, že funkce má exponenciální složitost, protože kromě hraničních případů na výpočet výsledku potřebuje sama sebe zavolat dvakrát.

To by chtělo samozřejmě zdůvodnit podrobněji (což je náplň jiného předmětu), ale můžeme také zkusit test pomocí makra `time` (neuvádím všechno, co makro tiskne, aby nebyl výpis příliš dlouhý):

```
CL-USER 84 > (time (fib-1 31))
Timing the evaluation of (FIB-1 31)
Elapsed time = 0.021
```

```
2178309

CL-USER 85 > (time (fib-1 32))
Timing the evaluation of (FIB-1 32)
Elapsed time = 0.031
3524578

CL-USER 86 > (time (fib-1 33))
Timing the evaluation of (FIB-1 33)
Elapsed time = 0.044
5702887

CL-USER 87 > (time (fib-1 34))
Timing the evaluation of (FIB-1 34)
Elapsed time = 0.064
9227465

CL-USER 88 > (time (fib-1 35))
Timing the evaluation of (FIB-1 35)
Elapsed time = 0.098
14930352

CL-USER 89 > (time (fib-1 36))
Timing the evaluation of (FIB-1 36)
Elapsed time = 0.154
24157817

CL-USER 90 > (time (fib-1 37))
Timing the evaluation of (FIB-1 37)
Elapsed time = 0.242
39088169

CL-USER 91 > (time (fib-1 38))
Timing the evaluation of (FIB-1 38)
Elapsed time = 0.383
63245986

CL-USER 92 > (time (fib-1 39))
Timing the evaluation of (FIB-1 39)
Elapsed time = 0.616
102334155

CL-USER 93 > (time (fib-1 40))
Timing the evaluation of (FIB-1 40)
Elapsed time = 0.993
165580141
```

```
CL-USER 95 > (time (fib-1 41))
Timing the evaluation of (FIB-1 41)
Elapsed time = 1.601
267914296

CL-USER 96 > (time (fib-1 42))
Timing the evaluation of (FIB-1 42)
Elapsed time = 2.594
433494437
```

Vidíme, že časy se s rostoucím argumentem rychle prodlužují. Výpočtem můžete zjistit, že i když budou absolutní časy na vašem počítači jiné, budou vždy zhruba představovat geometrickou posloupnost s koeficientem přibližně 1.618. Časová složitost výpočtu bude tedy opravdu zřejmě exponenciální.

Pro zajímavost: přesná hodnota koeficientu je tzv. *zlatý řez*: $\frac{1+\sqrt{5}}{2}$. Je to poměr známý už mnoho století a používaný v umění. Matematicky jde o poměr rozdělení úsečky na dvě tak, že poměr delší a kratší části je stejný jako poměr celé úsečky a delší části.

Memoizovaná verze funkce bude mnohem rychlejší (na přednášce jsem použil hashovací tabulku, tady raději pro zjednodušení používám seznam):

```
(let ((mem '()))

  (defun fib-2 (n)
    (let ((pair (assoc n mem)))
      (unless pair
        (setf pair (cons n
                         (if (<= n 1)
                             1
                           (+ (fib-2 (- n 1)) (fib-2 (- n 2))))))
        (setf mem (cons pair mem)))
      (cdr pair)))

)
```

Funkce je vytvořena v novém lexikálním prostředí s vazbou symbolu `mem` na prázdný seznam. Seznam bude postupně doplňován o páry `(`n `.` fib(n)`)`, kde n je index a fib(n) je n-tý člen Fibonacciho posloupnosti. Seznam tedy slouží k zapamatování už jednou vypočítaných hodnot.

Funkce `fib-2` se nejprve do seznamu podívá (pomocí funkce `assoc`, kterou už známe), zda hodnotu fib(n) už nemá zapamatovanou. Pokud ano, bude v proměnné `pair` příslušný pár. Jinak bude proměnná obsahovat `nil`. Ve druhém případě funkce hodnotu fib(n) vypočítá, pár uloží do proměnné `pair` a také ho přidá k seznamu `mem`.

Nyní je jisté, že v proměnné `pair` je je pár $(n \ . \ \text{fib}(n))$. Stačí tedy vrátit jeho *cdr*.

Všimněte si, že funkce používá mutaci i k nastavení správné hodnoty proměnné `pair`. To je poměrně obvyklý trik. Jinak je mutace samozřejmě použita k úpravě proměnné `mem`.

Nyní si samozřejmě můžete vyzkoušet, jak je na tom funkce `fib-2` s efektivitou. Zjistíte, že hodnotu s argumentem `100` vypočítá jako nic.

6 Vylepšení pomocí maker

Tato část není povinná. Ukazuje zájemcům, jak dotáhnout řešení některých uvedených problémů do elegantnější podoby pomocí maker.

Makro `defcfun`

Pokud bychom chtěli podobně jako funkce `fact` a `fact-cc` definovat další funkce, mohli bychom použít stejnou metodu. To by ovšem vedlo k **nevhodnému opakování kódu**. Ukážu, jak se problému lze zbavit pomocí makra.

Idea je definovat makro `defcfun`, které by pracovalo stejně jako makro `defun`, ale kromě funkce se zadaným názvem ***name*** by definovalo ještě funkci s názvem ***name*`-cc`**, která by vracela počet aplikací funkce ***name***.

Makro by mohlo vypadat takto:

```
(defmacro defcfun (name lambda-list &body body)
  (let ((count (gensym "COUNT")))
    `(let ((,count 0))

       (defun ,name ,lambda-list
         (setf ,count (+ ,count 1))
         ,@body)

       (defun ,(cc-name name) ()
         ,count))

    ))
```

Pokud jste pochopili předchozí definici funkcí `fact` a `fact-cc` a pokud trochu rozumíte makrům, měla by vám tato definice být jasná. Jediný problém je s dosud nedefinovanou funkcí `cc-name`. Ta by měla k danému názvu funkce (což je symbol) vrátit symbol s přidanou příponou „`-cc`“:

```
CL-USER 43 > (cc-name 'fact)
FACT-CC
```

Jak ji definovat si ukážeme v poslední části.

Makro `defmemfun`

Teď ještě stručně ukážu obecné řešení problému memoizace funkce jednoho argumentu tak, aby funkci na výpočet členů Fibonacciho posloupnosti šlo napsat elegantněji. Napíšeme makro `defmemfun`, které bude fungovat stejně jako makro `defun`, ale navíc použije memoizaci k zapamatování předchozích výsledků. Zdrojový kód makra zkuste pochopit sami, pomocí definice funkce `fib-2` by to mělo jít:

```
(defmacro defmemfun (name lambda-list &body body)
  (let ((mem-sym (gensym "MEM"))
        (pair-sym (gensym "RES"))
        (var (car lambda-list)))
    `(let ((,mem-sym '()))

       (defun ,name ,lambda-list
         (let ((,pair-sym (assoc ,var ,mem-sym)))
           (unless ,pair-sym
             (setf ,pair-sym (cons ,var (progn ,@body)))
             (setf ,mem-sym (cons ,pair-sym ,mem-sym)))
           (cdr ,pair-sym)))

       )))
```

Funkci na výpočet n-tého členu Fibonacciho posloupnosti můžeme nyní napsat takto jednoduše:

```
(defmemfun fib (n)
  (if (<= n 1)
      1
    (+ (fib (- n 1)) (fib (- n 2)))))
```

Vidíme, že definice spojuje dvě zdánlivě neslučitelné věci: čitelnost zdrojového kódu a efektivitu výpočtu. To je síla maker.

Uvedenou definici je dobré zkusit si expandovat.

7 Pokročilý tisk a práce se symboly

K tisku už umíme používat funkci `print`. K pokročilejšímu tisku slouží funkce `format`.

Funkce `format`

se používá k vytištění formátovaného textu. Jako parametr se jí uvádí tištěný řetězec, v němž je možno použít nejrůznější direktivy uvedené vždy znakem `~`. Nás

budou zajímat direktivy `~s` a `~a`, pomocí nichž lze do řetězce umístit (a tedy vytisknout) libovolný objekt, jenž je použit jako parametr funkce `format`, a direktiva `~%`.

Direktiva `~s` tiskne objekty tak, aby byl výsledek použitelný ve zdrojovém textu programu (stejně jako u funkce `print`), tedy včetně znaků indikujících typ objektů, jako jsou například uvozovky u řetězců (*Simple Print*). Direktiva `~a` je tiskne přijatelněji pro oko, bez těchto znaků (*Aesthetic Print*). Direktiva `~%` způsobí přechod na další řádek (ten je ale možno také zajistit odřádkováním přímo v řetězci i jinak).

První argument funkce `format` určuje cíl tisku. Pokud jako tento argument použijeme symbol `t`, bude funkce tisknout do standardního výstupu, použijeme-li symbol `nil`, funkce vytištěný text vrátí ve formě řetězce jako svou hodnotu (je možno použít i další hodnoty a tisknout jinam, to ale zatím nebudeme potřebovat).

Příklad použití funkce `format`.

Volání

```
(format t "~%List ~s and string ~s" (list 1 1) "Ahoj")
```

přejde ve standardním výstupu na nový řádek a vytiskne

```
List (1 1) and string "Ahoj"
```

(slovo `"Ahoj"` je v uvozovkách).

Volání

```
(let ((n -1))
  (format nil
          "Number ~s is ~a"
          n
          (if (>= n 0)
              "non-negative"
              "negative")))
```

vrátí jako výsledek řetězec

```
"Number -1 is negative"
```

(slovo `"negative"` není v uvozovkách).

Práce se symboly

Už umíme vytvářet nové symboly, ke kterým se nikdo jiný nedostane, pomocí funkce `gensym`. Pokud chceme naopak vytvořit symbol, ke kterému chceme, aby se každý dostal pomocí jeho jména, použijeme funkci `intern`:

```
CL-USER 46 > (intern "TEST")
TEST
NIL
```

Funkce akceptuje jako argument řetězec (nejlépe složený z velkých písmen) a vrátí symbol téhož názvu. Kromě toho vrátí ještě jednu hodnotu (v daném příkladě to byl symbol `nil`), kterou můžeme ignorovat.

Naopak název daného symbolu získáme pomocí funkce `symbol-name`. To asi nebudeme nikdy potřebovat, ale pro úplnost:

```
CL-USER 48 > (symbol-name 'test)
"TEST"
```

Také lze symbol vytisknout do řetězce funkcí `format`:

```
CL-USER 50 > (format nil "~s" 'test)
"TEST"
```

Teď už můžeme funkci `cc-name` napsat:

```
(defun cc-name (name)
  (intern (format nil "~s-CC" name)))
```

Otázky a úkoly na cvičení

V úlohách nikdy nepoužívejte globální proměnné.

1. Mohlo by makro `time` být funkce? Jak byste ho napsali? (Stačí koncepčně, neprogramujte to.)
2. Co vytiskne a jakou hodnotu vrátí následující výraz?

```
(let ((x 1))
  (let ((x x))
    (setf x 2)
    (print x))
  x)
```

3. Implementujte pomocí uzávěrů (tedy nikoliv pomocí párů!) datovou strukturu *point* reprezentující bod v rovině daný kartézskými souřadnicemi. Struktura bude mít konstruktor `make-point`, selektory `x` a `y` a mutátory `set-x` a `set-y`.

4. Napište funkci `point-distance` zjišťující vzdálenost dvou bodů implementovaných jako struktury z předchozího příkladu. Čím se bude funkce lišit od analogických funkcí z minulého semestru?

5. Implementujte pomocí uzávěrů (tedy nikoliv pomocí párů!) datovou strukturu *circle* reprezentující kruh v rovině daný středem a poloměrem. Poloměr kruhu je číslo, střed je bod reprezentovaný strukturou z minulých příkladů. Struktura bude mít konstruktor `make-circle`, selektory `radius` a `center` a mutátor `set-radius` (mutátor `set-center` neprogramujte).

6. Napište funkci `circle-area`, která zjistí plochu kruhu reprezentovaného datovou strukturou z předchozího příkladu.

7. U datových struktur z předchozích příkladů nám chybí typové predikáty. Jak by bylo možné struktury upravit, aby šlo napsat predikáty `pointp` a `circlep` na ověření typu? Upravte definici struktur a predikáty napište. (Predikáty nemusí fungovat na nic jiného než naše body a kružnice.)

8. Napište funkce `fact` a `last-fact` podle těchto požadavků: funkce `fact` počítá faktoriál z daného čísla, jak jsme zvyklí. Funkce `last-fact` vrací v páru poslední argument a výsledek vypočítaný funkcí `fact`:

```
CL-USER 116 > (fact 3)
6

CL-USER 117 > (last-fact)
(3 . 6)

CL-USER 118 > (fact 20)
2432902008176640000

CL-USER 119 > (last-fact)
(20 . 2432902008176640000)
```

9. Definujte funkci `val`, která při volání s jedním argumentem si jeho hodnotu zapamatuje a při volání bez argumentu ji vrátí:

```
CL-USER 14 > (val 1)
1

CL-USER 15 > (val)
1

CL-USER 16 > (val 2)
2
```

```
CL-USER 17 > (val)
2
```

10. Zkontrolujte, zda vaše funkce `val` z minulé úlohy pracuje správně pro všechny hodnoty argumentů. Například následující chování je nesprávné:

```
CL-USER 19 > (val 10)
10

CL-USER 20 > (val nil)
10                                                    nesprávně

CL-USER 21 > (val)
10                                                    nesprávně
```

    K opravě můžete použít novou možnost `&optional`-parametrů, která je popsána v nové verzi textu k první přednášce.

11. Napište makro `defvalf` pro snadné definovaní funkce z předchozího příkladu. Makro by mělo být napsané tak, aby funkci `val` šlo definovat takto:

```
(defvalf val)
```

    Volitelně můžete i umožnit nepovinné nastavení výchozí hodnoty:

```
(defvalf val 10)
```

    Je třeba v definici makra řešit problém zabrání symbolu a vícenásobného vyhodnocení?