3
Højereordens Funktioner.

Motivation: mapping.
Motivation: parameterombytning.
Sektioner af binære operatorer.
Currying.
Filtrering.
Sammensætning.
Produkt.
Reducering og akkumulering.

Noter

Som omtalt allerede ved første forelæsning, er højereordens funktioner et nøgleområde inden for funktionsorienteret programmering.

Alle scheme funktioner i dette afsnit findes på filen /user/normark/public/ps4/ho-fn.scm.

En funktion som ombytter sine parametre.

- En funktion

  rev: ((X × Y) -> Z) -> (Y × X) -> Z
- rev(f(x,y)) = f(y,x)

```
(define (rev f)
  (lambda (x y)
    (f y x)))
```

```
> (- 2 3)
-1
> ((rev -) 2 3)
1
> (member 3 (list 1 2 3))
(3)
> ((rev member) (list 1 2 3) 3)
(3)
> (cons 5 6)
(5 . 6)
> ((rev cons) 5 6)
(6 . 5)
```

Andre "tilsvarende funktioner":

deriv: (R -> R) × R -> (R -> R)
inverse: (X -> Y) -> (Y -> X)

Noter

Vi har ved tidligere forelæsninger set på funktionen deriv, som jo returnerer den numerisk differentiede funktion af en funktion.

Inverse er funktionen som returnerer den inverse funktion f-inv af en funktion f. Denne opfylder:

f-inv(f(x)) = x for alle x i X.

Der er naturligvis vanskeligheder og udfordringer i at definere funktionen inverse. For hvilke funktioner skal den gælde? Hvordan bærer vi os ad med at bestemme den?

Opgave: En simple (men møjsommelig ide) kunne være at tabellægge funktioner og deres invese, og at lade funktionen inverse benyttet sig af dette. Demonstrer denne ide i praksis i Scheme.

Mapping.

Det er let at skrive en funktion, som "mapper" en bestemt funktion på en liste:

```
(define (map-square lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (map-square (cdr lst)))))
```

Denne kan let **generaliseres** til at mappe en vilkårlig funktion på listen.

```
(define (map f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (map f (cdr lst)))))
```

```
> (map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Det ville være nyttigt og kunne håndtere funktionen (map f) som et førsteklasses objekt.

Noter

Mapping hører så absolut til blandt de klassiske højereordens funktioner. Endvidere er mapping anvendelig i mange praktiske sammenhænge.

Mapning af en funktion på en liste foregår ved at anvende funktionen på hvert af listens elementer, hvorved der fremkommer og opsamles nye elementer, der igen danner en liste. Mapning af en funktion på en liste L giver således en ny liste af samme længde som L.

En praktisk bemærkning: Husk at den tomme liste i Scheme skal skrives som '(). Det er altså ikke nok blot at skrive ().

Mapping opfattet som et funktionale.

```
(define (map f)
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              ((map f) (cdr lst))))))
```

```
> (map square)
#<PROCEDURE>

> (let ((f (map square)))
       (f (list 1 2 3 4 5)))
(1 4 9 16 25)

> ((map square) (list 1 2 3 4 5))
(1 4 9 16 25)
```

- map kaldes undertiden for et *funktionale*.
- map: (X -> Y) -> (X* -> Y*) er udtryk for yderligere abstraktion ift. map: (X -> Y) × X* -> Y*.
- Funktionen (map f), for en vilkårlig funktion f, er en *funktionel byggeklods*, der kan bruges som parameter, f.eks i andre mapninger:

```
> ((map (map square)) '((1 2 3) (7 8 9)))
((1 4 9) (49 64 81))
```


Noter

Man kan opfatte funktionalet map, som vist ovenfor, som en funktionsgenerator. Det tager en funktion som input (som parameter) og giver en funktion som output (som resultat, returværdi).

Egentlig er 'højereordens funktion' og 'funktionale' synonyme betegnelser.

Vi vil dog bestræbe os på at bruge betegnelsen 'funktionale' om de højereordens funktioner, der fremkommer ved at identificere og isolere en funktion ved funktionel abstraktion, som omtalt på forrige slide. En sådan funktion -- et funktionale -- returnerer en anden funktion som resultat.

Hvis man studerer ovenstående version af map nærmere vil man måske få øje på et problem. Det problem vi har i tankerne er

```
(map f)
```

kaldet i den sidste linie i funktionen. Her kaldes map rekursivt, hvorved der genereres og returneres *et nyt funktionsobjekt* fra map. Dette nye funktionsobjekt er lige så god som den oprindelige (map f) funktion, der blev skabt ifm. anvendelsen af map. Men der er ingen grund til at lave nye funktionsobjekter gentagne gange. Det kræver både tid, og tager plads, som blot skal genindrives igen. Nedenunder har vi arrangeret tingene således, at denne re-generering ikke finder sted. I stedet for er local-map nu et frit navn i lambda-udtrykket, som er bundet i en letrec form mellem map niveauet og lambdaudtrykket:

```
(define (map f)
  (letrec
   ((local-map
     (lambda (lst)
       (if (null? lst)
           '()
           (cons (f (car lst))
                 (local-map (cdr lst)))))))
    local-map))
```

Eksempel: (map map).

map: (X ->Y) -> (X* -> Y*)

Map kan specielt anvendes på map:

X = (A -> B), Y = (A* -> B*)

((A -> B) -> (A* -> B*)) -> ((A -> B)* -> (A* -> B*)*) (1)

(map map): (A -> B)* -> (A* -> B*)*

```
> ((map map) (list power2 square))
(#<Function> #<Function>)                    ~ (power2-map square-map)

> (let ((f-map-list ((map map) (list power2 square))))
     ((map (lambda(f) (f (interval 1 10)))) f-map-list))
((2 4 8 16 32 64 128 256 512 1024) (1 4 9 16 25 36 49 64 81 100))
```

Vi anvender successivt power2-map og square-map på listen (1 .. 10)

Noter

Ovenstående er et lidt kompliceret eksempel. Eksemplet er dog spændende idet vi anvender map på sig selv. En sådan form for "selvanvendelse" virker ofte dragende og det forekommer i mange forskellige sammenhænge. (Tænk for eksempel på en compiler til et programmeringssprog: skriv compileren i sproget selv, og lad dernæst compileren oversætte sig selv).

Venstresiden af signaturen (1) angiver kravet til map-funktionen, opfattet som parameter.

Højresiden af (1) angiver selve signaturen af resultatet af (map map).

Det ses altså, at (map map) tager en liste af funktioner, og at den returnerer en liste af mapping-funktioner (hvis I forstår...). I det ovenstående kalder jeg konkret de to resulterende mapping funktioner for power2-map og square-map.

Bemærk, at i en praktisk sammenhæng kan det være upraktisk eller måske ulovligt at redefinere map, som er pre-defineret i Scheme. Det anbefales, at kalde map, som blev defineret på forrige slide, for noget andet end map. I filen med højereordensfunktioner har jeg kaldt den map2.

Her følger definitionerne på funktionerne power2 og interval, som bruges ovenfor i eksemplet.

Først interval, som genererer en *liste* af tal fra og med x til og med y.

```
(define (interval x y)
    (if (> x y)
      '()
       (cons x (interval (+ x 1) y))))
```

Udtrykket (power2 x) returnerer 2^x:

```
(define (power2 x)
  (if (= x 0) 1 (* 2 (power2 (- x 1)))))
```

Sektioner af binære operatorer.

Binær infix operation	x op y	(op x y)
Venstre sektion	[x op] y	((lambda (z) (op x z)) y)
Højre sektion	[op y] x	((lambda (z) (op z y)) x)

```
> (+ 1 5)
6

> (define (1+ x)
    (+ 1 x))

> (1+ 5)
6
```

Generering af sektioner:

```
(define (left-section OP x) ; infix: x OP y, fixing x
    (lambda (y)
      (OP x y)))

(define (right-section OP y) ; infix: x OP y, fixing y
    (lambda (x)
      (OP x y)))
```

```
> (define lig-med-nul (left-section = 0))
lig-med-nul
> (lig-med-nul 5)
#f
```

```
> (define unit-list (right-section cons '()))
unit-list
> (unit-list 5)
(5)
```

PS4 - Højereordens funktioner

Noter

Venstresektionering låser den venstre parameter af en binær operator. Tilsvarende låser højresektionering den højre parametmer.

Som illustreret ovenfor, kan venstresektionering give os en tilnærmet infix notation, hvis vi er smarte med navngivning af den resulterende funktion. Eksempelvis:

(1- 3) = (- 1 3) = 1 - 3.

Der kan i nogle Scheme systemer være leksikalske problemer ved navnet '1-', som vi benyttede ovenfor, idet det starter med et tal. Endvidere kan man forvirre sig selv og andre ved at bruge notationen. Så måske skal vi glemme det igen...

Højresektionerede funktioner kan navngives tilsvarende:

(-1 3) = (- 3 1) = 3 - 1.

Bemærk, at højre sektionering ikke notationsmæssig er så elegant som venstre sektionering, idet vi jo bruger prefix notation i vore funktionsorienterede sprog, i hvert fald i Scheme.

Nogle Scheme systemer understøtter 1- og -1, som beskrevet ovenfor. Men det er ikke standard Scheme, så undlad at bruge dem, hvis de er der. Når du flytter dit program til en anden Scheme implementation, duer det måske ikke.

Flere eksempler:

(define one-minus (left-section - 1))

> (one-minus 5) = -4

(define decrement-by-one (right-section - 1))

> (decrement-by-one 5) = 4

Currying

- *Currying* er en systematik som transformerer en funktion af flere parametre til en funktion af én parameter:

  f: A × B -> C *ikke curried.*

  f: A -> (B -> C) *curried.*

Eksempler:

- + 5 6 fortolkes som [+ 5] 6
- (+ 5 6) fortolkes som ((lambda(x) (+ 5 x)) 6)

- Nogle funktions-orienterede sprog understøtter kun curried funktioner.
- Positiv side: Med notationsmæssig behændighed opnåes, at prefixes af et udtryk er meningfuldt, nemlig en funktion.
- Negativ side: Asymmetri i parametrene.
- Negativ side: Mister muligheden for at betragte parameter-listen som første klasses objekt (en tupel).

Noter

På de første slides i dette kapitel startede vi med at se på ikke curried og curried udgave af mapping.

Venstresektionering af en binær operator er et specialtilfælde af currying.

Ovenfor er [+ 5] funktionen, som lægger 5 til sit argument.

Currying generaliserer naturligvis til funktioner af mere end to parametre:

f: A1 × A2 × A3... × An -> B ikke-curried.

(f a1 a2 a3 ... an)

f : A1 -> (A2 -> (A3 -> ...(An -> B) ...)) curried.

((((f a1) a2) a3) ... an)

Generelt er anvendelse af curried funktioner notationsmæssig tung i lisp på grund af de eksplicitte parenteser. Man får jo udtryk på formen ((((f a1) a2) a3) ... an). Currying af binære funktioner giver sig udslag i ((f a) b), hvilket man vel kan holde ud?

Et funktionsorienteret sprog uden Lisp parenteser er notationsmæssig overlegen her. I et sådant sprog kan vi let opsætte konventioner så ((((f a1) a2) a3) ... an) blot skrives som

f a1 a2 a3 ... an

Konventionen er venstreassociativitet, og at alle funktioner er unære (altså af én parameter).

Generering af curried funktion.

```
(define (curry f)
; f: A × B -> C er en binær,
; ikke curried funktion
  (lambda(x)
    (lambda(y)
      (f x y))))
```

```
(define (uncurry f)
; f: A -> (B -> C) er en curried funktion
  (lambda (x y)
    ((f x) y)))
```

```
> (((curry +) 5) 6)
11

> ((uncurry (curry +)) 5 6)
11

> ((map square) (list 5 6 7))
(25 36 49)

> ((uncurry map) square (list 5 6 7))
(25 36 49)
```


Noter

Signatur for curry:

(A × B -> C) -> (A -> (B -> C))

Signatur for uncurry:

(A -> (B -> C)) -> (A × B -> C)

Opgave: Fortolk følgende udtryk.

(let ((cm (curry -))) (map cm (list 1 2 3 4 5)))

Som det ses, antager vi at map er uncurried.

Hvis resultatet af udtrykket betegnes med res, hvad giver da følgende udtryk?

(map (lambda(x) (x 5)) res)

Løsningen findes på notearket til sidste slide i dette kapitel.

Filtrering.

Filtreringen af listen L med prædikatet P giver den længste delliste af L, hvori alle elementer opfylder P.

```
(define (filter pred)
  (lambda(lst)
    (cond ((null? lst) '())
          ((pred (car lst))
            (cons (car lst)
                  ((filter pred) (cdr lst))))
          (else ((filter pred) (cdr lst))))))
```

```
> (let ((lst (list 1 2 3 4 5 6)))
    (append
          ((filter even) lst)
          ((filter (negate even)) lst)))
(2 4 6 1 3 5)
```

```
; pred:  (X -> Bool) -> (X -> Bool)
(define (negate pred)
  (lambda(x)
    (not (pred x))))
```

PS4 - Højereordens funktioner

Noter

For at definitionen foroven af filtrering skal være præcis, skal der gælde at elementerne i en delliste D af en liste L forekommer i samme rækkefølge som elementerne i L.

Man kan naturligvis i filter indføre en tilsvarende optimalisering som indført for curried mapning (på sliden "Mapning opfattet som et funktionale", se notearket).

Bemært at filter er curried, og har følgende signatur:

(A -> Boolean) -> (A* -> A*).

Funktionssammensætning og krydsprodukt.

Sammensætning:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

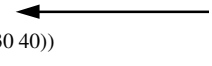
Konstruktion:

```
(define (construct f g)
  (lambda(x)
    (list (f x) (g x))))
```

```
> (define average (compose / (construct sum length)))
average

> (average (list 10 20 30 40))
ERROR:  Non-numeric argument to /
1
(100 4)

> (average (list 10 20 30 40))
25
```



```
(define (compose f g)
  (lambda (x)
    (apply f (g x))))
```

I Scheme er parameterlister ikke af første klasse.
Vi kan bruge lister som parameterlister, men vi kan ikke skelne mellem hvornår vi står med en parameterliste, og hvornår vi står med "alm." liste.

PS4 - Højereordens funktioner

Noter

Signaturen af compose er følgende:

Compose: (Y -> Z) × (X -> Y) -> (X -> Z)

Signaturen af construct er følgende:

Construct: (X -> Y) × (X -> Z) -> (X -> (Y × Z))

Problem med compose: Hvis g returnerer en liste, kunne vi have lyst til at kalde f på denne liste. Et eksempel på dette er

(compose length (left-section interval 3)).

Her ønskes length altså kaldt på et interval (en liste fra 3 til parameteren givet til ovenstående). Et eksempel på en anvendelse af dette (med den uindrammede version af compose):

> ((compose length (left-section interval 3)) 7)

5

Vi får altså længden af intervallet fra og med 3 til og med 7.

I andre tilfælde ville dette naturligvis ikke være tilfældet. I average eksemplet ovenfor ønsker vi at

(construct sum length)

bliver selve parameterlisten til divisionsfunktionen "/". I average eksemplet ovenfor er vi altså oppe mod forskellen mellem

(/ '(10 5))

som fåes via den uindrammede compose (og som giver en fejl) og

(/ 10 5)

der fåes via den indrammede compose, og som er OK.

Problemet er, at vi ikke kan kende forskel på en tupel og en liste, her som returneret fra construct. I Scheme er parameterlister, i form af tupler, ikke 1. klasses objekter.

Reducering.

reduce-right (f, (e_1, e_2, ... ,e_n)) = f(e_1, f(e_2, ... f(e_{n-1}, e_n)))
reduce-left (f, (e_1, e_2, ... ,e_n)) = f(f(... f(f(e_1, e_2), e_3).. e_{n-1}), e_n)

```
(define (reduce-right f)
  (lambda(lst)
    (if (null? (cdr lst))
        (car lst)
        (f (car lst)
           ((reduce-right f) (cdr lst))))))
```

```
(define (reduce-left f)
  (define (help-reduce f res lst)
    (if (null? lst)
        res
        (help-reduce f (f res (car lst))
                     (cdr lst))))
  (lambda(lst)
    (help-reduce f (car lst) (cdr lst))))
```

```
(define (reduce-left f)
  (compose (reduce-right (rev f)) reverse))
```

Anvendelse:

```
> ((reduce-right -) (list 1 2 3 4 5))
3
> ((reduce-left -) (list 1 2 3 4 5))
-13
```

PS4 - Højereordens funktioner

Noter

Ideen i reducering er at komponere elementer parvis (enten fra venstre eller fra højre) i en liste af værdier, således at vi gennem gentagen komponering ender op med netop én værdi. Eksempelvis er

(reduce-left + '(1 2 3)) = ((1 + 2) + 3) = 6.

Højre-reduceringen giver det samme resultat, idet + er associativ.

Signaturen af både reduce-left og reduce-right er følgende:

reduce-right: (T × T -> T) -> T* -> T

I det indrammede skema, hvor vi viser hvordan reduce-funktionerne er definerede, viser vi ikke-curried funktioner, altså funktioner af to parametre. I implementationen nedenfor rammen definerer vi curried funktioner.

I det indrammede skema gør vi os den skjulte antagelse at der mindst er to elementer i den liste af elementer, som vi reducerer. I de viste Scheme funktioner er basistilfældet i rekursionen: "listen er af længden 1", i hvilket tilfælde vi blot returnerer dette ene element. Det er forbundet med begrebsmæssige problemer at give en tom listen til reduceringsfunktionerne (se næste slide og diskusionen der).

Bemærk i reduce-right hvordan vi fanger tilfældet lige inden vi står med en tom liste i lst, og hvordan rekursionsudviklingen standser med at listens to sidste elementer bliver gjort til input til f.

En rekursiv definition af reduce-left i stil med definitionen af reduce-right er vanskelig, idet det er svært og ineffektivt (men ikke umuligt) at arbejde på bagenden af lister i Lisp.

Den halerekursive formulering af help-reduce i reduce-left er dog lige ud ad landevejen (og effektiv).

Bemærk, at der er vist to versioner af reduce-left. Den til venstre er defineret ud fra reduce-right. Den til højre er defineret "direkte" (ved halerekursion).

((reduce-right -) (list 1 2 3 4 5)) = (- 1 (- 2 (- 3 (- 4 5)))) = 3.

((reduce-left -) (list 1 2 3 4 5)) = (- (- (- (- 1 2) 3) 4) 5) = -13.

Akkumulering.

- Reduktion virker ikke på den tomme liste.
- Akkumulering defineres til også at virker på den tomme liste. Værdien af akkumulering af den tomme liste angives eksplicit ved en ekstra parameter.
- accumulate-right: (S × T -> T) -> T -> S* -> T.
- Bemærk forskellen i forhold til reduce-right.

accumulate-right (f, i, (e_1, e_2, ... ,e_n)) = f(e_1, f(e_2, ... f(e_{n-1}, f(e_n , i))))

```
(define (accumulate-right f init)
  (lambda (lst)
    (if (null? lst)
        init
        (f (car lst) ((accumulate-right f init) (cdr lst))))))
```


Noter

Typen af initial-værdien T behøver ikke af være den samme som elementtypen S af listen, som skal akkumuleres. Det hele kommer til at gå op, hvis akkumuleringsfunktionen har signaturen

S × T -> T.

Signaturen af accumulate-right i uncurried form (altså i forhold til situationen i rammen) er:

(S × T -> T) × T × S* -> T.

Der akkumuleres en listen med elementtype S. Den binære fuktion tager et element af typen S og et element af typen T (initielt den eksplicit angivne initalværdi i), og returnerer et element af typen T. Hermed ender vi helt naturligt med at så med et element af typen T i hånden.

En praktisk anvendelse af dette:

```
(define (new-append x y)
     ((accumulate-right cons y) x))
```

Signaturen af cons skal her betragtes som

cons: T × T* -> T*.

Herved bliver signaturen af new-append

new-append: T* × T* -> T*

altså som forventet.

Man kan naturligvis også definere venstre akkumulering; dette overlades dog til læseren.

Bemærk at vi i den indrammede definition af accumulate-right angiver en funktion af hele tre parametre. Dette er usædvanligt i dette kapitel, hvor vi efterhånden har vænnet os til at arbejde med curried funktioner.

Et sammensat eksempel.

Opgaven: Givet to lister L og M af tal. Skriv en funktion, som returnerer listen af alle mulige par (l, m), hvor l er et tal fra L og m er et tal fra M, og hvor hvor ydermere l+m er et lige tal.

```
(define (pair-list e lst)
  ((map (right-section cons e)) lst))

(define (pairs-in-list L M)
  ((map (right-section pair-list L)) M))

(define (all-even-pairs L M)
  ((filter (lambda (p)
             (even (+ (car p) (cdr p)))))
   ((reduce-left append)
    (pairs-in-list L M))))
```

```
> (pair-list 5 (list 1 2 3 4))
((1 . 5) (2 . 5) (3 . 5) (4 . 5))

> (pairs-in-list (list 1 2 3 4) (list 5 6 7 8))
(((1 . 5) (2 . 5) (3 . 5) (4 . 5))
 ((1 . 6) (2 . 6) (3 . 6) (4 . 6))
 ((1 . 7) (2 . 7) (3 . 7) (4 . 7))
 ((1 . 8) (2 . 8) (3 . 8) (4 . 8)))

> (all-even-pairs (list 1 2 3 4) (list 5 6 7 8))
((1 . 5) (3 . 5) (2 . 6) (4 . 6) (1 . 7)
 (3 . 7) (2 . 8) (4 . 8))
```


Noter

Dette er et typisk eksempel på et problem, som lader sig løse ved brug af mapning, filtrering og reducering. Endvidere illustrerer eksemplet, hvordan øvrige funktionaler er nyttige for at danne de funktioner, som bruges i map, filter og reduce.

Den egentlige årsag til at definere tre funktioner er at kunne illustrere mellemresultater på vej mod den endelige løsning. Det er, som bekendt generelt lettere at forstå en løsning, hvis man definerer og navngiver delløsninger.

(right-section pair-list L) svarer pr. definition til funktionen

```
(lambda(y) (pair-list y L))
```

Når denne mappes hen over M får vi *en liste af lister af par*. Den første sådanne liste er alle par dannet ud af L og det første element af M.

Løsning på opgave på slide "Generering af curried funktion":

Udtrykket

```
(let ((cm (curry -)))   (map cm (list 1 2 3 4 5)))
```

er en liste af funktioner, der hhv. subtraherer deres respektive parametre fra 1, 2, 3, 4 og 5. Hvis res er den resulterende liste af funktioner får vi følgende:

```
(map (lambda(x) (x 5)) res) = (-4 -3 -2 -1 0)
```