4
Evalueringsrækkefølge og forsinket evaluering.

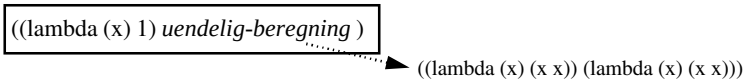
Motiverende eksempel.
Teoretiske resultater.
Sammenligning med parameteroverførsel.
Betingede udtryk.
Lazy evaluering.
Forsinket evaluering i Scheme.
Streambegrebet i Scheme.
Eksempler.
Forfinede streambegreber.
Streams i forhold til imperativ programmering.

PS4 - Evalueringsrækkefølge

Noter

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

Motiverende eksempel.



- Forskellige evalueringsrækkefølger giver forskellige "resultater":
  - Værdien 1.
  - Ikke-terminerende beregning.
- Svarer til to forskellige semantikker af funktioner.
  - Strict ("streng"): En funktion er veldefineret hvis og kun hvis alle funktionens argumenter er veldefinerede.
  - Lenient ("mild"): En funktion kan være veldefineret selv om én eller flere argumenter ikke er veldefinerede.

Noter

Der er naturligvis mange måder at give en parameter, hvis beregning ikke terminerer. Hvis man imidlertid vil undgå eksplicit rekursion, er

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

et godt eksempel på et funktionskald, som blot genererer den samme kaldsekvens om og om igen.

Genskrivningsregler og normalform.

Genskrivningsregler.	**Eksempler.**
α-konvertering. Formelle parameternavne kan erstattes med andre navne, som ikke er frie i kroppen.	(lambda (x y) (f x y)) <=> (lambda (a b) (f a b)) *men ikke* (lambda (x y) (f x y)) <=> (lambda (a f) (f a f))
β-konvertering. Et kald af en funktion kan erstattes af funktionskroppen, hvori formelle parametre erstattes af tilsvarende argumenter.	((lambda(x) (f x)) a) <=> (f a)
η-konvertering. (lambda(x) (e x)) <=> e forudsat x ikke er fri i e.	(lambda (x) (square x)) <=> square *men ikke* (lambda(x) ((lambda(y) (f x y)) x) <=> (lambda(y) (f x y))

Et udtryk er i *normalform* hvis det ikke kan reduceres yderligere ved brug af β-konvertering eller η-konvertering, anvendt fra venstre mod højre (=>).

PS4 - Evalueringsrækkefølge

Noter

Ovenstående viser tre genskrivningsregler (konverteringer) fra fundamental lambda calculus. Formålet med at bringe disse på banen er at kunne definere hvad det vil sige at reducere et udtryk i lambda calculus, og derved beskrive en normalform. Normalformen svarer nøje til det vi plejer at betegne som resultatet af beregningen af et udtryk.

Ovenfor giver vi eksempler i Scheme. Det er klart, at der er forholdsvis stor afstand mellem Scheme og rå Lambda Calculus, men i denne sammenhæng er vi kun interesseret i intuitionen og ideen i genskrivningerne; og ikke så meget i detaljerne og forudsætningerne for reglernes anvendelse. Derfor kan vi godt anvende Schemeudtryk som eksempler.

Den anden konvertering (beta konverteringen) er central. Der mindes om, at netop denne konvertering svarer til anvendelse af den model, som vi i 1. forelæsning kaldte substitutionsmodellen.

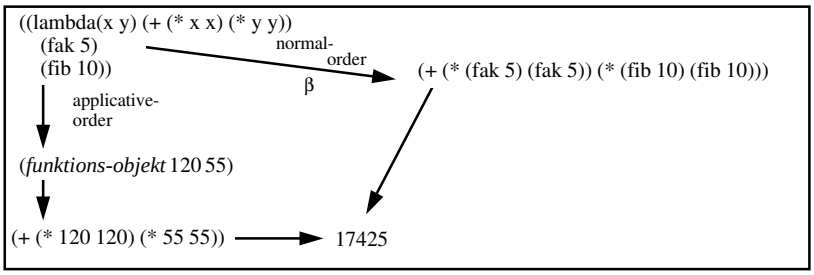
Den tredie konvertering, som kaldes “eta konvertering”, udtaler sig om under hvilke omstændigheder en funktion kan afnestes af en abstraktion (“lambda konvolut”).

I både α og η konvertering er der undtagelser for frie navne. Et frit navn i et udtryk binder sig til en defintion uden for det pågældende udtryk. I α konvertering vil vi introducere en fejlagtig binding til en parameter, hvis ikke reglen om frie navne overholdes. I η konvertering kan vi ikke undvære bindingen af x i parameterlisten, hvis x refereres fra udtrykket e.

Der henvises iøvrigt til Hudaks oversigtsartikel Conception, Evolution, and Application of Functional Programming Languages, Computing Surveys 21, 3, September 89.

Normal- og applicative-order reduktion.

Normal-order reduktion: Den *yderste* reduktion mest til venstre først.
Applicative-order reduktion: Den *inderste* reduktion mest til venstre først.



Konsekvenser af *Church-Rosser sætningerne*:

- Et lambda-udtryk kan ikke reduceres til to forskellige normal-former (modulo α-konverteringer).
- Hvis lambda-udtrykket *e* kan reduceres i ét eller flere trin til *f*, kan dette også ske udelukkende ved normal-order reduktion (men ikke nødvendigvis ved applicative-order reduktion).

PS4 - Evalueringsrækkefølge

Noter

Det illustreres hvordan et udtryk (et lambdaudtryk i Scheme) reduceres ved hhv. applicative-order og normal-order.

Ved normal-order reduktion anvendes som det første beta-reduktionen på hele kaldet. Bemærk hvordan (fak 5) substitueres ind to forskellige steder, og derfor kan blive beregnet to gange (nødvendigvis med samme resultat til følge).

Ved applicative-order reduktion i Scheme beregnes de tre udtryk

(lambda(x y) (+ (* x x) (* y y))),

(fak 5) og

(fib 10).

Resultatet af det første udtryk er et *funktionsobjekt* (også kaldet en *closure*), som antydet i grenen med applicative-order evaluering. Vi minder om, at et funktionsobjekt er karateriseret af syntaktiske bestanddele og et environment. Resultaterne af de sidste udtryk er to tal.

Church-Rosser sætningerne udtykker nogle fundamentale konsistensresultater omkring konvertering og reduktion. Se Hudaks artikel for detaljer. Her gives kun nogle, vigtige afledte konsekvenser.

Vi ser, at hvis både applicative-order og normal-order reduktioner giver et veldefineret resultat på normal-form, er resultatet ens. I visse situationer (som på den motiverende slide) vil det dog kun være normal-order resuktionerne der vil give et veldefineret resultat. Applicative-order reduktioner vil aldrig terminere i disse tilfælde.

Givet et udtryk E. Hvis E har en værdi (kan reduceres til normal form), kan dette altid ske via normal-order reduktion. Det vil med forskellige evalueringsrækkefølger kunne ske, at der under beregningen af E både kan forekomme en fejl, en uendelig beregning, og en værdi. Ifølge den første af Church-Rosser sætningerne vil vi dog aldrig kunne reducere udtrykket til to forskellige normal former (modulo α–konverteringer).

Evalueringsrækkefølge og parameteroverførsel.

Algol call-by-value	*Algol call-by-name*
integer procedure f(x); **value** x; **integer** x **begin** f:= 1 **end**;	**integer procedure** f(x); **integer** x **begin** f:= 1 **end**;
f(1/0)	f(1/0)
fejl	*1*
Sædvanlig call-by-value konvention.	En call-by-name parameter substitueres tekstuelt ind i kroppen, (med skyldig hensyntagen til uheldige navne-fejlbindinger).

Reduktions rækkefølge	**Funktions semantik**	**Parameteroverførsels mekanisme**
applicative-order	strict	value
normal-order	lenient	name

PS4 - Evalueringsrækkefølge

Noter

Her illustreres den tætte sammenhæng med call-by-name parametermekanismen, lenient funktions semantik, og normal-order reduktion.

Betingede udtryk og sekventielle boolske operatorer.

Der findes sproglige konstruktioner hvor det er meningsløst eller uhensigtsmæssigt at anvende applicative-order reduktion.

(if b x y)

Afhængig af værdien af b beregnes og returneres enten x eller y

```
(define (fak n)
  (if (= n 0) 1 (* n (fak (- n 1)))))
```

Det ville være meningsløst at beregne fak(-1), i tilfældet n=0

(and x y z)

and beregner sine parameter-udtryk fra venstre mod højre, indtil et af disse bliver *false*, eller indtil alle viser sig at være *true*.

```
> (let ((x 5)
        (y 0))
    (and (not (= y 0))
         (odd (divide x y))))
#f
```

Det ville være unødvendigt og skadeligt at beregne (divide x y)

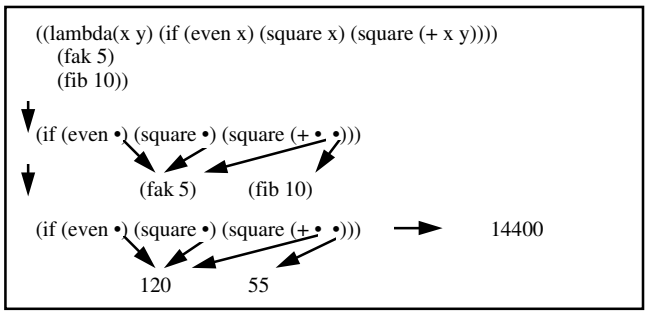
Noter

Det velkendte selektive udtryk, altså if, har et iboende element af normal-order reduktion. Årsagen er naturligvis at vi kun ønsker x evalueret hvis b er sand (se ovenfor), og kun y evalueret hvis b er falsk. Meget ofte vil det være meningsløst at evaluering y hvis b er sand, og vise versa.

Sekventielle boolske operatorer (ala *and then*, og *or else* fra Eiffel og en del andre sprog) er også eksempler på islæt af normal-order reducering. Funktionen *and* fra Scheme (og andre tilsvarende) er pr. definition i Scheme rapporten "sekventielle" (og er dermed ikke normale funktioner, men derimod at betragte som "syntaks").

Lazy evaluering (call by need).

- Ved *lazy evaluering* forståes en implementation af normal-order reducering, som undgår gentagen beregning af det samme udtryk.
- En del moderne funktions-orienterede sprog anvender lazy evaluering.



Praktisk fordel ved lazy evaluering:

•Større effektivitet: Et udtryk beregnes højst én gang.

PS4 - Evalueringsrækkefølge

Noter

På denne slide illustreres, at man ønsker at undgå genevalueringer af det samme udtryk under normal-order reduktion.

Den større effektivitet opnåes netop som følge af dette; dog kun hvis besparelsen i køretid som følge af dette opvejer den administration på køretidspunktet, der skal til for at realisere *"graf reduktionen"*.

Scheme primitiver til forsinket evaluering.

- Scheme anvender applicative-order reduktion.
- Scheme forsyner os med eksplicitte mekanismer til hhv. forsinkelse og gennemtvingelse af en beregning.

(delay *expr*) => *løfte*

(force *løfte*) => *indfrielse af løfte*

Simpel Scheme implementation af delay og force:

(delay expr) ~ (lambda () expr)

(define (force promise) (promise))

Noter

Denne slide viser en meget simpel implementering af delay og force, som de findes i Scheme. Vi vender til sidst i dette kapitel tilbage til disse, for at opnå fordelene ved lazy evaluering.

Delay kan selvsagt ikke implementeres som funktionen

(define (delay expr) (lambda () expr)).

Problemet er, at expr i kaldet (delay expr) ville blive evalueret -- stik imod intensionen.

Hvis vi forcer en værdi, som ikke er et løfte (promise) kan vi få to forskellige resultater, afhængige af implementationen: fejl eller blot værdien.

Hvis vi udfører en "almindelig" operation på et løfte, ala (+ 1 (delay (- 6 1))), kan vi, igen afhængig af implementationen af Scheme, få to forskellige resultater: fejl eller 6, sidstenævnte via *implicit forcing* af løftet.

Streambegrebet i Scheme.

En *stream* er en liste, hvor "halen" af listen kan være delayed.

Streams kan opfattes som potentielt uendelig lange lister.

Stream primitiver:

(cons-stream a b) ~ (cons a (delay b))

(define head car)

(define (tail stream) (force (cdr stream)))

(define empty-stream? null?)

(define the-empty-stream '())

Noter

Streams, eller på dansk strømme, svarer begrebsmæssigt til listebegrebet, dog med den vigtige udvidelse, at en strøm på "hale pladsen" (cdr pladsen i en cons-celle) indeholder et *løfte* (promise) om, at listehalen kan beregnes, hvis eller når der er behov for den. Beregningen af listehalen er altså forsinket.

De strømme vi arbejder med overholder altså invarianten:

> En strøm repræsenteres som en cons-celle, hvis car-plads refererer til strømmens hoved, og hvis cdr-plads refererer til et løfte.

Der er andre muligheder, såsom at en strøm som helhed blot repræsenteres som et løfte, eller større dele af strømmen kan være tilstede på ren listeform (flere cons-celler).

Bemærk at vi ikke kan implementere cons-stream som en funktion. Årsagen er naturligvis, at hvis cons-stream var en funktion ville vi anvende applicative-order reduktion, og dermed ikke opnå forsinkelsen af halen, som vi er ude efter. På denne måde svarer cons-stream til selve delay primitivet. Cons-stream er en *syntaktisk abstraktion*, som kan realiseres med det man i Lisp kaldes *makroer*.

Både head og tail kan implementeres som funktioner. Head er faktisk blot et alias for car.

For fuldstændighedens skyld giver vi også funktioner der returnerer den tomme strøm, og som tester for, hvorvidt en strøm er tom.

Strømmen af 1-taller og naturlige tal.

En uendelig lang strøm af 1-taller:

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

```
> ones
(1 . #<Promise>)
> (head ones)
1
> (tail ones)
(1 . #<Promise>)
```

En uendelig lang strøm af de naturlige tal:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define nat-nums (integers-starting-from 1))
```

```
> nat-nums
(1 . #<Promise>)
> (head (tail (tail nat-nums)))
3
> (stream-section 7 nat-nums)
(1 2 3 4 5 6 7)
```

En uendelig lang strøm af de naturlige tal, hvori 7 ikke går op:

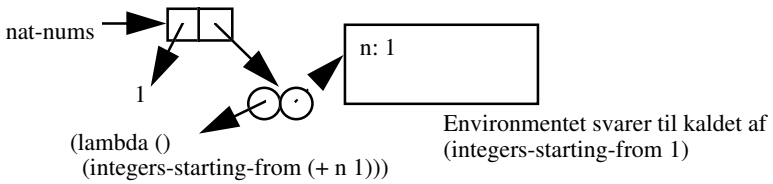
```
(define no-sevens
  (filter (lambda (x) (not (divisible? x 7)))
          nat-nums))
```

```
> (stream-section 25 no-sevens)
(1 2 3 4 5 6 8 9 10 11 12 13 15
16 17 18 19 20 22 23 24 25 26
27 29)
```

PS4 - Evalueringsrækkefølge

Noter

Nedenstående viser et billede af strømmen af de naturlige tal, som defineret af nat-nums. Man ser at der er netop én cons-celle, hvor car-positionen er det første naturlige tal, og crd-positionen er et løfte om at kunne beregne resten, hvis eller når der bliver behov for det.



Funktionen stream-section ovenfor konverterer et afsnit af en strøm til en liste ved at "force" og "conse" strømmens elementer:

```
(define (stream-section n s)
    (if (= n 0)
        '()
        (cons (head s) (stream-section (- n 1) (tail s)))))
```

I strømmen no-sevens anvendes funktionen filter, som svarer nøje til den filter operation, vi studerede i kapitlet "højereordens funktioner". Desværre skal vi dog udskifte liste primitiver med stream primitiver:

```
(define (filter pred stream)
  (cond ((empty-stream? stream) the-empty-stream)
        ((pred (head stream))
          (cons-stream (head stream)
                       (filter pred (tail stream))))
        (else (filter pred (tail stream)))))
```

Strømmen af naturlige tal og Fibonacci-tal.

Alternativ og "mere elegant" formulering af nat-nums:

```
(define nat-nums
  (cons-stream
     1
     (add-streams ones nat-nums)))
```

ones:	**1** 1 1
nat-nums:	**1** 2 3 ...
(add-stream ones nat-nums):	2 3 4

Strømmen af fibonacci tal:

```
(define fibs
  (cons-stream 0
       (cons-stream 1
            (add-streams (tail fibs) fibs))))
```

fibs:	**0** **1** 1 2 3
(tail fibs):	**1** 1 2 3
(add-stream (tail fibs) fibs):	1 2 3 5

PS4 - Evalueringsrækkefølge

Noter

I ovenstående definitioner af nat-nums og fibs anvendes en funktion, som adderer to strømme. Denne funktion kan implementeres således:

```
(define (add-streams s1 s2)
  (cond
      ((empty-stream? s1) s2)
      ((empty-stream? s2) s1)
      (else
       (cons-stream
          (+ (head s1) (head s2))
             (add-streams (tail s1) (tail s2))))))
```

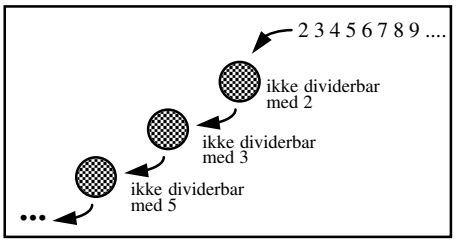
Man kan naturligvis let lave en højereordens stream-funktion, der tager en binær operation *op* som parameter, og som dermed generaliserer add-stream til *op*-stream.

Strømmen af primtal: Eratosthenes' si.

Strømmen af primtal beregnet gennem "Eratosthenes' si":

```
(define (sieve stream)
  (cons-stream
    (head stream)
    (sieve (filter
             (lambda (x) (not (divisible? x (head stream))))
             (tail stream)))))

(define primes (sieve (integers-starting-from 2)))
```




```
> (stream-section 25 primes)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43
47 53 59 61 67 71 73 79 83 89 97)
```

PS4 - Evalueringsrækkefølge

Noter

Ideen i Eratosthenes' si er følgende:

Antag at vi har en liste af de første n primtal: p1 p2 ... pn.

Vi ønsker nu at finde det n+1'te primtal.

Startende med pn + 1, eller mere præcist med strømmen (integers-from (+ pn 1)) , overstreg (filtrer fra) alle multipla af p1, p2, ... pn. Det første tal i den resulterende strøm er det n+1'te primtal.

Om den konkrete virkemåde af funktionen sieve ovenfor kan man observere:

- sieve kalder sig selv rekursivt
- et kald af sieve, som umiddelbart giver primtallet p på hoved-pladsen fra-filtrerer alle multipla af p i halen, som resulterer fra kaldet.
- et kald af sieve er kaldsmæssigt indlejret i andre kald af sieve, som fra-filtrerer multipla af mindre primtal.

Forfinede Scheme primitiver til forsinket evaluering.

Problem med de simple primitiver delay og force:
Gentagen 'forcing' af samme *promise* giver samme værdi, men forårsager et betydeligt overhead, da beregningen i forbindelse med forcing gennemføres igen og igen.

```
(delay expr)  ~ (make-promise (lambda () expr))

(define (force promise) (promise))

(define (make-promise proc)
 (let ((already-run? #f) (result nil))
   (lambda ()
     (if (not already-run?)
         (sequence (set! result (proc))
                   (set! already-run? #t)
                   result)
         result))))
```

Memoisering af proc

Procedure, som har en privat tilstand.

Anvendelse af **imperative teknikker** til implementation af eksplicit lazy evaluering.

Ligheder mellem dette og simuleringen af klasser.

PS4 - Evalueringsrækkefølge

Noter

Dette stammer fra Abelson og Sussmans bemærkelsesværdige bog: *"Structure and Interpretation of Computer Programs"*, side 264, modulo nyt navn for memo-proc.

Scheme rapporten (version 4) er lidt mere kompleks hvad angår make-promise.

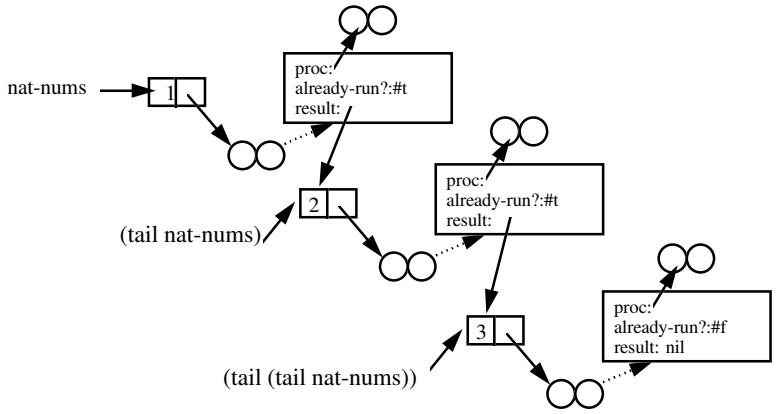
Bemærk iøvrigt, at vi i kroppen af make-promise anvender imperativ stil idet tilstanden af de lokale variable already-run? og result ændres (med Scheme assignmentet set!) når løftet indfries første gang.

Hvis vi ikke programmerer i funktionel stil, kan delayed evaluation mixet med assignment give meget uigennemskuelige programmer.

Nat-nums med memoisering.

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define nat-nums (integers-starting-from 1))
```



PS4 - Evalueringsrækkefølge

Noter

Vi antager her, at vi arbejder med den forfinende, memoriserede delay, fra forrige slide.

Diagrammet på denne side viser de bagved-værende strukturer af hhv. nat-nums, (tail nat-nums) og (tail (tail nat-nums)) ved anvendelse af memoisering.

Environmentet, hvoraf der vises tre inkarnationer, stammer fra kald af proceduren make-promise.

Streams i forhold til imperativ programmering.

- Streams (og lister) er primært velegnede til løsning af problemer, som er orienterede mod "input" og "output".
- I stedet for at foretage mutation af datastrukturer, returneres nye, totale kopier af datastrukturerne.
- Streams er bedre end lister til input og output, idet der kan produceres output uden at al input skal være til stede.

Eksempel:

Modellering af transaktioner på en bankkonto:

```
(define my-account                    transactions: 4 6 8 10  -8 -12 -16 -20
  (add-streams                                      16 24 32 40  -32 -48 -64 -80
   (cons-stream 100 my-account)                     ...
   transactions))
```

Kan i princippet genereres interaktivt.

```
> (stream-section 16 my-account)
(104 110 118 128 120 108 92 72 88 112 144 184 152 104 40 -40)
```

PS4 - Evalueringsrækkefølge

Noter

Opgave: Transaktionerne i ovenstående eksempel er genereret med udgangspunkt i en liste (i eksemplet af længde 4), som gentages om og om igen. Før hver gentagelse mappes den oprindelige liste med en funktion, som fordobler og skifter fortegn på tallene. Skriv funktionen

```
(define (cycle lst alternating) ...)
```

der givet listen lst, og funktionen alternating, producerer en uendelig strøm efter det beskrevne princip.

I forlængelse af ovenstående eksempel kunne man let lade hver transaktion bestå af et transaktionsnavn (eller alternativt en transaktionsfunktion) og nogle transaktionsdata. Hver transaktion skal så *udføres* på det foregående beløb på kontoen, og ligesom ovenfor, skulle resultatet være strømmen af beløb (eller evt. en mere sammensat struktur).