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

9. Funkce vyššího řádu

verze z 3. prosince 2019

1 Funkce jako hodnoty

Na minulých přednáškách jsem několikrát upozornil na velkou podobnost některých naprogramovaných funkcí. Ukažme si jeden příklad. Následující funkce k danému seznamu čísel vrací seznam s každým prvkem o jedničku větším:

```
(defun list-inc (list)
  (if (null list)
      '()
    (cons (+ (car list) 1)
          (list-inc (cdr list)))))
```

Test:

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

Další funkce počítá desítkový logaritmus každého prvku seznamu a výsledky opět vrací v seznamu:

```
(defun list-log-10 (list)
  (if (null list)
      '()
    (cons (log (car list) 10)
          (list-log-10 (cdr list)))))
```

Test:

```
CL-USER 6 > (list-log-10 '(1 10 100 1000 10000))
(0.0 1.0 2.0 3.0 4.0)
```

Vidíme, že definice funkcí se velmi málo liší — jen na jednom místě.

Abychom podobnost zdůraznili, definujme dvě pomocné funkce:

```
(defun 1+ (n)
  (+ n 1))

(defun log-10 (x)
  (log x 10))
```

(Funkci `1+` jsem tady definoval jen pro ilustraci, v Lispu ve skutečnosti přesně tato funkce už je.)

Teď můžeme napsat naše dvě funkce ještě podobněji:

```
(defun list-inc (list)
  (if (null list)
      '()
    (cons (1+ (car list))
          (list-inc (cdr list)))))

(defun list-log-10 (list)
  (if (null list)
      '()
    (cons (log-10 (car list))
          (list-log-10 (cdr list)))))
```

Psát dvě různé funkce, které se liší tak málo jako v tomto příkladě, není vhodné. Problém volá po vytvoření **programové abstrakce**, jak jsme o ní mluvili na druhé přednášce.

Když pomineme rozdíly v našich dvou funkcích zjistíme, že obě dělají následující: *Na každý prvek daného seznamu aplikují nějakou funkci a výsledky vrátí opět v seznamu.*

Rádi bychom tedy jako novou abstrakci napsali funkci, která vytvoří seznam, jehož prvky vznikly aplikací dané funkce na prvky daného seznamu. Aby bylo možné to udělat, musíme umět zadat funkci jako argument jiné funkce. Potřebujeme tedy **pracovat s funkcemi jako s hodnotami**.

Funkce v Lispu jsou hodnoty, tj. mohou být ukládány do proměnných a složek datových struktur a používány jako argumenty i návratové hodnoty jiných funkcí. V Listeneru jsou zapisovány pomocí špičatých závorek `#< ... >`, které říkají, že Listener o nich nemohl vytisknout všechny informace (ale funkce si i tak můžeme prohlížet v inspektoru). Ke zjištění funkce daného jména slouží speciální operátor `function`, k aplikaci funkce dané jako hodnota používáme funkci `funcall`.

Operátor `function`

K danému symbolu vrátí jeho funkci.

```
(function symbol) => funkce
```

Zkratka:

```
#'symbol => funkce
```

Příklad:

```
CL-USER 7 > (function cons)
#<Function CONS 40F008970C>
```

Operátor `function` se poměrně často používá, proto místo něj (podobně jako u operátoru `quote`) můžeme používat zkratku:

```
CL-USER 8 > #'cons
#<Function CONS 40F008970C>
```

Funkce `funcall`

Pokud máme funkci danou jako hodnotu, můžeme ji aplikovat na argumenty pomocí funkce `funcall`:

```
CL-USER 9 > (setf f #'cons)
#<Function CONS 40F008970C>

CL-USER 10 > (funcall f 1 2)
(1 . 2)
```

Použití funkce `funcall`:

```
(funcall fun arg1 ... argn)
```

Aplikuje funkci *fun* na argumenty *arg1 ... argn*. Jejich počet je libovolný, ale musí odpovídat počtu argumentů přijímaných funkcí *fun*. Ještě příklady:

```
CL-USER 11 > (funcall #'list 1 2 3 4)
(1 2 3 4)

CL-USER 12 > (funcall #'list #'list #'list)
(#<Function LIST 40F0085AFC> #<Function LIST 40F0085AFC>)

CL-USER 13 > (setf lst (funcall #'list #'car #'cdr))
```

```
(#<Function CAR 40F007DA8C> #<Function CDR 40F00A9894>)

CL-USER 14 > (car (funcall (cadr lst) lst))
#<Function CDR 40F00A9894>
```

V této části jsme si ukázali různé možnosti používání funkce jako hodnoty, tj. jejich ukládání do proměnných a složek datových struktur a používání jako argumentů i návratových hodnot jiných funkcí. Pokud chceme zdůraznit, že nějaká funkce pracuje s jinými funkcemi jako s hodnotami, říkáme o ní, že je to **funkce vyššího řádu**.

Funkce vyššího řádu

Je to funkce, která je určena k tomu, aby pracovala s jinými funkcemi jako s hodnotami, tj. aby jiné funkce přijímala jako své argumenty případně aby funkce vracela jako svůj výsledek.

2 Mapování

Teď už můžeme vyřešit problém z předchozí části:

```
(defun my-mapcar (fun list)
  (if (null list)
      '()
    (cons (funcall fun (car list))
          (my-mapcar fun (cdr list)))))

(defun list-inc (list)
  (mapcar #'1+ list))

(defun list-log-10 (list)
  (my-mapcar #'log-10 list))
```

Funkce `my-mapcar` je zjednodušenou variantou funkce `mapcar`, která v Lispu už je. V dalších příkladech budu rovnou používat funkci `mapcar`. Následuje tedy několik dalších příkladů na použití této funkce:

```
CL-USER 20 > (setf l1 '(1 2 3 4 5 6))
(1 2 3 4 5 6)

CL-USER 21 > (mapcar #'- l1)
(-1 -2 -3 -4 -5 -6)
```

```
CL-USER 22 > (mapcar #'1+ l1)
(2 3 4 5 6 7)

CL-USER 23 > (mapcar #'evenp l1)
(NIL T NIL T NIL T)

CL-USER 24 > (setf l2 '((a) (b c) (d e f) (g h i j)))
((A) (B C) (D E F) (G H I J))

CL-USER 25 > (mapcar #'car l2)
(A B D G)

CL-USER 26 > (mapcar #'length (mapcar #'cdr l2))
(0 1 2 3)
```

(Predikát `evenp` zjišťuje, zda je jeho argument sudé číslo. Už jsme se s ním krátce setkali ve čtvrté přednášce.)

Obecná verze funkce `mapcar` připouští i více než jeden seznam. Obecně ji lze použít takto:

```
(mapcar fun list1 list2 ... listn)
```

Funkce ***fun*** pak musí přijímat n argumentů. Jak funkce `mapcar` pracuje v tomto obecném případě, vidíme z příkladů:

```
CL-USER 27 > (mapcar #'+ '(1 2 3 4) '(5 6 7 8))
(6 8 10 12)

CL-USER 28 > (mapcar #'cons '(1 2 3 4) '(5 6 7 8))
((1 . 5) (2 . 6) (3 . 7) (4 . 8))

CL-USER 29 > (mapcar #'* '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))
(45 120 231 384)

CL-USER 30 > (mapcar #'list '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))
((1 5 9) (2 6 10) (3 7 11) (4 8 12))
```

Jedním z (těžších) úkolů k této přednášce je naprogramovat tuto obecnou verzi funkce `mapcar`.

Pomocí funkce `mapcar` lze zjednodušit mnohé funkce, které jsme si už napsali na práci se seznamy. Například funkci `sum-lists-2` na součet dvou seznamů jako vektorů jsme na šesté přednášce napsali takto:

```
(defun sum-lists-2 (list1 list2)
  (if (null list1)
      ()
      (cons (+ (car list1) (car list2))
            (sum-lists-2 (cdr list1) (cdr list2)))))
```

Teď ji můžeme zjednodušit:

```
(defun sum-lists-2 (list1 list2)
  (mapcar #'+ list1 list2))
```

Zajímavé je, že další funkci, která pracuje se seznamem jako s vektorem a která je funkci `sum-lists-2` značně podobná, zjednodušit zatím nemůžeme. Jde o funkci na vynásobení všech prvků seznamu daným číslem:

```
(defun scale-list (list factor)
  (if (null list)
      ()
    (cons (* factor (car list))
          (scale-list (cdr list) factor))))
```

Za chvíli si ukážeme, jak to jde udělat.

3 Hodnotová a funkční vazba

Už jsme si všimli, že názvy funkcí v Lispu nekolidují s názvy proměnných. Například tato funkce bude správně fungovat:

```
(defun list-test (list)
  (list list))
```

Test:

```
CL-USER 1 > (list-test (list 1 2))
((1 2))
```

Ve zdrojovém kódu je vždy poznat, zda symbol určuje funkci, nebo proměnnou: funkci určuje, jedině když je na prvním místě vyhodnocovaného seznamu nebo když je argumentem operátoru `function`.

Prakticky si můžeme představit, že existují **dva typy vazeb symbolů** tzv. *hodnotová* a *funkční*. Kdykoli jsme až doteď mluvili o vazbách symbolů, měli jsme na

mysli hodnotovou vazbu. Je to vazba, jejíž hodnotou je hodnota proměnné, kterou symbol označuje (v aktuálním prostředí). Kromě toho může mít každý symbol i vazbu funkční, jejíž hodnotou je funkce (nic jiného než funkce to být nemůže), které je symbol názvem.

Prostředí, jak jsme je popisovali na začátku semestru, mají tedy složitější tvar. U každé vazby je kromě symbolu a jeho hodnoty ještě uvedeno, jakého je vazba typu.

Představme si, že jsme v Listeneru právě vyhodnotili výraz

```
(setf r 0)
```

Globální prostředí si pak můžeme představit takto:

Globální prostředí

symbol	**hodnota**	**typ vazby**
`r`	`0`	*hodnotová*
`pi`	`3.141592653589793D0`	*hodnotová*
`*`	`0`	*hodnotová*
`*`	`#<Function * ...>`	*funkční*
`car`	`#<Function CAR ...>`	*funkční*
`list`	`#<Function LIST ...>`	*funkční*
⋮	⋮	⋮

Při následné aplikaci funkce `list-test` ukázané výše bude její prostředí takové:

Globální prostředí

symbol	**hodnota**	**typ vazby**
`list`	`#<Function LIST ...>`	*funkční*
⋮	⋮	⋮

Prostředí funkce `list-test`

symbol	**hodnota**	**typ vazby**
`list`	`(1 2)`	*hodnotová*

Hledání funkční vazby symbolu probíhá stejně jako hledání vazby hodnotové. Při vyhodnocování výrazu

```
(list list)
```

v těle funkce `list-test` se hledá jednak hodnotová, jednak funkční vazba symbolu `list`. Obě dvě se najdou. První rovnou v prostředí funkce `list-test` a druhá v jeho předkovi, tj. v globálním prostředí.

Už jsme se naučili vytvářet pomocí operátoru `let` nová prostředí s hodnotovými vazbami symbolů (není to ovšem jediný způsob, jak je vytvářet; další jsou operátor `let*` a aplikace uživatelsky definované funkce). K vytváření prostředí s funkčními vazbami symbolů slouží speciální operátor `labels`.

Operátor si nejprve vysvětlíme na příkladě. Na třetí přednášce jsme ukazovali iterativní verzi funkce na výpočet faktoriálu:

```
(defun fact-iter (n ir)
  (if (= n 0)
      ir
    (fact-iter (- n 1) (* ir n))))

(defun fact (n)
  (fact-iter n 1))
```

Takto by mohla vypadat její podoba s použitím operátoru `labels`:

```
(defun fact (n)
  (labels ((iter (n ir)
             (if (= n 0)
                 ir
               (iter (- n 1) (* ir n)))))
    (iter n 1)))
```

Operátor `labels` jsme použili na definování tzv. *lokální funkce* `iter`. Funkce není vidět nikde jinde než v těle funkce `fact`. Při vyhodnocování výrazu s operátorem `labels` totiž vzniká nové prostředí, ve kterém je funkční vazba symbolu `iter` (v našem případě) nastavena na novou funkci. Prostředí je aktuální pouze v těle operátoru `labels`, tedy pouze při vyhodnocování výrazu `(iter n 1)`.

Nová lokální funkce má tedy název `iter`. Její seznam parametrů je `(n ir)` a tělo je následující výraz s operátorem `if`.

Při vyhodnocování výrazu `(fact 10)`, v momentě, kdy je poprvé aplikována lokální funkce `iter`, vypadá aktuální prostředí takto:

Globální prostředí

symbol	**hodnota**	**typ vazby**
`fact`	`#<Function FACT ...>`	*funkční*
⋮	⋮	⋮

Prostředí funkce `fact`

symbol	**hodnota**	**typ vazby**
`n`	10	*hodnotová*

Prostředí operátoru `labels`

symbol	hodnota	typ vazby
`iter`	`#<Closure (LABELS ITER) ... >`	*funkční*

Prostředí lokální funkce `iter`
(aktuální prostředí)

symbol	hodnota	typ vazby
`n`	`10`	*hodnotová*
`ir`	`1`	*hodnotová*

Označení lokální funkce `#<Closure (LABELS ITER) ... >` jsem v obrázku musel zkrátit. Pokud bychom je chtěli zjistit, můžeme si pomoci tiskem:

```
(defun fact (n)
  (labels ((iter (n ir)
             (if (= n 0)
                 ir
               (iter (- n 1) (* ir n)))))
    (print #'iter)
    (iter n 1)))
```

A teď:

```
CL-USER 37 > (fact 10)

#<Closure (LABELS ITER) subfunction of FACT 4060001B64>
3628800
```

LispWorks (má verze na mém počítači) tedy značí lokální funkci `iter` takto:

```
#<Closure (LABELS ITER) subfunction of FACT 4060001B64>
```

Slovo *Closure* překládáme jako *uzávěr*. Je to důležitá vlastnost lokálních funkcí, o které budeme mluvit na příští přednášce. Dále vidíme, že jde o lokální funkci (*podfunkci* funkce `fact` vytvořenou operátorem `labels`.

Operátor `labels` může definovat několik nových lokálních funkcí současně. Jeho obecný tvar je tento:

```
(labels ((jméno1 parametry1 . tělo1)
         (jméno2 parametry2 . tělo2)
                    ⋮
         (jménon parametryn . tělon))
  . tělo)
```

Význam jednotlivých částí:

Jménoi Název i-té lokální funkce.

Parametryi Seznam parametrů i-té *lokální funkce*.

Těloi Tělo i-té lokální funkce. Tělo je vždy seznam výrazů, protože v těle funkce může být více výrazů (kvůli vedlejšímu efektu).

Tělo Tělo samotného výrazu s operátorem `labels`. Opět je to seznam výrazů. Například v posledním příkladě je to dvouprvkový seznam `((print #'iter) (iter n 1))`.

Pomocí lokální funkce můžeme vyřešit problém funkce `scale-list`, o kterém jsme mluvili před chvílí:

```
(defun scale-list (list factor)
  (labels ((prod (x) (* x factor)))
    (mapcar #'prod list)))
```

Pokud si ale zkusíme nakreslit prostředí, která vznikají při aplikaci funkce `mapcar`, zjistíme, že nevíme, proč funkce `scale-list` vlastně funguje! Analýza problému nás vede k tomu, že je třeba správně rozhodnout, v jakém prostředí se při aplikaci lokální funkce vyhodnocuje její tělo. Podrobné řešení této otázky nechám na příští přednášku. Na této přednášce budeme zatím lokální funkce používat bez znalosti toho, co přesně se při jejich aplikaci děje.

4 Funkce vyššího řádu na práci se seznamy

Jednu základní, funkci `mapcar`, jsem už ukazoval. Už víme, že úkonu, který se seznamy provádí, říkáme **mapování**. Další základní úlohy uvedeme teď.

Prohledávání. Funkce `find-t` najde první prvek seznamu, který se rovná zadanému prvku. K testování rovnosti prvků použije zadaný predikát. V případě neúspěchu vrací `nil`.

```
(defun find-t (x list test)
  (cond ((null list) nil)
        ((funcall test x (car list)) (car list))
        (t (find-t x (cdr list) test))))
```

Testy:

```
CL-USER 41 > (find-t 2 '(1 2 3) #'=)
2

CL-USER 42 > (find-t 4 '(1 2 3) #'=)
NIL

CL-USER 43 > (find-t 'b '(a b c) #'eql)
B

CL-USER 44 > (find-t 2 '(1 2 3) #'>)
1

CL-USER 45 > (find-t 2 '(1 2 3) #'<)
3
```

Funkce `find-f` najde první prvek seznamu splňující daný predikát. V případě neúspěchu vrací `nil`.

```
(defun find-f (pred list)
  (cond ((null list) nil)
        ((funcall pred (car list)) (car list))
        (t (find-f pred (cdr list)))))
```

Test:

```
CL-USER 50 > (find-f #'oddp '(2 4 6 7 10))
7
```

Filtrace. V Lispu je funkce `remove`, která z daného seznamu vypustí všechny výskyty daného prvku:

```
CL-USER 57 > (remove 'a '(a b r a k a d a b r a))
(B R K D B R)
```

Funkce by mohla být napsána takto:

```
(defun my-remove (x list)
  (cond ((null list) '())
        ((eql x (car list)) (my-remove x (cdr list)))
        (t (cons (car list) (my-remove x (cdr list))))))
```

To ještě není funkce vyššího řádu, ale její varianty `remove-t` a `remove-f`, vytvořené podobně jako dříve, už ano:

```
(defun remove-t (x list test)
  (cond ((null list) '())
        ((funcall test x (car list)) (remove-t x (cdr list) test))
        (t (cons (car list) (remove-t x (cdr list) test)))))

(defun remove-f (pred list)
  (cond ((null list) '())
        ((funcall pred (car list)) (remove-f pred (cdr list)))
        (t (cons (car list) (remove-f pred (cdr list))))))
```

Testy:

```
CL-USER 58 > (remove-t 3 '(1 2 3 4 5 6) #'<)
(1 2 3)

CL-USER 63 > (remove-f #'evenp '(1 2 3 4 5 6))
(1 3 5)
```

Akumulace (též redukce). Následující funkce vyššího řádu je díky své univerzálnosti použitelná v mnoha situacích:

```
(defun foldr (fun list init)
  (if (null list)
      init
    (funcall fun (car list) (foldr fun (cdr list) init))))
```

Rozmanitost použití funkce je vidět z příkladů:

```
CL-USER 3 > (foldr #'+ '(1 2 3 4) 0)
10

CL-USER 4 > (foldr #'+ '(1 2 3 4) 10)
20

CL-USER 5 > (foldr #'* '(2 4 5) 1)
```

```
40

CL-USER 6 > (foldr #'cons '(a b c d) '())
(A B C D)

CL-USER 7 > (foldr #'cons '(a b c d) '(e f g h))
(A B C D E F G H)

CL-USER 8 > (foldr #'list '(a b c d) '(e))
(A (B (C (D (E)))))

CL-USER 9 > (foldr #'append '((a b) (c d e) (f)) '())
(A B C D E F)
```

Jak můžeme vidět, funkci `foldr` můžeme použít ke zjednodušení definic některých funkcí. Například

```
(defun append-2 (list1 list2)
  (foldr #'cons list1 list2))

(defun my-remove (x list)
  (labels ((rem (el accum)
             (if (eql x el) accum (cons el accum))))
    (foldr #'rem list '())))
```

Vidíme také, že funkci `foldr` můžeme použít k sečtení nebo vynásobení všech prvků daného seznamu.

Jako jednu úlohu k šesté přednášce jste programovali funkci `scalar-product` na výpočet skalárního součinu dvou vektorů zadaných seznamy. Řešení mohlo vypadat třeba takto:

```
(defun scalar-product (list1 list2)
  (if (null list1)
      '()
    (+ (* (car list1) (car list2))
       (scalar-product (cdr list1) (cdr list2)))))
```

Se znalostí obsahu této přednášky funkci můžeme přepsat:

```
(defun scalar-product (list1 list2)
  (foldr #'+ (mapcar #'* list1 list2)))
```

5 Explicitní aplikace funkcí `apply`

Během vyhodnocování seznamu, jehož první prvek jméno funkce, dochází k aplikaci této funkce na seznam argumentů, které vzniknou vyhodnocením prvků *cdr* vyhodnocovaného seznamu. Funkce `apply` umožňuje aplikovat zadanou funkci na seznam argumentů vzniklý až během práce programu:

```
CL-USER 1 > (apply #'+ '(1 2 3 4))
10

CL-USER 2 > (apply #'cons '(1 2))
(1 . 2)
```

Funkce přijímá i více než dva argumenty. Pokud je aplikována na více argumentů, tak argumenty předcházející poslednímu (kromě prvního, kterým je vždy aplikovaná funkce) s posledním argumentem spojí do seznamu, takže např. výrazy `(apply #'+ '(1 2 3))`, `(apply #'+ 1 '(2 3))`, `(apply #'+ 1 2 '(3))`, `(apply + 1 2 3 ())` povedou ke stejnému výsledku.

Příklady:

```
CL-USER 3 > (apply #'min 5 2 '(6 1 7))
1

CL-USER 4 > (apply #'* 10 (mapcar #'1+ (list 0 1 2)))
60
```

Funkce `apply` je tedy aplikována na následující argumenty: funkci, libovolný počet hodnot a seznam. Výsledkem aplikace je výsledek aplikace zadané funkce na uvedené hodnoty a prvky uvedeného seznamu.

6 Nepovinné parametry funkce

Víme, že některé funkce (například funkci `+`, ale také mnoho dalších) můžeme aplikovat na libovolný počet argumentů. Ukážeme si, jak lze takové funkce definovat. Bude to jeden ze způsobů, jak zařídit, aby funkce měla tzv. nepovinné parametry. Další způsoby si ukážeme v příštím semestru.

Připomeňme si nejprve, jak jsme dosud používali makro `defun`:

```
(defun název-funkce parametry-funkce
  . tělo-funkce)
```

Jak víme, *`název-funkce`* je symbol, *`tělo-funkce`* je seznam výrazů. *`Parametry-funkce`* je seznam symbolů:

```
(par1 par2 ... parn)
```

říkáme jim *parametry funkce*. Seznam může mít (teoreticky) libovolnou délku, může být i prázdný (pro $n = 0$).

Takto definovanou funkci lze aplikovat na přesně tolik argumentů, kolik prvků má seznam parametrů. Proto také tyto parametry nazýváme *povinné parametry*.

Seznam parametrů funkce může také mít tento tvar:

```
(par1 par2 ... parn &rest rest-par)
```

V něm symbol `&rest` není parametrem funkce, ale určuje, že po něm v seznamu přijde jeden zvláštní parametr, kterému můžeme říkat *parametr pro zbylé argumenty*.

Funkci definovanou s takovým seznamem parametrů můžeme aplikovat na libovolný počet (vyhodnocených) argumentů větší nebo roven n. Prvních n argumentů se jako obvykle naváže na povinné parametry *par1*, *par2* ... *parn*. Na parametr *rest-par* pro zbylé argumenty se pak naváže seznam všech dalších argumentů.

Z následujícího testu plyne, jak to funguje. Definujme funkci

```
(defun rest-test (a b &rest rest)
  (list a b rest))
```

a udělejme několik pokusů:

```
CL-USER 5 > (rest-test 1 2)
(1 2 NIL)

CL-USER 6 > (rest-test 1 2 3)
(1 2 (3))

CL-USER 7 > (rest-test 1 2 3 4 5 6 7)
(1 2 (3 4 5 6 7))

CL-USER 8 > (rest-test 1)

Error: The call (#<Function REST-TEST 41300472F4> 1) does not match
definition (#<Function REST-TEST 41300472F4> A B &REST REST).
```

Nepovinné parametry můžeme u funkcí použít nezávisle na tom, zda jde o funkce vyššího řádu. Jak ale uvidíme z následujících příkladů, u funkcí vyššího řádu je jejich použití výhodné.

Příklad (součet čísel). Představme si, že nemáme funkci na sčítání libovolného počtu čísel, ale jen dvou. Funkci na součet dvou čísel označíme `my-+-2`. Pracovně si ji můžeme definovat takto:

```
(defun my-+-2 (a b)
  (+ a b))
```

Chceme napsat funkci `my-+` na sčítání libovolného počtu čísel. Funkce tedy bude mít nepovinné parametry. Jak už jsme pochopili, můžeme ji definovat pomocí funkce `foldr`:

```
(defun my-+ (&rest numbers)
  (foldr #'my-+-2 numbers 0))
```

Příklad (součet libovolného počtu vektorů). Stejně můžeme zobecnit funkci `sum-lists-2` na součet libovolného nenulového počtu vektorů (nenulového proto, že u nulového počtu bychom nevěděli, kolik prvků má výsledný seznam nul mít):

```
(defun sum-lists (list1 &rest lists)
  (foldr #'sum-lists-2 lists list1))
```

Příklad (druhý způsob). Funkci `sum-lists` můžeme ale napsat i jinak, pomocí funkce `mapcar`:

```
(defun sum-lists (list1 &rest lists)
  (apply #'mapcar #'+ list1 lists))
```

Otázky a úkoly na cvičení

1. Tohle je dlouhá úloha. Na minulých přednáškách a cvičeních jsme napsali několik funkcí, které se současnými znalostmi můžete vylepšit. Týká se to (mimo jiné) těchto funkcí:

   Přednáška 3. Funkci `cos-fixpoint` můžeme napsat pomocí lokální funkce. Také můžeme napsat obecnou variantu, která jako další argument akceptuje funkci, jejíž pevný bod chceme spočítat (v tomto případě je to funkce `cos`). Napište tuto obecnou variantu a vyzkoušejte ji na funkci f z úlohy na výpočet odmocniny Heronovým vzorcem. Dále zkuste přepsat pomocí lokální funkce další funkce, které generují iterativní výpočetní proces a používají pomocnou funkci.

   Přednáška 5. Funkce `my-length`: přepsat pomocí `foldr`. Iterativní verze funkce `my-make-list`: napsat s lokální funkcí.

Přednáška 6. Pomocná funkce `add-to-all`: přepsat pomocí `mapcar`. Vylepšit funkci `merge-sort` tak, aby uživatel mohl zadat predikát na porovnávání: místo pevně zadané funkce `<=` by mohl použít například funkci `>=` a tím setřídit čísla od největšího po nejmenší. V Lispu je také funkce `string<`, pomocí které by šlo uspořádat seznam řetězců podle abecedy.

Přednáška 7. Některé funkce na práci se seznamy uzlů grafu, např. `node-value-multi`, `node-children-multi`, `tree-values-dfs-multi`, `tree-values-bfs-multi`, lze upravit pomocí funkcí `mapcar` nebo `foldr` podle toho, co se zrovna hodí.

Přednáška 8. V příkladech máme funkci `draw-street`, která vykreslí řadu domečků požadované velikosti a počtu. Funkci lze zobecnit tak, aby vykreslila řadu libovolných jiných obrázků. Dále můžeme napsat funkci na kreslení obrázku do sloupce nebo do tabulky.

2. Funkce `find-t` a `find-f` nám nepomohou, pokud nalezeným prvkem seznamu je `nil`. Například

```
CL-USER 51 > (find-f #'symbolp '(1 2 nil t))
NIL
```

Není jasné, jestli vrácená hodnota `nil` je nalezeným prvkem, nebo informací o neúspěchu hledání. (Predikát `symbolp` zjišťuje, zda daná hodnota je symbol.) V Lispu existuje funkce `member`, která hledá prvek v seznamu a pokud ho najde, vrátí celý příslušný zbytek seznamu. Je to vidět z následujícího testu:

```
CL-USER 55 > (member 3 '(1 2 3 4 5))
(3 4 5)

CL-USER 56 > (member 'a '(1 2 3 4 5))
NIL
```

Napište vlastní verzi funkce `member` a dále funkce `member-t` a `member-f` podobně jako funkce `find-t` a `find-f` z přednášky. Pomocí funkcí `member-t` a `member-f` pak funkce `find-t` a `find-f` zjednodušte.

3. Pomocí funkce `foldr` zjednodušte naši definici funkce `my-mapcar`.

4. Napište funkci `foldl`, která se chová stejně jako funkce `foldr`, ale daný seznam prochází zleva doprava. Budete potřebovat pomocnou funkci. Napište ji jako lokální funkci pomocí operátoru `labels`. Než funkci `foldl` začnete psát, odhadněte, co by mělo být hodnotou tohoto výrazu:

```
(foldl #'cons '(1 2 3 4 5) '())
```

5. Pomocí funkcí z této přednášky napište funkci `arithmetic-mean`, která vypočítá aritmetický průměr libovolného nenulového počtu čísel.

6. Zobecněte funkci `equal-lists-p` z cvičení k šesté přednášce, aby přijímala libovolný nenulový počet argumentů.

7. Napište obecnou verzi funkce `my-mapcar`, která se chová stejně jako funkce `mapcar`, tj. připouští libovolný počet argumentů.