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 9. března 2020

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 expr)
```

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

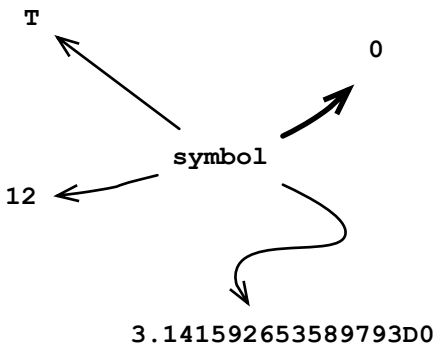
Obecnější verze:

```
(setf symbol1 expr1 ... symboln exprn)
```

je ekvivalentní

```
(progn (setf symbol1 expr1) ... (setf symboln exprn))
```

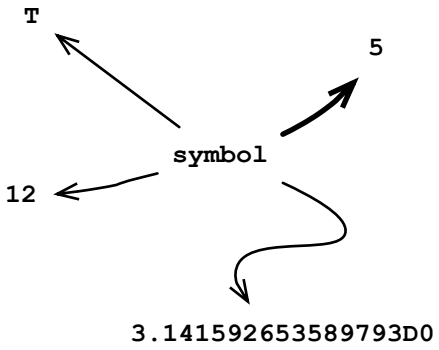
V minulém semestru jsme na jedné přednášce viděli takovýto obrázek:



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 `0`, takže výraz `(+ symbol 5)` se vyhodnotí na `5`. Poté se aktuální hodnota symbolu změní. Jeho vyhodnocením (stále ve stejném prostředí) bychom tedy dostali číslo `5`.

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`

- Příště se 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`.

- Pokud v Listeneru použijeme makro `setf` k nastavení hodnoty proměnné, která nebyla zavedena jako speciální makrem `defvar`, vede pokus o její použití v jiném než globálním prostředí (tedy mimo Listener) k nežádoucímu warningu při kompilaci.

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 Makra postavená na `setf`

V Common Lispu existuje kromě makra `setf` několik užitečných maker, která je používají. 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)
```

A:

```
CL-USER 6 > (psetf x y y z z x)
NIL

CL-USER 7 > (list x y z)
(-1 2 3)
```

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

Makro `rotatef`

```
CL-USER 14 > (setf a 1 b 2 c 3)
3

CL-USER 15 > (rotatef a b c)
NIL

CL-USER 16 > (list a b c)
(2 3 1)
```

Všimněte si, že makro `rotatef` z proměnných jak čte, tak do nich zapisuje. To dělá i následující makro `incf`.

Makro `incf`

Toto makro zjistí hodnotu proměnné (musí být číselná), inkrementuje ji a výsledek uloží zpět do proměnné. 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 (příklad s předchozí hodnotou proměnné `x`):

```
CL-USER 20 > (incf x 10)
16

CL-USER 21 > x
16
```

Makro `decf`

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

Všechna uvedená makra můžeme sami naprogramovat. Někdy je třeba dávat pozor na problém zabrání symbolu.

3 Mutace v lexikálních uzávěrech

Uvažme tento příklad:

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

Funkce `two-functions` vrací seznam dvou funkcí, které sdílejí prostředí vzniku. První z těchto funkcí vrací hodnotu proměnné `x` v tomto prostředí, druhá ji nastavuje. Funkci můžeme vyzkoušet:

```
CL-USER 3 > (setf list1 (two-functions 0))
(#<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` (pro stručnost v následujících obrázcích neoznačuji, že jde o hodnotové vazby symbolů).

Prostředí funkce `two-functions`

symbol	hodnota
x	0

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

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

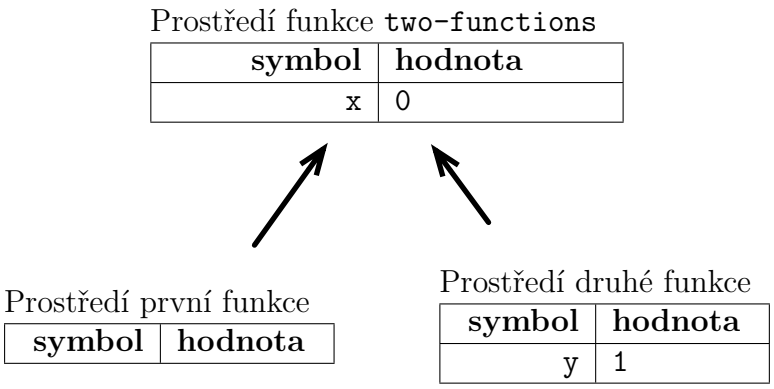
```
(funcall (first list1))
```

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



V těle druhé anonymní funkce je výraz `(setf x y)`. Ten nejprve vyhodnotí symbol `y` (samozřejmě v aktuálním prostředí, tedy v prostředí označeném na obrázku jako prostředí druhé funkce) a dostane hodnotu `1`. Potom najde aktuální vazbu symbolu `x` a její hodnotu změní.

Po této operaci tedy bude prostředí funkce `two-functions` vypadat takto:

Prostředí funkce `two-functions`

symbol	hodnota
x	1

Proto další volání první funkce, tedy vyhodnocení výrazu

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

bude mít jako výsledek číslo `1`.

Všimněme si důležité (ale nám už dobře známé) skutečnosti. Při aplikaci funkce `two-functions` vznikne nové prostředí, které **nepřestane existovat** ani poté, co aplikace funkce proběhne, a kterému i pak lze měnit hodnoty vazeb: vazba symbolu `x` má neomezenou životnost.

A druhá důležitá skutečnost: Víme, že při aplikaci funkce (ale také např. při vyhodnocování `let`-výrazu) vzniká **pokaždé nové prostředí**, nezávislé na prostředí předchozím. Následky tohoto faktu se plně projeví při modifikaci hodnot vazeb.

Po dvojí aplikaci funkce tedy budeme mít dvě různá prostředí. V prvním má nyní symbol `x` hodnotu `1`, v druhém může mít jinou. Např. po aplikaci

```
(setf list2 (two-functions 'a))
```

máme dvě prostředí:

Prostředí funkce `two-functions` (první aplikace)

symbol	hodnota
x	1

Prostředí funkce `two-functions` (druhá aplikace)

symbol	hodnota
x	a

Proto můžeme hodnotu vazby symbolu `x` v jednom z nich změnit, aniž by to ovlivnilo druhé prostředí. Obě prostředí můžeme používat současně (přes proměnné `list1` a `list2`).

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)))))
```

Poznámka: 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 Příklad: 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 budou přidruženy dvě funkce, `fact-cc` (jako *call count*), která bude vracet počet dosud provedených aplikací funkce `fact` a `fact-reset-cc`, která počet vynuluje:

```
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

CL-USER 37 > (fact-reset-cc)
0

CL-USER 38 > (fact-cc)
0
```

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)
    (incf count)
    (if (= n 0)
        1
      (* n (fact (1- n)))))

  (defun fact-cc ()
    count)

  (defun fact-reset-cc ()
    (setf count 0)))
```

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.)

Vylepšení: makro `defcfun`

Pokud bychom chtěli podobně jako funkce `fact`, `fact-cc` a `fact-reset-cc` definovat další funkce, mohli bychom použít stejnou metodu. To by ovšem vedlo k **nevhodnému opakování kódu**. Problému se 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: (nejprve bychom ale měli navrhnout expanzi, pak až makro)

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

       (defun ,name ,lambda-list
         (incf ,count)
         ,@body)
```

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

        (defun ,(reset-cc-name name) ()
          (setf ,count 0)))))
```

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

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

CL-USER 44 > (reset-cc-name 'fact)
FACT-RESET-CC
```

Funkce napíšeme pomocí už známé funkce `format` a funkce `intern`, která vrátí symbol zadaného názvu:

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

Funkce `intern` 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. Funkce je podobná už známé funkci `gensym` v tom, že také vrací jako hodnotu symbol. Rozdíl mezi nimi je, že zatímco funkce `gensym` vrací symbol, ke kterému se nikdo z venku nedostane, funkce `intern` vrací symbol dosažitelný pomocí svého názvu.

Funkce `cc-name` a `reset-cc-name` tedy můžeme napsat takto:

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

(defun reset-cc-name (name)
  (intern (format nil "~a-RESET-CC" name)))
```

V tomto příkladě jsme se setkali se **symbolickými manipulacemi** ve větším rozsahu než minulý semestr: vytváříme nové symboly zadaného názvu. (Z jiného pohledu jsme ovšem se symboly manipulovali už minule funkcí `gensym`.)

Poznámka: Funkce se volají v čase expanze makra. To je fakt, na který je třeba myslet při kompilaci programu bez vyhodnocování (my se s tímto problémem nesetkáme).

Měřením různých aspektů programu z pohledu rychlosti, použité paměti a podobně se zabývají speciální nástroje zvané *profilery*. Ukázali jsme začátek techniky, jak přidat profiler do Lispu. Pokračování by mohlo být následující: místo názvu `defcfun` bychom použili rovnou název `defun` (což jde), takže použití profileru by pro uživatele bylo zcela transparentní (fungoval by na stejný zdrojový kód). Místo jednoduchých funkcí funkce *`funkce`*`-cc`, *`funkce`*`-reset-cc` bychom zavedli jiný způsob, jak si pamatovat každé volání funkce, čas, po který funkce pracovala, atd. (Profiler programů v Lispu napsaný v Lispu skutečně existuje. Jako obvykle můžete přijmout následující výzvu: jak by se dal napsat profiler ve vašem oblíbeném programovacím jazyce?)

6 Příklad: 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.

Funkci `fib-1` můžete také definovat dříve uvedeným makrem `defcfun` a uvidíte, kolikrát sama sebe volá.

Memoizovaná verze funkce, tedy funkce, která si bude bude pamatovat své dosud vypočítané výsledky, bude mnohem rychlejší:

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

(let ((mem '()))

  (defun fib-2 (n)
    (let ((pair (find n mem :key #'car)))
      (unless pair
        (setf pair (cons n (fib-2-help n)))
        (setf mem (cons pair mem)))
      (cdr pair))))
```

(Tady byla na přednášce chyba a navíc jsem to zvídavému studentovi vysvětlil špatně.)

Funkce je vytvořena v novém lexikálním prostředí s vazbou symbolu `mem` na prázdný seznam. Seznam bude postupně doplňován o páry `(`n `.` $f(n)$`)`, kde n je index a $f(n)$ 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 `find`, kterou už známe), zda hodnotu $f(n)$ už nemá zapamatovanou. Pokud ano, bude v proměnné

`pair` příslušný pár. Jinak bude proměnná obsahovat `nil`. Ve druhém případě funkce hodnotu $f(n)$ vypočítá dříve napsanou funkcí `fib-1`, pár uloží do proměnné `pair` a také ho přidá k seznamu `mem`.

Nyní je jisté, že v proměnné `pair` je je pár `(`n `.` $f(n)$`)`. 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.

Poznámka: dalšího zrychlení bychom dosáhli, kdybychom zapamatované výsledky neukládali do seznamu, ve kterém čas strávený vyhledáváním závisí lineárně na počtu prvků, ale použili vhodnou vyhledávací strukturu s logaritmickým časem vyhledávání. V Lispu jsou na to k dispozici tzv. *hashovací tabulky*.

Vylepšení: 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 (var) &body body)
  (let ((mem-sym (gensym "MEM"))
        (pair-sym (gensym "RES")))
    `(let ((,mem-sym '()))

       (defun ,name (,var)
         (let ((,pair-sym (find ,var ,mem-sym :key #'car)))
           (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.

Otázky a úkoly na cvičení

V úlohách nikdy nepoužívejte globální proměnné.

1. Co vytiskne a jakou hodnotu vrátí následující výraz?

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

2. Napište vlastní verzi makra `incf`.

3. Napište makro `swap`, které vymění hodnoty uložené ve dvou proměnných, a to bez použití makra `rotatef` ani jiného makra postaveného na `setf`:

```
CL-USER 1 > (setf a 1 b 2)
2

CL-USER 2 > (swap a b)
2

CL-USER 3 > (list a b)
(2 1)
```

4. 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`.

5. 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 (přednáška 5)?

6. 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).

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

8. 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.)

9. Napište funkci `move`, která posune bod nebo kružnici o vektor daný dvěma čísly.

10. 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
```

11. 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
```

12. 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?

13. Mohlo by makro `time` být funkce?

14. Napište vlastní zjednodušenou verzi makra `time`, která změří a vytiskne čas potřebný k vyhodnocení výrazu. Makro by mělo vytisknout i informaci, který výraz se vyhodnocoval:

```
CL-USER 5 > (my-time (fib-1 42))
Vyhodnocení výrazu (FIB-1 42) trvalo 2.594s.
433494437
```

Aktuální čas v interních jednotkách zjistíte funkcí `get-internal-run-time` (bez parametrů), počet těchto jednotek za sekundu je uložen v konstantě `internal-time-units-per-second`.