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

6. Líné vyhodnocování, přísliby a proudy

verze z 26. března 2020

Abstraktní datovou strukturu jsme definovali jako hodnotu skládající se z více hodnot, ke kterým lze přistupovat funkcemi zvanými selektory a měnit je mutátory. Co přesně znamená, že se struktura *skládá z více hodnot*, nebylo řečeno, ale drželi jsme se představy, že hodnoty jsou uvnitř struktury nějak umístěny. To ovšem není nutné. Struktura může hodnoty vytvářet až na požádání a zvenku to nemusí být poznat. Tomuto přístupu se obecně říká *líné vyhodnocování*.

1 Líné vyhodnocování jako úspora času

Nejprve si ukážeme příklad datové struktury, jejíž jedna složka je spravována technikou líného vyhodnocování. Techniku použijeme v nejjednodušší podobě, tak, jak ji lze použít a jak bývá používána v běžných programovacích jazycích.

Náš příklad vychází z předpokladu, že chceme pracovat se seznamem čísel, u kterého občas potřebujeme znát některé statistické údaje, například aritmetický průměr jeho hodnot. Nechceme tyto údaje počítat, kdykoli si o ně uživatel řekne; jednou vypočítaný údaj si zapamatujeme a příště ho už počítat nebudeme. Proto si seznam uložíme do datové struktury, která bude další údaje uchovávat.

Také ovšem nechceme údaje přepočítávat, kdykoli dojde ke změně seznamu. Vycházíme z předpokladu, že uživatel mění seznam častěji, než zjišťuje údaje, takže jejich neustálé přepočítávání by nemělo smysl. Jak uvidíme, splnění obou těchto podmínek nám zajistí technika líného vyhodnocování.

Naši strukturu nazveme `elist` (jako *encapsulated list*, *zapouzdřený seznam*), ze statistických údajů zatím použijeme jen aritmetický průměr. Pro začátek budeme strukturu reprezentovat tříprvkovým seznamem

```
(elist list average)
```

Prvním prvkem je symbol `elist`, kvůli lepší čitelnosti. Druhým zapouzdřený seznam a třetím aritmetický průměr jeho prvků.

Z pohledu uživatele bude mít struktura konstruktor `elist`, ve kterém lze zadat počáteční obsah zapouzdřeného seznamu:

```
(elist el1 ... eln) => nový zapouzdřený seznam
```

Selektor zavedeme dva: první na seznam (tj. klasický lispový seznam), který je ve struktuře uložen, a druhý na aritmetický průměr jeho prvků:

```
(elist-contents el) => klasický seznam čísel
(elist-average el) => aritmetický průměr prvků (číslo)
```

Zde je ukrytá bezpečnostní díra: pokud by uživatel nějak upravil seznam vrácený funkcí `elist-contents`, struktura to nepozná a údaj o aritmetickém průměru bude chybný. Takové věci se obvykle řeší dvěma způsoby:

1. prostředky jazyka se natvrdo zakáže možnost úpravy seznamu,
2. do dokumentace se napíše něco jako „consequences are undefined if the returned value of `el-contents` is modified“.

Budeme volit druhou možnost, jak je v Lispu obvyklejší.

Koneckonců, nemůžeme uživateli ani zabránit upravovat strukturu zapouzdřeného seznamu jako takovou. Jedná se o obyčejný tříprvkový seznam, tak, kdyby chtěl, to může udělat. Zkušený programátor takové prohřešky ovšem pokud možno nepáchá, protože okamžitá radost z toho, jak něco hezky rychle hacknul, se mu v budoucnu může pěkně vymstít (a jeho uživatelům, což je horší).

Mutátory zapouzdřeného seznamu si na ukázku napíšeme dva. Budou vycházet z funkcí `add-val` a `delete-cons`, které jste měli jako úlohu k minulé přednášce.

```
(elist-add-val value elist cons) => value
(elist-delete-cons cons elist) => elist
```

Funkce `elist-add-val` přijímá jako argumenty hodnotu, zapouzdřený seznam a pár. Daný zapouzdřený 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 vrací (na rozdíl od původní funkce `add-val`) přidanou hodnotu.

Funkce `elist-delete-cons` přijímá jako argument pár a zapouzdřený seznam. Ze seznamu pár odebere a vrátí ho jako výsledek.

Pojďme teď postupně uvedené funkce naprogramovat. Nejprve musím podrobněji vysvětlit, jak budeme pracovat s hodnotou pro aritmetický průměr (základní technikou líného vyhodnocování).

Složka struktury obsahující průměr hodnot (tedy třetí složka seznamu reprezentujícího strukturu), může obsahovat

- Číslo. To je aktuální hodnotou aritmetického průměru prvků seznamu. Při volání jakéhokoliv mutátoru přestává být číslo platné. Nebude se aktualizovat, jen se smaže a místo něj uloží symbol `nil`.
- Symbol `nil`. Ten indikuje, že průměr není znám. Pokud si o něj uživatel zažádá (funkcí `elist-average`), průměr se vypočte, uloží a vrátí jako výsledek.

Konstruktor `elist` vytvoří novou strukturu. Aritmetický průměr zadaných hodnot nebude počítat (je *líný* to dělat), na dané místo jen uloží `nil`:

```
(defun elist (&rest elements)
  (list 'elist
        (copy-list elements)
        nil))
```

Selektor `elist-contents` prostě vrací lispový seznam schovaný ve struktuře:

```
(defun elist-contents (elist)
  (second elist))
```

Selektor `elist-average` by měl postupovat, jak jsem už řekl: podívat se na třetí prvek struktury. Pokud je `nil`, vypočítat průměr a uložit. V obou případech pak hodnotu vrátit.

Napíšeme si nejprve pomocnou funkci na výpočet průměru hodnot lispového seznamu:

```
(defun list-average (list)
  (/ (apply #'+ list)
     (if (null list) 1 (length list))))
```

Poznámky:

- Funkce prochází seznam zbytečně dvakrát, ale kvůli čitelnosti to nebudeme vylepšovat.
- Aritmetický průměr prázdné množiny čísel pokládáme roven nule.

A teď už samotný selektor:

```
(defun elist-average (elist)
  (unless (third elist)
    (setf (third elist) (list-average (elist-contents elist))))
  (third elist))
```

Mutátory se musí postarat o správné zacházení se složkou obsahující průměr. Nejjednodušší bude, když ji každý mutátor nastaví na `nil`:

```
(defun elist-add-val (value elist cons)
  (setf (third elist) nil)
  (setf (second elist)
        (add-val (value (elist-contents elist)) cons))
  value)

(defun elist-delete-cons (cons elist)
  (setf (third elist) nil)
  (setf (second elist)
        (delete-cons cons (elist-contents elist)))
  elist)
```

2 Líné vyhodnocování jako hodnota

Pomocí funkcionálního programování lze zavést zajímavý typ hodnoty, tzv. *příslib*. Příslib můžeme chápat jako dosud nevypočítanou hodnotu, se kterou ovšem můžeme jako s hodnotou pracovat. Můžeme ji tedy někam ukládat, používat jako argument či návratovou hodnotu funkce s tím, že hodnota dosud není vypočtená; její výpočet se provede až v momentě, kdy budeme výsledek opravdu potřebovat.

Jak takovou nevypočítanou hodnotu reprezentovat? Můžeme si uvědomit, že s podobnými hodnotami jsme už pracovali. Například v 10. přednášce minulého semestru jsme pracovali s matematickými posloupnostmi jako s funkcemi, aniž by se počítaly jejich členy. To se udělalo až v momentě, kdy byly členy opravdu potřeba. (K posloupnostem se za chvíli keště vrátíme.)

Podobný trik jsme nejméně jednou použili i na začátku tohoto semestru, když jsme dělali příklady na makra. V úloze 7 z 3. přednášky jsme se snažili vyhnout problému zabrání symbolu v makru `prog1` takto:

```
(defmacro my-prog1 (form1 &body forms)
  `(let ((result ,form1)
         (fun (lambda () ,@forms)))
     (funcall fun)
     result))
```

Při vyhodnocování výrazu s tímto makrem se výrazy `forms` nejprve nevyhodnotí, ale jejich vyhodnocení se *odloží* tím, že se jen vytvoří funkce s tělem rovným těmto výrazům. K vyhodnocení dojde až v momentě, kdy se nám to hodí (výrazem `(funcall fun)`).

Odložit vyhodnocení výrazu (výrazů) tedy můžeme tak, že vytvoříme funkci s daným výrazem jako tělem. Funkci můžeme používat jako hodnotu a vyhodnocení výrazu vynutit zavoláním funkce.

Datová struktura *příslib* pracuje právě tímto způsobem. Její konstruktor `delay` je makro, které k zadanému výrazu vytvoří funkci. Funkce `force` pak vynutí jeho vyhodnocení:

```
(defmacro delay (expr)
  `(lambda () ,expr))

(defun force (promise)
  (funcall promise))
```

Test:

```
CL-USER 1 > (setf p (delay (+ 1 2)))
#<anonymous interpreted function 406000568C>

CL-USER 2 > (force p)
3
```

Aby bylo jasné, že se výpočet vykoná až ve funkci `force`:

```
CL-USER 3 > (setf p (delay (progn (print "Počítám...")
                                  (+ 1 2))))
#<anonymous interpreted function 406000E15C>

CL-USER 4 > (force p)

"Počítám..."
3
```

K výpočtu ovšem dojde pokaždé:

```
CL-USER 5 > (force p)

"Počítám..."
3

CL-USER 6 > (force p)

"Počítám..."
3
```

To bychom mohli vylepšit. Zařídíme, podobně jako v předchozí části o zapouzdřených seznamech, aby se vypočtená hodnota zapamatovala. Uděláme z příslibu jednoduchou datovou strukturu:

```
(promise validp val fun)
```

Je to čtyřprvkový seznam. Na první místo jsme podobně jako minule dali symbol `promise`, aby bylo jasné, co má seznam reprezentovat. Na druhém místě bude údaj, zda je zapamatovaná hodnota validní (tedy zda se dá použít jako hodnota příslibu). Na třetím zapamatovaná hodnota a na čtvrtém funkce, která hodnotu počítá. Přepíšeme tedy makro `delay` a funkci `force` takto:

```
(defun make-promise (fun)
  (list 'promise nil nil fun))

(defmacro delay (expr)
  `(make-promise (lambda () ,expr)))

(defun validp (promise)
  (second promise))

(defun force (promise)
  (unless (validp promise)
    (setf (third promise) (funcall (fourth promise))
          (second promise) t))
  (third promise))
```

Test:

```
CL-USER 7 > (setf p (delay (progn (print "Počítám...")
                                  (+ 1 2))))
(PROMISE NIL NIL #<anonymous interpreted function 40600016BC>)

CL-USER 8 > (force p)

"Počítám..."
3

CL-USER 9 > (force p)
3

CL-USER 10 > (force p)
3
```

Je vidět, že teď se hodnota počítala jen poprvé. Můžeme se podívat, jak náš příslib vypadá teď:

```
CL-USER 11 > p
(PROMISE T 3 #<anonymous interpreted function 4130049B54>)
```

Druhý prvek ukazuje, že hodnota na třetím místě je validní, což ostatně můžeme otestovat přístupovou funkcí:

```
CL-USER 12 > (validp p)
T
```

V případě, že bychom (bohužel) museli používat vedlejší efekt, mohlo by se stát, že vypočítaná hodnota příslibu přestane být validní. V takovém případě, věrni zásadám líného vyhodnocování, novou hodnotu nepočítáme, ale příslib označíme jako ne-validní. uděláme to funkcí `invalidate`:

```
(defun invalidate (promise)
  (setf (second promise) nil))
```

Test:

```
CL-USER 13 > (invalidate p)
NIL

CL-USER 14 > (validp p)
NIL

CL-USER 15 > (force p)

"Počítám..."
3
```

Teď můžeme zkusit znovu naprogramovat naše zapouzdřené seznamy. Tentokrát s líným vyhodnocováním vyřešeným systémově pomocí příslibů (a také s přidanými funkcemi, ať je kód hezčí). Kód ponechám bez komentáře, zkuste se v něm vyznat (funkce `elist` je taková malá výzva).

```
(defun elist (&rest elements)
  (let ((result (list 'elist
                      (copy-list elements)
                      nil)))
```

```
    (setf (third result)
          (delay (list-average (elist-contents result))))
    result))

(defun elist-contents (elist)
  (second elist))

(defun (setf elist-contents) (val elist)
  (setf (second elist) val))

(defun elist-avg-promise (elist)
  (third elist))

(defun elist-average (elist)
  (force (elist-avg-promise elist)))

(defun elist-add-val (value elist cons)
  (invalidate (elist-avg-promise elist))
  (setf (elist-contents elist)
        (add-val value (elist-contents elist) cons))
  value)

(defun elist-delete-cons (cons elist)
  (invalidate (elist-avg-promise elist))
  (setf (elist-contents elist)
        (delete-cons cons (elist-contents elist)))
  (car cons))
```

3 Použití příslibů: proudy

Přísliby lze použít k vytvoření struktur, kterým někteří autoři říkají *proudy*. Ty lze chápat jako „líné seznamy“, neboli páry, jejichž *cdr* se vypočítá, až když je to potřeba.

Formálně lze definovat proud takto: proud je

- symbol `nil`, tzv. *prázdný proud* nebo
- pár, jehož *cdr* je příslib proudu, tj. hodnoty vyhovující opět této definici.

Základní stavební jednotkou proudů jsou tedy páry, jejichž *cdr* obsahuje příslib. Takovým párům budeme říkat *líné páry*. Budeme je vytvářet makrem `cons-stream`, jehož parametry budou

1. hodnota složky *car* nového páru,

2. výraz, kterým příslib vypočte *cdr*:

```
(defmacro cons-stream (a b)
  `(cons ,a (delay ,b)))
```

K příslibu uloženému ve složce *cdr* nebude mít uživatel přístup. Selektor složky *cdr* (nazveme ho `stream-cdr`) totiž příslib vyhodnotí. Selektor složky *car* zavolá prostě funkci `car`:

```
(defun stream-car (s)
  (car s))

(defun stream-cdr (s)
  (force (cdr s)))
```

S proudy budeme pracovat čistě funkcionálně, jejich hodnoty nebudeme měnit. Nedefinujeme tedy žádné mutátory. Operátory `cons-stream`, `stream-car` a `stream-cdr` budou jediné, které k práci s proudy budeme potřebovat. Samozřejmě si ale definujeme i své vlastní.

Díky těmto třem operátorům je práce s proudy velmi podobná práci se seznamy. Jediný rozdíl je v okamžiku, kdy se *cdr* líného páru počítá, jak je vidět z těchto testů:

```
CL-USER 1 > (setf s (cons-stream 1 nil))
(1 PROMISE NIL NIL #<anonymous interpreted function 40600228BC>)

CL-USER 2 > (stream-car s)
1

CL-USER 3 > (stream-cdr s)
NIL

CL-USER 4 > (setf s (cons-stream 1 (progn (print "Počítám...")
                                          (cons-stream 2 nil))))
(1 PROMISE NIL NIL #<anonymous interpreted function 40600229CC>)

CL-USER 5 > (stream-cdr s)

"Počítám..."
(2 PROMISE NIL NIL #<anonymous interpreted function 4060022A3C>)

CL-USER 6 > (stream-car *)
2
```

```
CL-USER 7 > (stream-cdr **)
NIL
```

Napíšeme si pár základních funkcí na práci s proudy. Nejprve vyhodnotíme tento výraz:

```
(shadow '(stream))
```

Tím zabráníme konfliktu symbolu `stream`, který už je v Lispu použit a nám by se přitom hodil. Co přesně se vyhodnocením výrazu stane, nemusíte řešit.

Makro na snadné vytváření proudů. Je to analogie funkce `list`, ale s líným vyhodnocením argumentů (jako vždy si můžete zkusit expanzi):

```
(defmacro stream (&rest exprs)
  (if (null exprs)
      '()
    `(cons-stream ,(car exprs) (stream ,@(cdr exprs)))))
```

Následující test ukazuje, že se argumenty opravdu vyhodnocují líně (všimněte si, kdy se vždycky zavolá funkce `print`), a to v prostředí, ve kterém byly napsány (uvnitř příslibů jsou schovány lexikální uzávěry pamatující si prostředí svého vzniku):

```
CL-USER 8 > (let ((a 1) (b 2) (c 3))
              (stream (print a) (print b) (print c)))

1
(1 PROMISE NIL NIL #<anonymous interpreted function 4060000D9C>)

CL-USER 9 > (stream-cdr *)

2
(2 PROMISE NIL NIL #<anonymous interpreted function 406000083C>)

CL-USER 10 > (stream-cdr *)

3
(3 PROMISE NIL NIL #<anonymous interpreted function 40600008AC>)

CL-USER 11 > (stream-cdr *)
NIL
```

Proudy, podobně jako posloupnosti z minulého semestru, mohou být i nekonečné. Tohle je funkce vracející proud samých jedniček:

```
(defun ones ()
  (cons-stream 1 (ones)))
```

Tahle vrací proud přirozených čísel (s volitelným začátkem, defaultně od nuly):

```
(defun naturals (&optional (from 0))
  (cons-stream from (naturals (1+ from))))
```

A oblíbená Fibonacciho posloupnost, tentokrát v proudu:

```
(defun fib1 ()
  (labels ((iter (n1 n2)
             (cons-stream n2 (iter n2 (+ n1 n2)))))
    (cons-stream 0 (iter 0 1))))
```

Všimněte si, že všechny funkce jsou rekurzivní (u třetí je rekurzivní lokální funkce) a nemají podmínku ukončení rekurze. K zacyklení nedojde, protože každá další rekurzivní aplikace se vykoná vždy až na vyžádání.

Pomocná funkce na převod proudu na seznam:

```
(defun stream-to-list (stream &optional max-count)
  (if (or (null stream)
          (eql max-count 0))
      '()
    (cons (stream-car stream)
          (stream-to-list (stream-cdr stream)
                          (when max-count (- max-count 1))))))
```

Nepovinný parametr je třeba využít, když si nejsme jistí, zda je seznam konečný. Testy:

```
CL-USER 13 > (stream-to-list (stream 1 2 3))
(1 2 3)

CL-USER 14 > (stream-to-list (ones) 20)
(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)

CL-USER 15 > (stream-to-list (naturals) 20)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
```

```
CL-USER 16 > (stream-to-list (fib1) 20)
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181)
```

A podobně jako s posloupnostmi se s proudy dají vykonávat různé operace, aniž by bylo nutné počítat jejich prvky.

Tato funkce vrátí k danému proudu proud obsahující každý druhý jeho prvek:

```
(defun stream-each-other (stream)
  (when stream
    (let ((car (stream-car stream))
          (cdr (stream-cdr stream)))
      (if cdr
          (cons-stream car (stream-each-other (stream-cdr cdr)))
        (cons-stream car nil)))))
```

Test:

```
CL-USER 21 > (stream-to-list (stream-each-other (stream)) 20)
NIL

CL-USER 22 > (stream-to-list (stream-each-other (stream 0)) 20)
(0)

CL-USER 23 > (stream-to-list (stream-each-other (stream 0 1)) 20)
(0)

CL-USER 24 > (stream-to-list (stream-each-other (stream 0 1 2))
20)
(0 2)

CL-USER 25 > (stream-to-list (stream-each-other (naturals)) 20)
(0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38)
```

Vypuštění prvků proudu, které splňují danou podmínku (filtrace):

```
(defun stream-remove-if (test stream)
  (cond ((null stream) '())
        ((funcall test (stream-car stream))
         (stream-remove-if test (stream-cdr stream)))
        (t (cons-stream (stream-car stream)
                        (stream-remove-if test
                                          (stream-cdr stream))))))
```

Vypuštění Fibonacciho čísel, která jsou druhou mocninou:

```
(defun squarep (n)
  (= (expt (round (sqrt n)) 2) n))
```

```
CL-USER 27 > (stream-to-list (stream-remove-if #'squarep (fib1))
                             20)
(2 3 5 8 13 21 34 55 89 233 377 610 987 1597 2584 4181 6765 10946
17711 28657)
```

Mapování na proudech:

```
(defun stream-mapcar-1 (fun s)
  (when s
    (cons-stream
     (funcall fun (stream-car s))
     (stream-mapcar-1 fun (stream-cdr s)))))
```

To bylo pár ukázek, jak lze s proudy pracovat jako s hodnotami, aniž by se počítaly jejich prvky. Jak jsem už psal, výpočty jsou hodně podobné práci se seznamy, ale je tady jeden podstatný rozdíl, který by neměl zaniknout: *Proudy své prvky neobsahují. Obsahují pouze funkce, které na požádání další prvek proudu počítají.* Tyto funkce se volají funkcí `stream-cdr`.

4 Složitější příklady na proudy

Eratosthenovo síto

Je to známá metoda na zjištění seznamu všech prvočísel až po zadané číslo n. Metoda začíná seznamem všech celých čísel od 2 do n, ze kterého postupně vypouští čísla, která určitě prvočísly nejsou. Pro připomenutí se podívejte někam na web, třeba na **Wikipedii**.

Následující funkce vrací posloupnost všech prvočísel ve formě proudu. její výhodou je, že výpočty nutné k nalezení dalšího prvočísla vykonává až v momentě, kdy je to třeba, i když zdrojový kód je napsaný, jako by se úpravy dělaly rovnou se všemi přirozenými čísly od 2, což ho činí celkem dobře čitelným. Zamyslete se sami, jak byste takového efektu dosáhli jiným způsobem (třeba ve svém oblíbeném programovacím jazyce).

```
(defun dividesp (m n)
  (= (rem n m) 0))
```

```
(defun sieve (stream)
  (let ((car (stream-car stream))
        (cdr (stream-cdr stream)))
    (cons-stream car
                 (sieve (stream-remove-if (lambda (n)
                                            (dividesp car n))
                                          cdr)))))

(defun eratosthenes ()
  (sieve (naturals 2)))
```

Duplicity

Následující funkce vypouští ze zadaného proudu duplicitní prvky (funkci `stream-remove` si napište sami):

```
(defun stream-remove-duplicates (stream)
  (when stream
    (let ((car (stream-car stream)))
      (cons-stream car
                   (stream-remove-duplicates
                    (stream-remove car stream))))))
```

Test:

```
CL-USER 40 > (stream-to-list (stream-remove-duplicates
                              (stream 'a 'b 'c 'd 'c 'b 'a)))
(A B C D)
```

Kladná racionální čísla

Funkce `positive-rationals` vytvoří proud všech kladných racionálních čísel. Dělá to tak, že projde všechny dvojice (*čitatel*, *jmenovatel*) a dá je do proudu, ze kterého pak vypustí duplicitní prvky:

```
(defun positive-rationals ()
  (labels ((pr (numerator denominator)
             (cons-stream (/ numerator denominator)
                          (if (= numerator denominator)
                              (pr 1 (1+ denominator))
                            (pr (1+ numerator) denominator)))))
    (stream-remove-duplicates (pr 1 1))))
```

Test:

```
CL-USER 44 > (stream-to-list (positive-rationals) 30)
(1 1/2 1/3 2/3 1/4 3/4 1/5 2/5 3/5 4/5 1/6 5/6 1/7 2/7 3/7 4/7 5/7
6/7 1/8 3/8 5/8 7/8 1/9 2/9 4/9 5/9 7/9 8/9 1/10 3/10)
```

Fibonacciho posloupnost podruhé

Skutečnost, že proudy pracují na principu líného vyhodnocování, čímž se výrazně liší od seznamů, vynikne u některých složitějších příkladů.

Všimněte si, že Fibonacciho posloupnost bez prvních dvou prvků je součtem Fibonacciho posloupnosti a Fibonacciho posloupnosti bez prvního prvku:

0	1	1	2	3	5	8	13	21	34	...
1	1	2	3	5	8	13	21	34	55	...
1	2	3	5	8	13	21	34	55	89	...

Tuto skutečnost můžeme využít k *definování* Fibonacciho posloupnosti.

Předpokládejme, že máme funkci `stream-mapcar`, která pracuje s proudy stejně jako funkce `mapcar` se seznamy. Na rozdíl od funkce `stream-mapcar-1`, kterou jsme psali před chvílí, tedy přijímá libovolný počet argumentů. Takto pak Fibonacciho posloupnost můžeme definovat:

```
(defun fib2 ()
  (let (result)
    (setf result (cons-stream
                  0
                  (cons-stream
                   1
                   (stream-mapcar #'+ result
                                  (stream-cdr result)))))))
```

Šlo by takto definovat (omezenou) Fibonacciho posloupnost jako seznam?

5 Proudy jako iterátory

Proud můžeme chápat jako jakýsi *pozastavený cyklus*. Každá iterace cyklu se provede až na náš pokyn. Abych zdůraznil tuto iterační povahu proudů, ukážu, jak se z proudu dá vytvořit tzv. *iterátor*. To je (v našem přístupu) funkce, která s každým zavoláním vrací novou hodnotu. Dá se tedy také říci, že postupně různé hodnoty *generuje* (jiný název pro iterátor je *generátor*, ale terminologie není jednotná, různí autoři to různě rozlišují).

Iterátory nepatří do funkcionálního programování, jelikož fungují pomocí vedlejšího efektu. Možná se k nim ještě někdy vrátíme.

Tato funkce vytváří iterátor ze zadaného proudu:

```
(defun make-stream-iterator (stream)
  (let ((s stream))
    (lambda ()
      (prog1 (stream-car s)
        (setf s (stream-cdr s))))))
```

Hodnoty iterátoru budeme zjišťovat funkcí `next`:

```
(defun next (iterator)
  (funcall iterator))
```

A test:

```
CL-USER 54 > (setf iterator (make-stream-iterator (fib2)))
#<Closure 1 subfunction of MAKE-STREAM-ITERATOR 4060002C04>

CL-USER 55 > (next iterator)
0

CL-USER 56 > (next iterator)
1

CL-USER 57 > (next iterator)
1

CL-USER 58 > (next iterator)
2

CL-USER 59 > (dotimes (x 20)
               (print (next iterator)))

3
5
8
13
21
34
55
89
144
233
377
610
987
```

```
1597
2584
4181
6765
10946
17711
28657
NIL
```

Otázky a úkoly na cvičení

Příklady na proudy řešte výhradně bez použití vedlejšího efektu.

1. Přidejte do struktury zapouzdřeného seznamu složky `elist-min` a `elist-max` s nejmenší a největší hodnotou v seznamu. Hodnoty se musí počítat technikou líného vyhodnocování, jak jsme ji použili v části 1 tohoto textu.
2. Udělejte totéž, ale s použitím příslibů (část 2).
3. Nešlo by konstruktor `elist` na konci části 2 zjednodušit takto?

```
(defun elist (&rest elements)
  (let ((result (list 'elist
                      (copy-list elements)
                      (delay (elist-average result)))))
    result))
```

   Zdůvodněte podrobně.

4. Přidejte do struktury `elist` mutátor podle vlastní volby. Udělejte to pro obě varianty (z části 1 i z části 2). Měl by to být netriviální mutátor, který ovlivní průměr, minimum i maximum seznamu.
5. Napište výraz, který vrátí nekonečný proud, v němž se budou střídat čísla 0 a 1.
6. Napište funkci `stream-ref`, která k zadanému proudu a indexu vrátí příslušný prvek proudu:

```
(stream-ref (stream 'a 'b 'c 'd) 2) => c
```

7. Napište funkci `stream-each-nth`, která k zadanému proudu vrátí proud obsahující každý `n`-tý prvek původního proudu:

```
CL-USER 1 > (stream-to-list
              (stream-each-nth
                3
                (stream 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j 'k 'l)))
(a d g j)
```

8. Napište funkci `stream-heads`, která k danému proudu vrátí proud součtů jeho začátků, tj. proud obsahující první prvek, součet prvního a druhého, součet prvního, druhého a třetího prvku atd. Příklad:

```
> (stream-to-list (stream-heads (naturals)) 10)
(0 1 3 6 10 15 21 28 36 45)

> (stream-to-list (stream-heads (stream 1 3 2 -5 10)))
(1 4 6 1 11)
```

9. *Prvočíselná dvojčata* jsou zajímavým problémem teorie čísel. Jde o dvojice prvočísel vzdálených o 2. První tři dvojice prvočíselných dvojčat jsou $(3, 5)$, $(5, 7)$, $(11, 13)$. Pomocí funkce `eratosthenes` napište funkci, která vrátí proud prvočíselných dvojčat. Prvky proudu budou tečkové páry; první tři prvky tedy budou `(3 . 5)`, `(5 . 7)`, `(11 . 13)`.

10. Bez použití makra `cons-stream`, ale zato s použitím probíraných funkcí a funkcí, které jste napsali jako řešení těchto úloh, napište funkci, která k danému proudu čísel vytvoří nový proud takto:

    1. Z původního proudu vytvoří proud s každým druhým prvkem,
    2. každý prvek tohoto proudu umocní na 3
    3. a vytvoří proud, jehož prvky jsou součty začátků proudu získaného v předchozím bodě.

    Pokud by například zadaný proud byl proud přirozených čísel od 1, tak výsledkem bude proud čísel

$$\begin{aligned} 1^3 &= 1 \\ 1^3 + 3^3 &= 28 \\ 1^3 + 3^3 + 5^3 &= 153 \\ 1^3 + 3^3 + 5^3 + 7^3 &= 496 \\ &\vdots \end{aligned}$$

    což je proud, který podle matematiků obsahuje všechna sudá dokonalá čísla (*perfect numbers*) kromě čísla 6.

11. V jedné z úloh ke 3. přednášce minulého semestru jste měli počítat aproximace odmocniny ze zadaného čísla a pomocí matematické funkce

$$f(x) = \frac{x + \frac{a}{x}}{2}.$$

Funkce se k výpočtu použije tak, že se nejprve spočítá $f(1)$, pak funkční hodnota funkce f z výsledku atd., vždy se bude počítat funkční hodnota předchozího výsledku. Výsledky jsou stále přesnějšími aproximacemi čísla $\sqrt{a}$. Definujte vhodný proud a pomocí něj pak iterátor, jehož postupná volání budou tyto aproximace vracet.

Příklad. Pokud by $a = 2$ a výsledný iterátor byl v proměnné `iterator`, mohl by se pak otestovat takto (vidíte, že se aproximace ke správné hodnotě blíží velmi rychle):

```
CL-USER 72 > (dotimes (x 7)
               (print (next iterator)))

1
1.5D0
1.4166666666666666D0
1.4142156862745097D0
1.4142135623746899D0
1.414213562373095D0
1.414213562373095D0
NIL
```