Funkcionální programování
Scheme

Jan Hric, KTI MFF UK,1997-2012a

http://ktiml.ms.mff.cuni.cz/~hric

Funkcionální programování - historie

lambda-kalkul, Alonzo Church 1936
LISP 1958 LISt Programming
ML 1973 - Hindley-Millnerův typový systém a odvozování
MetaLanguage - pův: skriptovací jazyk pro dokazovač
Scheme 1975
Common Lisp 1984 (ANSI CL 1994)
CLOS Common Lisp Object System
Erlang 1986: komerční, masivní paralelizmus: tel. ústředny, hot swapping
Haskell 1990
Haskell 98 Stable+lib
OCaml 1996 Objective (Categorical Abstract Mach.Lang)
Scala 2003 integrace s Javou, generuje JVM
F# 2005

SECD 1963 Virtuální a abstraktní stroj pro FP
(Stack, Environment, Code, Dump)
Mercury 1995 Logické progamování

Funkcionální programování

Literatura k Scheme (dialekt Lispu):

- **Structure and Interpretation of Computer Programs**
- **H. Abelson, G. J. Susmann, J. Susmann**
- **MIT Press 1985 – http://mitpress.mit.edu/sicp**

(M. Felleisen, …: How to Design Programs, MIT Press, http://www.htdp.org)

impl: Racket (dříve DrScheme): http://racket-lang.org/

(http://www.teach-scheme.org/)

MIT Scheme http://www.gnu.org/software/mit-scheme/

Výuka na MFF: NAIL078-9, prof. Štěpánek, Lambda-kalkulus a funkcionální programování I a II, 2/1

Proč FP a Scheme

- Základní myšlenky jsou jednoduché a přehledné
  - Nad jednoduchým jazykem se lépe budují rozšíření i teorie
    - (s programy, tj. synt. stromy, se lépe manipuluje; „generátory“ programů)
  - Přímý přepis matem. formulací do (části) programu (☺deklarativní, 😐neopt.)
  - na MIT se učí první jazyk Scheme
- … a aktuální: rysy FP se přidávají do skriptovacích jazyků (Python, Ruby) i jinde: Scala, F#
  - Lambda výrazy a funkce vyššího řádu
  - (Closures)
- … a další perspektivní znalosti
  - Jiný styl: Kompoziční programování (zákl. bloky a glue), bezstavové programy (lépe se skládají), zpracování celých struktur (vs. OOP) – interpretace a rek. průchod, (CPS - Continuation passing style, stavové programování), (jiné návrhové vzory a idiomy)
  - Přizpůsobení jazyka na problém (bottom-up), DSL – doménově specifické jazyky, (kombinátory)
  - Z Haskellu: Typy - implicitní typování, t. třídy, (fantomové t.); monády (pro podporu sideefektů (I/O), backtrackingu, stavu, kontextu, …), 2D layout

Jazyk Scheme 1

interpret: read - eval – print cyklus

prompt >

příklad volání, zapsáno pomocí s-výrazu:

(v Scheme: pouze prefixní notace se závorkami)

> (+ 2 (* 3 5 1)) ; komentář do konce řádku

17 *odpověď*

Lexikální části: ( ) mezera identifikátory čísla, apostrof …

(id. jsou: null? , atom? , number? , eq? , symbol? ; +compl ; set! ; let*)

Syntax: prefixní, první je operátor (tj. funkce), pak operandy; volná syntaxe

(fce a1 a2 ... an)

Bez čárek: složené argumenty jsou (opět) v závorkách

Data i programy mají stejnou syntax: s-výrazy

> (exit) ; ukončení práce

Jazyk Scheme 2

Výpočet: převedení výrazu na vyhodnocení tvar, podle def. fcí

Vyhodnocování – volání hodnotou (tj. dychtivě/eager):

vyhodnocení funkce (!) a argumentů, pak aplikace (vyhodnocené) funkce

Konstanty se taky vyhodnocují: na „objekty“ k nim přiřazené

každý výraz vrací hodnotu (ne nutně jednoduchou ->)

! vrácená hodnota může být taky datová struktura (dvojice, seznam, …) nebo funkce

Hodnota implementačně: jednoduchá hodnota nebo pointr na složenou

na sebe se vyhodnocují: čísla, #f, #t, (), …

Speciální formy – vestavěné operace (sémantika, místo klíčových slov)

Stejný syntaktický tvar jako funkce -> rozšíření jazyka se neodlišuje ;-)

Speciální formy (místo klíčových slov)

define	definice funkcí a hodnot
cond	podmínka, case
if	
lambda	vytvoření funkce jako objektu
let	lokální definice proměnné

Dvojice – Datové struktury

cons	vytvoření dvojice, (datový) konstruktor
car, cdr	selektory první a druhé složky z dvojice
list	vytvoření seznamu (pomocí dvojic)
quote	zabránění vyhodnocování, pro symbolické hodnoty zkracováno jako apostrof ‘

Aritmetika: +,-,*,/,=, <, >=, and, or, not, …

příklady

```
(define (append xs ys); vrací seznam, který vznikne spojením xs a ys
    ( if (null? xs)         ; test na prazdny seznam xs
         ys        ; then – vrať ys
         (cons (car xs) (append (cdr xs) ys))      ; else – rekurze; vracíme složený výraz
)    ) ; nepotřebujeme lok. proměnné – spočítaná hodnota se hned použije jako arg.

(define (isRectTag c) (eq ‘rect (car c)))         ; test tagu ‘rect
(define (toRect c) …)              ; … moc složité -> později
(define (mkRect x y) (cons ‘rect (cons x y)) ); konstruktor, s tagem ‘rect
(define (+compl x y) ; součet komplexních čísel, ve dvojí reprezentaci: rect a polar
  (define (re c) (car (cdr x)))    ; selektory
  (define (im c) (cdr (cdr x)))  ; lokální definice – lépe v knihovně
    ; krátké definice: 0. bez opakování kódu☺, 1. kompoziční prog, 2. nesouvisí s inline
  (if (and (isRectTag x) (isRectTag y))           ; test obou pravoúhlých tvarů
    (mkRect (+ (re x) (re y)) (+ (im x) (im y)))          ; součet po složkách
    (+compl (toRect x) (toRect y)) ))             ; anebo převod a (rekurzivní) volání
```

define

umožňuje pojmenovat funkce a hodnoty

syntaktický tvar:

(define (*jméno formalní_parametry*)
lokalni_definice ; funkci i hodnot
tělo)

dvojice jméno – objekt se uchovávají v prostředí (environment)
abstrakce paměti (tj. dvojic adresa-hodnota)

př:

```
> (define delka 7)
delka          ; define vrací právě definované jméno
> (define (plus4 x) (+ x 4))       ; definice funkce
plus4
> (define (kvadrat x) (* x x ))
kvadrat
> (kvadrat delka)                  ; volání funkce
49
```

cond

podmíněné výrazy, podmínky se vyhodnocují postupně, celková hodnota cond je hodnota výrazu za první pravdivou podmínkou

pokud není žádná podm. pravdivá, hodnota cond je nedefinovaná

pokud interpretujeme výraz jako boolovskou hodnotu, pak výraz nil, tj. (), tj. #f , je *nepravda*, cokoli jiného *pravda*

obecný tvar

```
(cond       (p1 e1)
            (p2 e2)
            ...
            (pn en) )
```

př:

```
(define (abs x)
   (cond    ((> x 0) x)
            ((< x 0) (- x))
            ((= x 0) 0)    ; typicky záchytná podmínka: else, #t
)  )
```

if

obecný tvar

(if *podmínka výraz alternativa*)

př:

```
(define (abs x ) (if (> x 0) x (- x)) )
```

logické spojky:

př:

```
(define (>= x y ) (or (> x y) (= x y)) )
```

- and, or: libovolný počet arg., vyhodnocení zleva, líně/lazy
- not - negace, jeden argument

faktorial - rekurzivně

```
(define (faktorial n)
    ( if (= n 1)
         1
         (* n (faktorial (- n 1)))
)   )
```

vyhodnocování, paměť O(n) – zásobník rozpracovaných volání

(f 3) ~>(if (= 3 1) 1 (…))~>(if #f 1 (* 3 (f (- 3 1))) ~> (* 3 (f (- 3 1)))

(* 3 (f 2))

(* 3 (* 2 (f 1)))

(* 3 (* 2 1))

(* 3 2)

6

faktorial - akumulátor

```
(define (faktorial n)        ; není rekurzivní, pouze „interface“
    (define (f_iter soucin counter n)  ; invariant: soucin = (counter-1)!
        (if (> counter n)
            soucin           ; konec rekurze - vydání akumulatoru
            (f_iter (* soucin counter) (+ counter 1) n)  ; rekurze, TRO
    )    )                   ; konec lokálního define
    (f_iter 1 1 n)           ; tělo, pouze volání vlastní rekurze s akum.
)
```

vyhodnocování, paměť O(1):

```
(faktorial 3)
(f_iter 1 1 3) ~>(if(> c n)s(f_i(* 1 1)(+ 1 1)3)~*>(f_i (* 1 1)(+ 1 1)3)
(f_iter 1 2 3)
(f_iter 2 3 3)
(f_iter 6 4 3)
6
```

gcd

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))
```

let

umožňuje lokální označení hodnoty, tělo se vyhodnocuje v prostředí, kde každá prom. *varI* označuje hodnotu *expI*

Proměnné jsou charakterem matematické, tj. označení hodnoty, ne adresy

obecný tvar:

(let ((*var1 exp1*)
(*var2 exp2*)
...
(*varn expn*))
tělo)

- časté použití: pro zvýšení efektivity
- hodnoty proměnných platí pouze v těle, ne v expI
  - nejde použít pro vzájemně rekurzivní definice

Varianty: let* – dosud definované proměnné platí v dalších expN

letrec – všechny definované proměnné platí ve všech expN – pro rekurzivní definice

let - příklad

```
(define (f x y)
    (let ( (prum     (/ (+ x y) 2)            )
           (kvadrat (lambda (x) (* x x)) ) )   ; def. lok. funkce
       (+ (kvadrat (- x prum)) (kvadrat (- y prum)) )
)   )
(define (qsort xs)
  (if (null? xs) xs
    (let ((pair (split (car xs) (cdr xs)) )) ; vraci dva seznamy
       (append (qsort (car pair))   ; použití první části
                  (cons (car xs) (qsort (cdr pair))) )))) ; a druhé
```

dvojice

(Jediná) složená datová struktura, konstruktor cons, selektory car a cdr [kudr],

> (define a (cons 1 2))

a

> a

(1 . 2) *vytvoří a vypíše se tečka-dvojice*

> (car a)

1

> (cdr a)

2

> (cons ‘1 ‘(2 3))

(1 2 3) *seznam vzniká z dvojic*

> (cdr ‘(1 2 3))

(2 3)

Dvojice 2

- pro pohodlí zápisů: odvozené fce cadr, caddr, ... a uživatelské selektory:

```
> (define (caddr x) (car (cdr (cdr x))) )
> (define (tretiZeCtverice x4) (caddr x4))
> (define (mkCtverice x y u v) (cons x (cons y (cons u v))) )
> (define (mkCtvericeT x y u v)   ; …s tagem
                (cons 'ctverice (cons x (cons y (cons u v)))) )
```

- nemáme unifikaci (ani pattern matching) -> typický přístup na složky: test druhu struktury (tagu) a následně selektor

seznamy

seznamy (i jiné termy) z dvojic

prázdný seznam: nil, ()

```
> (cons 1 (cons 2 (cons 3 ())))
(1 2 3)
```

funkce list: konstrukce seznamu, s proměnným počtem arg.

```
> (list 1 2 3)
(1 2 3)
> (let ((a 1)(b 3)) (list a b (+ a b)) )     ; args. se vyhodnocují
(1 3 4)
> (null? ())            test na prázdny seznam
#t
> (list? 'a)
#f
```

zpracování seznamů

```
(define (nth n xs)    ; vybere n-tý prvek z xs, bez ošetření chyby
    (if (= n 1)                ; podm. ukončení rekurze
        (car xs)               ; koncová hodnota
        (nth (- n 1) (cdr xs))        ; rekurzivní volání, na tělo xs
) )

(define (length xs) ; spočítá délku seznamu
    (if (null? xs)
        0
        (+ 1 (length (cdr xs)))  ; není TRO, length je uvnitř +
)   )
```

reverse

```
(define (reverse xs)
    (if (null? xs)
        ()
        (append (reverse (cdr xs)) (list (car xs)))    ; v O(n^2)
))
```

- Pro lineární složitost: reverse iterativně (s akum. param.)

```
(define (append xs ys)       ; předdefinováno
    ( if (null? xs)
        ys     ; then – vrať ys
        (cons (car xs) (append (cdr xs) ys))      ; else – rekurze
)  )       ; cons - buduje se nová datová struktura
```

lambda

vytvoření anonymní funkce (v interním tvaru, za běhu), tj. konstruktor funkce

- funkci můžeme aplikovat na argument(„selektor“), předat jako parametr (časté použití), vložit do d.s., vypsat (neúspěšně): <<function>>
- “funkce jsou entity první kategorie” (funkce neinteraguje s jinými konstrukty jazyka, nemá výjimky použití): může být argument, výstup, prvek seznamu …
- vliv na styl: vytvoření funkce je laciné, tj. stručné (malá režie) -> krátké funkce

obecný tvar:

(lambda (*formální_parametry*) *lokDefs tělo*)

př:

```
> ((lambda (x) (+ x 1)) 6)            ; funkce přičtení 1
> ((lambda (x y) (+ x y)) 3 4)        ; sečtení argumentů
> ((lambda (x) (+ x y)) 3)            ; přičtení y, hodnotu vezmeme z prostředí
```

define pro funkce je syntaktický cukr, následující výrazy jsou ekv.:

```
(define (succ x) (+ x 1))
(define succ (lambda (x) (+ x 1)))   ; proměnná succ se vyhodnotí na fci
```

lambda

- lambda se používá typicky při předávání funkcionálních argumentů
  - Programy/fce lze parametrizovat (jednorázovým) kódem – tzv f. vyšších řádů, higher-order -> ! Jedna f. pro různé použití
  - Pozn.: „lambdu“ nevyužijete, pokud ji nemáte kam předat (nebo uschovat) -> jiný styl programování

! Funkcionální parametr je obecný způsob předání kódu

Obecnější než výběr z možností, než parametr pro funkci:

Další funkce lze definovat voláním s vhodnými parametry: mapSleva sleva s

```
> map (lambda (x) (* x 0.9)) '(20 30 40)          ; 10% sleva
```

- `map f s` vytvoří nový seznam stejné délky, nové prvky jsou výsledky použití funkce `f` na jednotlivé prvky vstupního seznamu `s`

funkce jako výstupní hodnoty

vytvoření konstantní funkce 1 prom.

```
(define (mkConstFnc c) (lambda (x) c))
    ; příklad closure: „c“ je zachyceno lokálně,  žije s funkcí
> ((mkConstFnc 7) 2)     ; odpověď 7, !dvojitá úvodní závorka
```

- funkce pričítání dané konstanty:

```
(define (mkAddC c) (lambda (x) (+ x c)))
> ((mkAddC 7) 5)          ; argumenty se předávají samostatně (viz Hs)
12
> (define (twice f) (lambda (x) (f (f x)) ))     ; DC pro n aplikací
> ((twice (twice kvadrat)) 3)     ; 3^8
```

Spomeňte si:

Integrál, derivace, ale zde: výpočet numericky, ne symbolicky

Aho-Corasicková: (Q,Sigma, **g**, **f**, **out**) ;-) konečné f.

Sčítání carry-look-ahead: generate, kill, propagate - se skládáním

Filter, qsort – jeden kód pro všechny použití (i typy)

```
(define (filter p xs)        ;ve výstupním s. prvky x€xs, pro které platí p(x)
   (cond ((null? xs)    xs                                    )   ; konec sezn.
         ((p (car xs)) (cons (car xs) (filter p (cdr xs))) )   ; opakování☹
         (else          (filter p (cdr xs))                   ) ));zdvojení nevadí


(define (qs cmp xs)        ; 1. (cmp a b) vrací a < b
   (if (null? xs)          ; 2. cmp je rozumná abstrakce: detaily schované
       xs                  ; 3: třídí vždy vzestupně a vždy celé objekty
       (append (qs cmp (filter (lambda (x) (cmp x (car xs))) (cdr xs)))
         (cons (car xs)             ; přidání hlavy xs „doprostřed“ výsledku
           (qs cmp (filter (lambda (x) (not(cmp x (car xs)))) (cdr xs)))
)   )    )  )      ;; neefektivní: (car xs) se opakuje, odstranit pomocí let
```

qsort – funkcionální parametry

- Předávání cmp – např. jako lambda funkce:

> (qs (lambda (d1 d2) (< (car d1) (car d2))) *dvojice*) ;podle první složky

😐 různé aplikace, typy a polymorfizmus: Vlastní zpracování dat je „delegováno“ do funkc. parametrů: cmp, p (ale bez dedičnosti)

- Lze zavést globální definici, srovnání:

> (define (mujCmp x y) …) ; nevhodné pro jednorázové definice

> (qs mujCmp ‘(1 5 2 4))

„Sestupné“ třídění, s chybou

>* (qs (not mujCmp) ‘(1 5 2 4)) ; not neguje hodnoty, ne fce

> (qs (lambda (x y) (not (mujCmp x y))) ‘(1 5 2 4)) ; OK

- Definice zdvihu (progr. idiom)

> (define (zdvihF2 op f) (lambda (x y) (op (f x y))))

> (qs (zdvihF2 not mujCmp) ‘(1 5 2 4))

Postupná konstrukce funkce skládáním z jednodušších bez použití lambda

Quicksort a split, s fnc. parametrem **cmp**

```
(define (qsort cmp xs)     ; cmp je funkcionální param/2
  (if (null? xs) xs
     (let ((pair (split cmp (car xs) (cdr xs)) )) ; vrací dva seznamy
          (append (qsort cmp (car pair))
                  (cons (car xs) (qsort cmp (cdr pair))) ))))
(define (split cmp x ys)   ; vrací dvojice, tj. cons
    (if (null? ys)
          (cons () ())   ; dvojice prázdných
          (let ((pair (split cmp x (cdr ys))))   ; pamatuje si dvojici z rek.
             (if (cmp (car ys) x)   ; přidá hlavu ys ke správné složce
                 (cons (cons (car ys) (car pair))  (cdr pair)                  )
                 (cons (car pair)                  (cons (car ys) (cdr pair))
) ) ) )
```

Definiční výskyt

Předávání fun.par.

Volání, tj. použití

quote

pro zápis symbolických dat, argument se nevyhodnocuje, typicky složený argument

stručná forma: apostrof

konstanty lze psát bez quote: čísla 3 3.14, znaky #\a, řetězce “a”, logické #t #f

```
> (define a 1)
a
> (define b 2)
b
> (list a b)
(1 2)
> (list (quote a) b)
(a 2)
>  (list ‘a ‘b)    ; stručněji: ‘(a b), tj. (quote (a b))
(a b)
> (car (quote (a b c) ))    ; quote celé struktury, pokud se v ní nic nevyhodnocuje
a
> (list ‘car (list ‘quote ‘(a b c)))          ; stručněji ‘(car (quote (a b c)))
(car (quote (a b c)))                         ; 😐! reprezentace (zdrojového) programu
```

Skládání po úrovních, pokud se část str. vyhodnocuje

- Pozn.: s reprezentací *můžete* manipulovat, s přeloženou (lambda-)funkcí nikoli
- (neprobírané backquote a čárka): `((a ,a) (b ,b)(s ,(+ a b)) ~>((a 1)(b 2)(s 3))

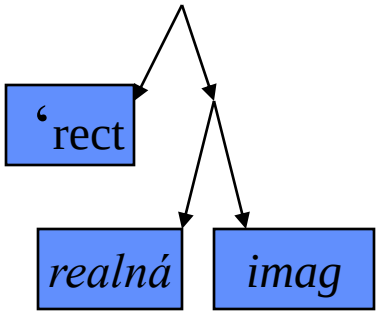
př: komplexní čísla

•aplikace: konstruktory, s “typovou” informací (tag) o druhu struktury

```
(define (mkPolar z fi)            ; konstruktory datové struktury
        (cons ‘polar (cons z fi)) )       ; polární tvar, tag ‘polar
(define (mkRect re im)            ; analogicky konstruktor pro pravoúhlý tvar, tag ‘rect
        (cons ‘rect (cons re im)) )       ; implementace: trojice, ze dvou dvojic
```

-funkce mají/musí přijímat obě reprezentace

```
(define (isRectTag c) (eq? ‘rect (car c)))        ; test tagu
(define (re c) (car (cdr x)))     ; selektory
(define (im c) (cdr (cdr x)))     ; !není to třetí prvek seznamu:
(define (toRect c)        ; převod na pravoúhlý tvar
  (if (isRectTag c)       ; test tagu
        c                 ; ‘rect tvar c bez změny
        (mkRect (*  (cadr c)(cos (cddr c))) ; selektory: velikost, úhel
                (*  (cadr c)(sin (cddr c)))       ; výpočet imag. složky
) ) )                     ; výstup je vždy pravoúhly tvar – v else konstr. mkRect
```



•Jazyk přizpůsobím doméně – budování programu (knihoven) odspodu

Komplexní čísla - součet

```
(define (+compl x y) ; součet komplexních čísel, ve dvojí reprezentaci
  (if (and (isRectTag x) (isRectTag y))
     (mkRect (+ (re x)  (re y))              ; srovnej s (A)
                (+ (im x) (im y)) )
     (+compl (toRect x) (toRect y)) ))
```

- Bez pomocných definic, části kódu se opakují – náchylné k chybám☹:

```
(define (+compl x y)   ; sčítání komplexních čísel
     (if (and (eq? ‘rect (car x)) (eq? ‘rect (car y)))
          (cons ‘rect (cons (+ (cadr x) (cadr y))          ; selektor re, (A)
                                    (+ (cddr x) (cddr y)) ))        ; im
          (if (eq? ‘polar (car x))          ; else: urči špatný arg., převeď a volej rekurzivně
                (+compl (toRect x) y)          ; buď převod x
                (+compl x (toRect y))          ; anebo převod y, neefektivní ☹
)    )     )
```

- Ad (A): varianta těla se selektory se lépe mění (tj. udržuje), ladí (1. kód se používá častěji/opakovaně; 2. je kratší), pochopitelná pro doménového experta …
  - DRY : Don‘t Repeat Yourself

binární stromy

- strom: buď nil, anebo trojice, kterou vytváří konstruktor mk_t

```
(define (mk_t l x r)              ; konstruktor (neprázdneho) vrcholu
    (cons x (cons l r)) )         ; trojice ze dvou dvojic, (implementace: dvě buňky)
(define (getl t) (cadr t))        ; selektory: get levy
(define (getr t) (cddr t))        ; get pravy
(define (getval t) (car t))       ; get hodnoty
(define (nullT? t) (null? t))     ; test druhu struktury
```

- varianta: (méně efektivní na paměť – tři buňky)

```
(define (mk2_t l x r) (list x l r))
```

- Vkládání do binárního stromu

```
(define (insert x t)
  (if (nullT? t) (mk_t () x ())    ; ukončení rekurze, prázdný strom
     (if (< x (getval t)) (mk_t (insert x (getl t)) (getval t) (getr t))
                          (mk_t (getl t) (getval t) (insert x (getr t)))
)))
> (insert 3 (mk_t () 1 (mk_t () 2 () )) )   ; příklad tvorby stromu, s. typicky generujeme
```

DC: konstruktory pro list ve stromě (mkList/1), strom s pouze levým/p. podstromem

Funkcionální parametry, podruhé

```
(define  (map f xs) ; map - transformace xs fcí f po prvcích 1:1
    (if (null? xs)
        ()
        (cons (f (car xs)) (map f (cdr xs))) ))
(define (matrix_map f m)
    (map (lambda (vect) (map f vect)) m) ) ; dvě úrovně map
> (matrix_map (lambda (x) (* 10 x)) '((1 2 3) (4 5 6)) )
(define (transp m)  ; transpozice matice, použití map
  (if (null? (car m))         ; je první řádek prázdný?
    ()                        ; vrať prázdnou matici
    (cons (map car m) (transp (map cdr m))) )) ; spoj 1.ř. a transp. zbytku
```

Pozn☺: do map předáváme *funkce* bez quote, tj. ne jména fcí

DC: zarovnání matice na délku nejdelšího řádku, doplnit 0

Další příklady - cvičení

Výpočet kombinačního čísla

Numerická derivace (a integrace)

a dvojice parciálních derivací

Výpočet Hornerova schématu 1) přímo 2) kompilace do funkce 3) kompilace do výrazu

Filtrace signálu – konvoluce (, skalární součin)

Histogram: (zpracování: filter, map, *reduce*; sort,group,hash)

… průměr, (mergesort), rychlé umocňování (a opakovaná aplikace f. – pro stavové prog.), DFT Fourierova transformace, Fibonacciho čísla – 2 způsoby - a řady, součet čtverců odchylek – pro metodu nejmenších čtverců, a chi^2

Numerická derivace v bodě x

- Předpokládám spojitou funkci *f*, počítám směrnici zleva a zprava v *x* a *x*+-*delta*, rozdíl musí být menší než *eps* (nefunguje pro inflexní body)

```
(define (der f x eps delta)
  (if (> eps (abs (- (smernice f x (+ x delta) delta)
                     (smernice f (- x delta) x delta) )))
      (smernice f x (+ x delta) delta)    ; konec iterace
      (der f x eps (/ delta 2))      ; zmenšení delta, a rekurze
) )
```

- volání: (der (lambda (x) (* x x x)) 2 0.001 0.1)
- DC: použijte let pro zapamatování směrnic
- DC: počítejte pouze derivaci zprava a změna směrnic musí být pod eps
- Dvojice parciálních derivací g(u,v) v bodě (x,y)
  - Příklad na vytváření jednoúčelových funkcí pomocí lambda

```
(define (parcDer  g  x y  eps delta)     ; DC: nebo x a y v dvojici
    (cons (der (lambda (z) (g z y)) x eps delta)
          (der (lambda (z) (g x z)) y eps delta) ))
```

Hornerovo schema 1

Koeficienty od a_0 k a_n v seznamu

DC: koeficienty v opačném pořadí s využitím akumulátoru

- Varianta 1: Přímý výpočet

```
(define (hs1 koefs x)
  (if (null? koefs)
      0
      (+ (car koefs) (* x (hs1 (cdr koefs) x))) ))

> (hs1 '(1 2 3 4) 1.5)  ; pro 4x^3+3x^2+2x+1
```

Hornerovo schema 2 - kompilace

Konvence stejné, hs2 vrací funkci jedné proměnné

```
(define (hs2 koefs)
    (if (null? koefs)
        (lambda (x) 0)   ; výsledek je konstantní funkce
        (lambda (x) (+ (car koefs) (* x ((hs2 (cdr koefs)) x))))
    ))
```

```
> ((hs2 '(1 2 3 4)) 1.5)  ; dvě úvodní závorky:
```

- volání hs2 na svůj arg. (tj. koefs) vrací funkci, která se aplikuje na svůj arg. (tj. hodnotu x)

Hornerovo schema 3 – vytvoření výrazu

Vstup: koeficienty *koefs* a (symbolické) jméno proměnné *pr*

```
(define (hs3 koefs pr)
  (if (null? koefs)
      0
      (list ‘+ (car koefs) (list ‘* pr (hs3 (cdr koefs) pr))) ; (A)
) )
> (hs3 ‘(1 2 3 4) ‘x)
(+ 1 (* x (+ 2 (* x (+ 3 (* x (+ 4 (* x 0))))))))  ; ☹ závorky
> (let ((a 1.5)) (eval (hs3 ‘(1 2 3 4) ‘a)))
```

Backquote(A): `(+ ,(car koefs)(* ,pr ,(hs3 (cdr koefs) pr)))

Backquote (`) buduje strukturu bez vyhodnocování, (skelet)

Čárka (,) vyhodnocuje podvýrazy (uvnitř - vyplňuje díry/hooks)

Filtrace signálu

- Aplikovat daný (lineární) filtr na signál, filtr daný seznamem

```
(define (filtrace koefs xs)
    (if (> (length koefs) (length xs))    ; test délky vstupu, (A)
        ()       ; krátký vstup
        (cons (skalSouc koefs xs) (filtrace koefs (cdr xs)))
)   )
(define (skalSouc xs ys)  ; predp. |ys| >= |xs|
    (if (null? xs)
        0
        (+ (* (car xs) (car ys)) (skalSouc xs (cdr ys))) ))
```

DC: změňte f. filtrace, aby se délka koefs (A) nepočítala opakovaně

DC: převeďte skalSouc na počítání pomocí akumulátoru

DC-pokročilé: pro dané xs vraťte funkci (jednoho argumentu: seznamu ys), která počítá filtr (konvoluci) s koeficienty xs

DC: Fast Discrete Fourier Transform, 2 verze

Ad Komplexní čísla – (jednoduché) kombinátory

- anyCompl použije správnou z dvou daných funkcí (jednoho arg.) podle tagu, ale nechceme zbytečně převádět reprezentace (-lift na var.)

```
(define (anyCompl fRect fPolar)       ; kombinujeme 2 fce pro různé repr.
   (lambda (x)                ; vracíme fci
      (if (isRectTag x)   ; „dispatch“ podle tagu
            (fRect x)     ; použijeme jednu f.
            (fPolar x) ))) ; nebo druhou
```

- Pokud mame jen jednu fci, druhou reprezentaci musíme převést

```
(define (rectF2compl fRect)     ; fce pro ‘rect tvar -> pro lib. kompl.č.
   (lambda (x)                  ; vracíme fci
      ((anyCompl fRect          ; použijeme kombinátor, Arg1 stejný
                 (lambda (y) (fRect (toRect y))) ; Arg2 převod
       ) x ) ))       ; argument pro anyCompl
> ((rectF2compl conjugateR) (mkPolar (- pi 4) 2))          ; použití (+DC)
```

- V Haskellu se tato úroveň práce s funkcemi ztratí (☺zjednoduší)

Další rysy jazyka

Closure – uschování funkce s několika fixovanými argumenty (a/nebo proměnnými v prostředí)

Definice funkcí s proměnným počtem argumentů

set!, set-car!, set-cdr! … – destruktivní přiřazení

Eval – vyhodnocení výrazu (nelze v Hs kvůli typům)

Apply – aplikace f. na args.

Backquote – vytvoření dat/kódu, dovoluje vyhodn. podvýrazy (vs. quote), čárka pro vyhodnocované výrazy

Makra – syntaktické transformace

Struktury, alist – association list

Objekty

Call-with-current-continuation – call/cc - ~skoky, řídící struktury korutiny

(Shell skripty, CGI skripty)

Historie a …

- Syntax pro programy (a la Algol 60) se mezi programátory nechytla – programy jako data
- Ad DSL: P. J. Landin, The Next 700 Programming Languages, Communications ACM, 1966, vol. 9(3):157—166
- *"Greenspun's Tenth Rule of Programming: any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp."* Philip Greenspun
  - *Každý dostatečně rozsáhlý systém v C nebo Fortranu obsahuje ad-hoc, neformálně specifikovanou, chybovou a pomalou implementaci aspoň poloviny Common LISPu*
- Lots of Irritating Simple Parenthesis
- Beating the Averages, http://www.paulgraham.com/avg.html: Vývoj systému pro elektronické obchody (1995): interaktivní vývoj a rychlé doprogramování rysů (na žádost/otázku „A máte tam toto?“)
  - (hypotetický jazyk Blub na škále síly: když se progr. dívá nahoru, neuvědomí si to, protože myslí v Blub -> pohled shora, Eric Raymond: „Lisp udělá programátora lepším“) („větší síla“: nejde naprogramovat funkcí, ale nutno interpretovat zdroják)
- Scheme (vs. LISP): bloková struktura, lexikální rozsahy platnosti proměnných
- …

What makes LISP different

Paul Graham, http://www.paulgraham.com/diff.html

1. Podmíněný příkaz if-then-else (Fortran neměl, jen goto)
2. Funkční typ, jako objekt první kategorie
3. Rekurze
4. Nový koncept promenných, jako ptr. (matematické p)
5. GC - Garbage Collector
6. Programy se skládají z výrazů (ne příkazů)
7. Symboly (nejsou to stringy)
8. Notace pro kód (synt. stromy) – s-výrazy
9. Je vždy dostupný celý jazyk/systém. Nerozlišuje se read-time, compile-time, run-time. (makra, DSL …)