Funkce vracející funkce

- funkce může být nejen argumentem, ale také výsledkem funkce
- příklad: vybrat odpovídající aritmetickou funkci

```
(define (vyberfunkci jmeno)
   (cond
      [(string=? jmeno "+") +]
      [(string=? jmeno "–") –] ...
   ))
```

není na začátku seznamu – nevolá se, ale použije jako hodnota

- ((vyberfunkci "+") 79 28) → 107

vyhodnocením vznikne funkce a zavolá se

Vytváření specifických funkcí

- typická funkcionální abstrakce:

```
(define (generator X)
   (local ( (define (F argumenty) výraz) )
      F
   ))
```

- výsledkem je funkce F přijímající *argumenty*
- lokální definice F je uchována
- *výraz* v těle funkce zpravidla závisí na X (parametrizuje vytvářenou funkci)

Příklad: generátor filtrů

```
(define (filtrgen podmínka?)
   (local (
      (define (filtr L)
         (cond [(empty? L) '()]
               [(podmínka? (car L))
                     (cons (car L) (filtr (cdr L)))]
               [else (filtr (cdr L))] )) )
   filtr ))

(define vyberkladne (filtrgen positive?))

(vyberkladne '(26 -8 31 90 -84 0 -57))
```

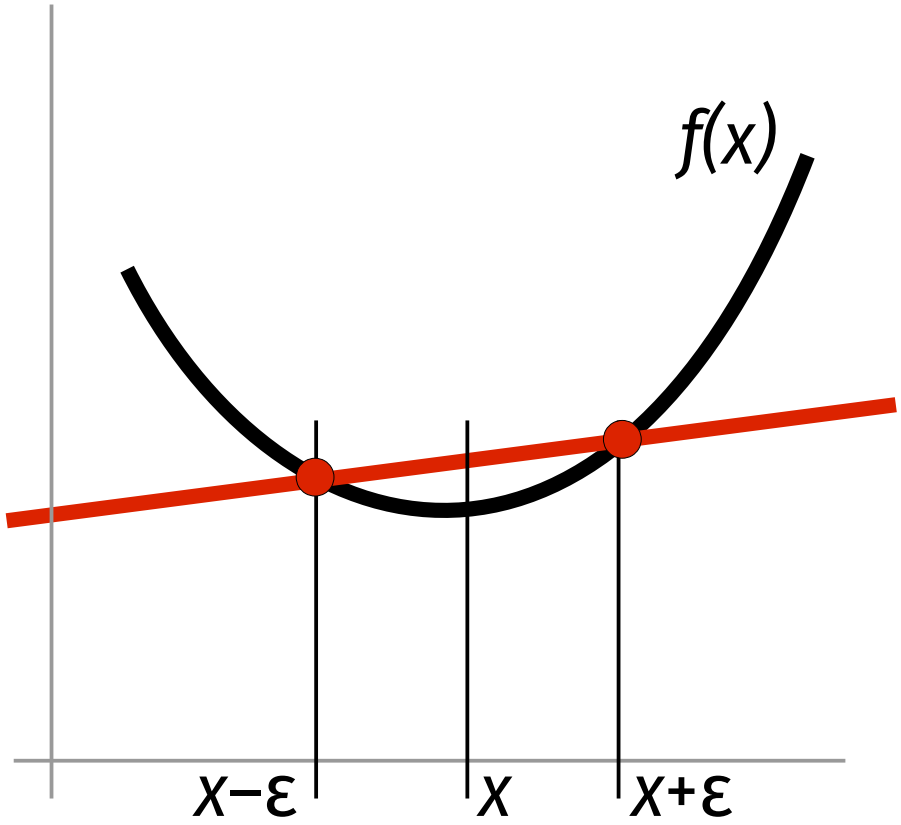
Dvě úrovně abstrakce

- **generické funkce**
  - chování definováno (funkcionálním) argumentem
  - konvenčnější přístup
- **generátory funkcí**
  - vytvářejí funkce na žádost
  - obecný vzor, přizpůsobený konkrétní potřebě argumentem (často funkcionálním)
  - používají se k vytváření konkrétních funkcí

Příklad: numerická derivace

- chceme znát derivaci funkce v určitém bodě
- numerické řešení: odhadneme pomocí dvou blízkých bodů

$$f'(x) = \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}$$



Numerická derivace

```
(define (d/dx f)
    (local (
        (define epsilon 0.001)
        (define (deriv x)
            (/ (– (f (+ x epsilon)) (f (– x epsilon)))
               (* 2 epsilon)))
        )
    deriv
    ))
(define sqrderiv (d/dx sqr))
```

; počítá derivaci x^2 v libovolném x

Funkce 1. třídy

- first-class functions
- funkce je hodnota jako každá jiná – lze ji
  - předávat jako argument
  - vracet jako výsledek
  - ukládat do proměnných a datových struktur
- jeden z požadavků na „opravdové“ funkcionální jazyky

Jak na pomocné funkce

- definování pomocných funkcí může být nešikovné
- např. chceme derivační funkci pro $x^2 + x$

```
(define derivace
   (local ( (define (f x) (+ (sqr x) x)) )
   (d/dx f)))
```

nebo lze local použít tam, kde ho potřebujeme

```
(define derivace
   (d/dx (local ( (define (f x) (+ (sqr x) x)) )
                f)))
```

Lambda výrazy (1)

- druhá varianta je lákavá, ale local je v ní poněkud navíc – komplikuje syntaxi
- Scheme nabízí kratší variantu: lambda
- (v DrRacket Intermed. Student with Lambda a víc)
- syntaxe lambda výrazu:
  (lambda (*argumenty*) *výraz*)
- definuje anonymní funkci; *výraz* je tělem funkce, využívají se v něm *argumenty*
- à la define bez jména funkce

Lambda výrazy (2)

- lambda výraz popisuje, jak z hodnot argumentů vypočíst hodnotu funkce
- funkci definovanou pomocí lambda nelze později volat (nemá jméno), ale lze ji použít jako hodnotu funkcionálního argumentu jiné funkce nebo uložit do konstanty – tím jméno dostane
- náš derivační příklad:

```
(define derivace
   (d/dx (lambda (x) (+ (sqr x) x)))
)
```

Lambda výrazy (3)

- lambda výraz
  (lambda (*arg1 ... argN*) *výraz*)
  lze chápat jako
  (local (
  (define (*jméno arg1 ... argN*) *výraz*))
  jméno)
- ve *výrazu* se ale nesmí vyskytovat *jméno* – omezení pro rekurzivní funkce

Lambda výrazy (4)

- vyhodnocením lambda výrazu vznikne funkce (čili oficiálně hodnota) popisující vztah argumenty → výsledek
- lze ji zavolat (ale nemá valný smysl):
  ((lambda (a b) (sqrt (+ (sqr a) (sqr b))))) 3 4)
- **doporučení:** lambda výrazy se hodí pro nerekurzivní funkce, které potřebujete definovat jednorázově jako funkcionální argument jiné funkce

Uchovávání hodnot

- hodnoty a mezivýsledky standardně předávány v argumentech/výsledcích funkcí
- lze uložit permanentně
  (set! *proměnná výraz*)
  - v DrRacket vyžaduje alespoň Advanced Student
  - proměnná musí být dříve zavedena pomocí define
  - účinek je globální – i když provedete uvnitř funkce, hodnota už zůstane změněna
  - prázdný výsledek (void) – cílem je změnit definici, ne generovat hodnotu

Příklad globální změny

```
(define x 5)
(define (x-add! a)
   (local (
      (define nic (set! x (+ x a)))
      )
      x
   ))
```

```
x            ⟶ 5
(x-add! 8)   ⟶ 13
x            ⟶ 13
```

Příprava definic

- použití local pro nastavení hodnoty je neorganické, viz umělá konstanta nic v předchozím příkladu
- **(begin *výraz1***
  ...
  ***výrazN*)**
- vyhodnotí postupně všechny výrazy, výsledky prvních N – 1 ignoruje, výsledek posledního je celkovým výsledkem
- počáteční výrazy typicky set!

Globální změna lépe

```
(define x 5)
(define (x-add! a)
   (begin
      (set! x (+ x a))
      x
   ))

x
(x-add! 8)
x
```

Globální datové struktury

- změny globálních datových struktur jsou typickou aplikací set!
- např. přidání do starého známého telefonního seznamu (globální definice telseznam):

```
(define (pridej-osobu! jmeno cislo)
  (set! telseznam
        (cons (make-osoba jmeno cislo)
              telseznam)))
```

Komentáře

- popis funkce v komentáři by měl zmiňovat účinek na globální datové struktury
- ; pridej-osobu!: řetězec číslo -> void
  ; přidá novou strukturu osoba do glob. telseznam
  (define (**pridej-osobu!** jmeno cislo)
- samostatný komentář by měl popisovat vlastní datovou strukturu
- jména funkcí měnících globální proměnné bývá zvykem zakončovat **!**

set! a funkce

- pomocí set! lze změnit i tělo funkce
  (define (**f** x) (* x x x))
  (f 2) … 8
  (set! **f** (lambda (x) (* x x x x)))
  (f 2) … 16
- změna může být radikálnější – lze změnit počet parametrů, ale i z funkce udělat konstantu a naopak
  (set! **f** 33)
  (f 2) … chyba, f už není funkce

Funkcionální hodnoty (1)

- definice funkce vlastně zavádí novou konstantu, jejíž hodnotou je **lambda-výraz odpovídající tělu**
- (define (**f** x) (* x x x))
  znamená interně
  (define **f** (lambda (x) (* x x x)))
- případná rekurze v těle nevadí – funkce volá hodnotu konstanty, shodou okolností sama sebe

Funkcionální hodnoty (2)

- definice funkce přiřazením jiné funkce přiřadí tělo, pozdější změna přiřazené funkce se neprojeví
- například

```
(define (f x) (* x x x))        f = (lambda (x) (* x x x))
(define g f)                    g = (lambda (x) (* x x x))
(set! f (lambda (x) (* x x x x)))
(f 2) ... 16                    f = (lambda (x) (* x x x x))
(g 2) ... 8
```

- něco jiného by bylo `(define (g x) (f x))`

Funkcionální hodnoty (3)

- pozor na rekurzi
-
```
(define (delka L)
   (cond   [(empty? L) 0]
           [else (+ 1 (delka (cdr L)))]
   ))
(define g delka)
(set! delka "nic nebude")
(g '(a b c))
```
... havaruje – obsahuje volání delka, to ale už není funkce (resp. lambda výraz)

Změny datových struktur

- při nastavení Advanced Student přidává DrScheme při definici datové struktury
  (define-struct kontakt (jmeno telefon))
  navíc další dvě funkce:
  - (set-kontakt-jmeno! *struktura nové_jméno*)
  - (set-kontakt-telefon! *struktura nový_telefon*)
- umožňují změnit hodnotu položky v existující datové struktuře
- označovány jako **mutátory**

Chování datových struktur (1)

- nekopírují se, ale sdílejí
- připomíná chování dat přístupných přes ukazatele

```
(define raz (list (make-kontakt "Hugo" 12345)))
(define dva (list (car raz)))
(set-kontakt-telefon! (car raz) 67890)
(kontakt-telefon (car dva)) ... vydá 67890
```

Chování datových struktur (2)

- při definici

```
(define prvy (make-kontakt "Hugo" 12345))
(define druhy prvy)
```

- interně proběhne něco jako

```
(define struktura1 (make-kontakt "Hugo" 12345))
(define prvy struktura1)
(define druhy struktura1)
```

Omezení set!

- není funkční hodnotou – nelze např.

```
(aplikuj  set!  cosi)
(define  nastav!  set!)
```

- argument určující proměnnou musí být fixní, nelze

```
(set!  (if  (odd?  x)  licha  suda)
       1)
```

- pro mutátory tato omezení neplatí

Zapouzdření

- lokální definice s přístupovou funkcí, která ji mění pomocí set!, se stává trvalou
- lze zavést cosi jako nový typ, jehož instance pak slouží k uložení konkrétních hodnot
- jeden z pilířů striktního objektově orientovaného programování – uložená hodnota je dostupná jen prostřednictvím přístupových funkcí
- **uzávěr (closure)** – okolí funkce (lokální definice) zůstává, dokud funkce existuje

Příklad: Čítač (1)

- funkce, která vytvoří čítač (výsledkem volání je funkce pridej využívající lokální čítač):

```
(define (novy-citac)
    (local (
        (define citac 0)
        (define (pridej)
            (begin   (set! citac (+ 1 citac))
                     citac))
        )
      pridej
    ))
```

lokální proměnná – existuje, dokud existuje funkce pridej

zvýší čítač o 1 a vydá jeho aktuální hodnotu

novy-citac vrací přístupovou funkci

Příklad: Čítač (2)

- použijeme v programu počítajícím sudá/lichá čísla:

```
(define suda (novy-citac))
(define licha (novy-citac))
(define (pocitej L)
   (cond
      [(empty? L) (list (– (suda) 1) (– (licha) 1))]
      [(odd? (car L)) (begin (licha)
                             (pocitej (cdr L)))]
      [else (begin (suda)
                   (pocitej (cdr L)))]
   ))
```

výsledkem dvouprvkový seznam obsahující počet sudých a lichých čísel

Více přístupových funkcí

- často je třeba více než jedna přístupová funkce
- např. čítač:
  - zvýšení
  - zjištění aktuální hodnoty
  - reset na nulu
- funkce definovány lokálně
- řešení: výsledkem bude řídicí funkce, která podle hodnoty parametru provede jednu z lokálních

Lepší čítač (1)

```
(define (novy-citac)
    (local  (
        (define  citac  0)
        (define  (pridej)  (set! citac (+ 1 citac)))
        (define  (reset)  (set! citac 0))
        (define  (hodnota)  citac)
        (define  (vyber  fce)
            (cond
                [(string=?  fce  "+")    (pridej)]
                [(string=?  fce  "0")    (reset)]
                [else                    (hodnota)]
            ))
        )
     vyber
    ))
```

řetězec v argumentu určí volanou funkci

Lepší čítač (2)

```
(define suda (novy-citac))
(define licha (novy-citac))
(define (pocitej L)
   (cond
      [(empty? L) (list (suda "?") (licha "?"))]
      [(odd? (car L)) (begin (licha "+")
                             (pocitej (cdr L)))]
      [else (begin (suda "+")
                   (pocitej (cdr L)))]
   ))
```

Různý počet argumentů

- přístupové funkce čítače nemají argumenty
- co když se různé funkce liší počtem argumentů?
  (např. reset by dostal hodnotu, kterou má nastavit)
- lze předávat jako **další argumenty výběrové funkci**
  - pak se ale musí předávat vždy, i když se nevyužijí
- **výsledkem výběrové funkce** může být **funkce**
  - [(string=? fce "0") reset]
    volá se „zvenčí“ – např. ((suda "0") 100)

vytvořeno s podporou
projektu ESF