

Introduktion til Funktionsorienteret Programmering.

Sammenligning med det imperative paradigme
Assignment kontra navnebinding.
Iterative kontra rekursion.
Introduktion til højereordensfunktioner.
Evalueringsrækkefølge.
Beregning ved substitution.
Funktioner som klasser.

PS3 - Funktionsorienteret Programmering

© Kurt Nørmark, Aalborg Universitet

9/25/96

s. 1

Noter

I denne lektion introducerer vi funktionsorienteret programmering. Vi forsøger at motivere stilarnten ved at sætte den op som en kontrast til den mere velkendte imperative stilarnt. Og vi vil studere nogle fundamentale egenskaber ved funktionsorienterede programmer.

Vi eksemplificerer primært ved brug af Lisp dialekten Scheme - eller i pseudosprog.

Funktionsorienteret kontra imperativ programmering.

- **Funktionsorienteret Programmering:**
 - Funktioner er abstraktioner over udtryk.
 - Applikativ programmering.
 - En programmeringsstil som bruger én fundamental syntaktisk udtryksmåde: anvendelse (applikation) af *funktioner* på argumenter.
 - Navne kan bindes til værdier, men eksisterende navnebindinger's værdier kan ikke ændres.
 - Definition og anvendelse af *højereordens funktioner*.
 - Funktioner er fulgyldige værdier, på lige fod andre mere simple værdier.
- **Kontrast: Imperativ Programmering.**
 - Procedurel programmering: Procedurer er abstraktioner over kommandoer.
 - En programmeringsstil som gør brug af kommandoer i bestemte kontrolmønstre.
 - Modellerer et system, hvis tilstand ændres inkrementelt som en funktion af tiden.
 - En kommando har en målelig effekt på sine omgivelser.
 - Ærke-kommandoen: assignment

PS3 - Funktionsorienteret Programmering

Noter

Man kan vælge at forstå skridtet fra imperativ programmering til funktionsorienteret programmering på samme måde som skridtet fra (ustruktureret) goto-programmering til struktureret programmering. I begge tilfælde går vi fra en relativ rodet verden til en meget "renere verden".

Med funktionsorienteret programmering bliver det væsentligt lettere at ræsonnere omkring programmet, med henblik på sikring af korrekthed i forhold til en specifikation.

En væsentlig forandring fra imperativ programmering til funktions-orienteret programmering er, alt andet lige, at vi "dropper" assignmentet i funktionsorienteret programmering. Mere generelt afskærer vi os fra at ændre på programmets tilstand (værdierne bundet til de eksisterende navne) som funktion af tiden. Vi afskærer os også fra at lave andre sideeffekter, så som "print" operationer.

Man kan naturligvis argumentere for, at væsentlige applikationsområder kun dårligt lader sig understøtte af programmer, der skal leve med disse begrænsninger. Men der er en overraskende stor mængde af applikationsområder, der meget fint kan understøttes af funktionsorienterede programmer. Og der er ofte væsentlige dele af imperative programmer, som lader sig behandle funktionsorienteret.

Hvorfor betegnelsen højereordens funktioner?

0-te orden: data eller konstante funktioner.

1-orden: funktioner som tager 0-te ordens funktioner som parameter, og returnerer sådanne som resultat.

2-orden (højere orden): funktioner som tager 1-ordens funktioner som parameter, og returnerer sådanne som resultat.

Definition og anvendelse af funktioner i Scheme

• Funktionsobjekter: *Closures*.

`(lambda (parametre) body)`

- Returnerer et funktionsobjekt.
- Lukker af over (indfanger) navnene i funktionsobjektets omgivelser:
 - Statisk navnebinding.
 - Giver en closure
 - Environment + syntaktisk form.

• Definitioner:

`(define name expression)`

Formen af funktionsdefinition:

`(define name (lambda (parameters) body)) ~`

`(define (name parameters) body)`

- Globale definitioner: Binder navne i det globale environment.

• Funktionsanvendelse (-kald):

`(e1 e2 ... en)`

- Alle udtryk i kaldet evalueres på en uniform måde.
- Værdien af e₁ skal være et funktionsobjekt.
- Værdierne af e₂ ... e_n kan være vilkårlige objekter.

PS3 - Funktionsorienteret Programmering

Noter

Denne slide studerede vi allerede i forelæsningen om sprog i Lisp kulturen. Men da funktionsbegrebet selvsagt er vigtigt i denne lektion repeterer vi den her.

Bemærk *closure begrebet*, som jo altså dækker over en funktion opfattet som et objekt. Dette objekt består dels af funktionens syntaktiske form, og de omgivelser (det environment = samling af navnebindinger), hvori funktionen er defineret.

Nederst til højre vises to forskellige former for funktionsdefinition:

(1) `(define name (lambda (parameters) body))`

og

(2) `(define (name parameters) body)`

Det er meget væsentligt at forstå at (1) er den oprindelige form, som er i overensstemmelse med den generelle form af define

`(define name expression)`

hvor 'expression' evalueres til en værdi, der bindes til navnet 'name'. Formen (2) er syntaktisk sukker for (1), og den har følgende to fordele:

Kortere og mindre indlejret.

Bestanddelen (name parameters) svarer til kaldsformen af funktionen.

Derfor bliver (2) næsten altid benyttet. Men vær sikker på at forstå, at hver gang I ser formen (2) er det blot en overfladisk omskrivning af formen (1). Scheme-systemet er fuldstændig blind for, om man bruger (1) eller (2). Så snart den ser en definition på formen (2) omformer systemet det internt til en definition af formen (1).

Navnebinding i forhold til assignment.

```
solve(a,b,c: real): root-list is
-- returner løsninger af ligningen  $ax^2+bx + c = 0$ 
local determinant: real
do
determinant := b * b - 4 * a * c;
if determinant > 0
then result := to løsninger
elseif determinant = 0
then result := én løsning
else result := ingen løsning
end
```

Misbrug af
assignment.

```
solve(a,b,c: real): root-list is
do
let determinant be b * b - 4 * a * c
in if determinant > 0
then return to løsninger
elseif determinat = 0
then return én løsninge
else return ingen løsning end
end
end
```

PS3 - Funktionsorienteret Programmering

© Kurt Nørmark, Aalborg Universitet

9/25/96

s. 4

Noter

Dette eksempel viser et eksempel på et imperativt program (skrevet i et sprog, som ligner Eiffel), nemlig løsning af en 2. gradsligning. Vi sætter fokus på assignment af variabelen 'determinant' til den velkendte værdi

$$b * b - 4 * a * c.$$

I langt de fleste programmer, som vi skriver, har vi behov for at tilskrive variable værdier ala 'determinant' af hensyn til *klarhed* i udtryksform. Der kan også være en *effektivitetsmæssig årsag*, nemlig det at undgå genberegning af det samme udtryk flere gange.

Vores pointe er her, at dette er en meget speciel anvendelse af et assignment. Det specielle består i, at vi aldrig laver værdien af variabelen om igen. Det forhold, at variabelens værdi aldrig ændres efter første tilskrivning retfærdiggør en anden udtryksform: Vi går fra brug af assignment til navnebinding. Dette er emnet på næste slide.

Navnebinding i Scheme

Med syntaktisk sukker

```
(define (solve a b c)
  (let ((det (- (* b b) (* 4 a c))))
    (cond ((> det 0) (list (/ (- (- b) (sqrt det)) (* 2 a))
                          (/ (+ (- b) (sqrt det)) (* 2 a))))
          ((= det 0) (list (/ (- b) (* 2 a))))
          (else '() )))))
```

Uden syntaktisk sukker

```
(define (solve a b c)
  ((lambda (det)
    (cond ((> det 0) (list (/ (- (- b) (sqrt det)) (* 2 a))
                          (/ (+ (- b) (sqrt det)) (* 2 a))))
          ((= det 0) (list (/ (- b) (* 2 a))))
          (else '() ))))
    (- (* b b) (* 4 a c))))
```

Let realiserer *simultan* navnebinding. Der findes variation af let til *sekventiel* navnebinding og *gensidig rekursiv* navnebinding.

PS3 - Funktionsorienteret Programmering

Noter

I eksemplet løses en andengradsligning i Scheme, i forlængelse af det samme eksempel fra forrige slide.

Ovenstående illustrerer også **cond** og **list** funktionerne. Cond er et eksempel på et *betinget udtryk*, altså et udtryk der udvælges og beregnes på baggrund af beregning af én eller flere boolske udtryk. List returnerer en liste med de elementer, som sendes med som parametre.

I funktionen forinden ses hvordan man kan anvende en lokal, anonym funktion til navnebindingsformål. Ideen er at funktionens formelle parameter er navnet, som bindes ved funktionskaldet. Med denne løsning har vi gjort navnebinding mulig uden at introducere nye konstruktioner i sproget. Men løsningen er ikke attraktiv. Den er "kluntet" i almindelighed, og der er i særdeleshed for stor afstand mellem introduktionen af navnet (her den formelle parameter i den anonyme funktion) og værdien (det aktuelle parameterudtryk tilsidst i solve).

Hvis mere end ét navn bindes i en let konstruktion bindes navnene simultant. Dette medfører, at de sidste navnebindinger ikke kan anvende de første navnebindinger. (Overbevis dig om at vi har simultan navnebinding i let ved at studere den nederste version af solve nøje). Da det imidlertid er praktisk at kunne anvende de netop bundne navne i efterfølgende højresider af navnebindinger, findes der en variant af let, som kaldes let*, der netop binder navnene sekventielt. Let* kan laves ud af let ved at indlejre let's i hinanden. Hvis man har behov for at navngive to lokale funktioner i en let, og hvis disse to funktioner kalder hinanden rekursivt, kan vi hverken bruge let eller let*. Derfor findes endnu en variant som kaldes letrec.

Om brugen af typer.

- Svag typning (*weak typing*).
 - Typefejl kan føre til fejl-beregninger.
- Stærk typning (*strong typing*).
 - En anvendelse af typer der sikrer, at typefejl ikke fører til fejl-beregninger.
 - Typecheck kan forekomme på udførelsestidspunktet.
- Statisk typning (*static typing*).
 - Typer af udtryk kan bestemmes før programudførelse.
 - Typecheck udføres før programudførelse (f.eks. under oversættelsen).
 - *EksPLICIT typedekoration* eller *implicit typeinferens*.
 - Statisk typning medfører stærk typning.

PS3 - Funktionsorienteret Programmering

Noter

Denne slide viser forskellige tilgange til typer i programmeringssprog.

Svag typning forekommer i lavniveau sprog, som med eller mod intensionen kan operere på data af alle typer. Dette forekommer på bitniveau.

I typiske dynamiske programmeringssprog, som omtalt på en tidligere slide, er der hverken eksplicit type dekoration (ala Pascal) eller implicit typeinferens (ala ML). I dynamiske programmeringssprog har vi altså ikke statisk typning. Man kunne være fristet til at sige, at disse programmeringssprog har 'dynamisk typning', men ifølge ovenstående klassificering af typning, har vi faktisk ikke behov for en sådan betegnelse. Det forholder sig nemlig sådan, at dynamiske programmeringssprog sikrer ved typecheck på udførelsestidspunktet, at typefejl ikke fører til fejlberegninger. Der er altså ifølge ovenstående definitioner stærk typning i de fleste dynamiske programmeringssprog.

Iteration, rekursion og mapping.

- Sædvanlige *iterative kontrolstrukturer* passer dårligt til funktions-orienteret programmering.
- Iteration kan generelt set udtrykkes ved anvendelse af *rekursive funktioner*.
- Iteration kan udtrykkes effektivt ved *hale-rekursive funktioner*.
- Iterative gennemløb o.l. af forskellige datastrukturer kan defineres ved at *abstrahere over bestemte rekursive mønstre* af funktionskald.

PS3 - Funktionsorienteret Programmering

Noter

Sædvanlige kontrolstrukturer sekventierer nogle mere primitive kommandoer (også ofte kaldet sætninger), som har en effekt på beregningens tilstand. Man kan lidt plat sige, at kontrolstrukturerne blot bestemmer hvordan vi skal blande assignments og andre kommandoer, der har en umiddelbar effekt på programtilstanden. Ud fra denne betragtning er det klart, at vi ikke har brug for sådanne kontrolstrukturer i funktionsorienterede programmer.

På denne slide diskuterer vi iteration. Der er naturligvis behov for gentagelser i en eller anden forstand i funktionsorienterede sprog. Men iterationen foregår altid på data, og det resulterer i en værdi, der er resultatet af at gentage en beregning på disse data. Som det skulle være de fleste bekendt allerede, kan rekursive funktioner bruges til noget sådant.

Rekursion er forbundet med et lagermæssigt overhead, idet vi skal bruge plads på at repræsentere rekursionsudviklingen indtil vi når til basistilfældet. Der findes imidlertid en form for rekursion, som eliminerer dette effektivt. Dette vil vi stifte bekendtskab med på én af de næmeste slides.

Når vi indfører højereordens funktioner kan vi indfange "rekursive mønstre", og abstrahere over disse. Herved kan vi opbygge et bibliotek af iterations-primitiver på forskellige former for datastrukturer, som er specialdesignede til gentagne beregninger på disse strukturer. Dette er bemærkelsesværdigt, og vi vil selvfølgelig vende tilbage til dette i de kommende kapitler.

Iteration og rekursion.

En funktion, som gennemløber en liste af tal, og som returnerer en liste af samme længde af fordoblede tal.

Imperativ løsning

```
double(lst: list[integer]): list[integer] is
do
  from result.create; lst.goto_beginning
  until lst.at_end
  do
    result.insert_after(lst.retrieve_after * 2);
    result.advance; lst.advance
  end;
end;
```

Rekursiv løsning

```
double(lst :list[integer]): list[integer]
if empty(lst)
then empty-list
else concat (head(lst) * 2,
            double(tail (lst)))
```

Rekursiv løsning i Scheme

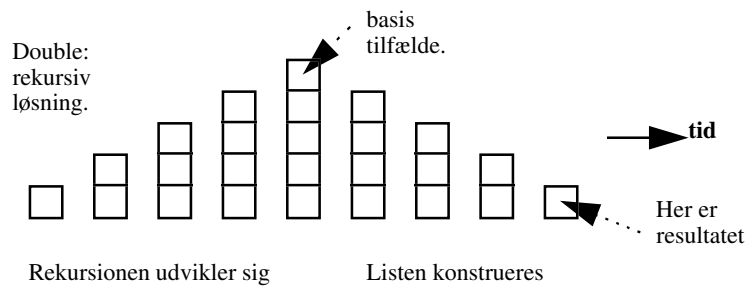
```
(define (double lst)
  (if (null? lst)
      '()
      (cons (* 2 (car lst))
            (double (cdr lst)))))
```

PS3 - Funktionsorienteret Programmering

Noter

Vi viser her hvordan et kald af funktionen double udvikler sig.

Stakudvikling:



Kald af rekursive funktioner kræver en lagermængde, der er proportional med højden af rekursionsstakken. I ovenstående tilfælde betyder dette, at højden af rekursionsstakken på et tidspunkt bliver proportional med længden af listen, som vi vil fordoble. Dette kan være et problem, hvis der ikke er sat tilstrækkeligt meget lager af til køretidsstak.

Iteration, halerekursion og mapning.

Halerekursiv løsning

```
double(lst: list[integer]): list[integer]
double-help(lst, empty-list)

double-help(lst, res: list[integer]): list[integer]
if empty(lst)
then res
else double-help(tail(lst),
                 concat-end(res, 2 * head(lst)))
```

Løsning via mapning

```
double(lst: list[integer]): list[integer]
map(*2, lst)

map(f: integer -> integer; lst: list[integer]):
list[integer]

if empty(lst)
then empty-list
else concat (f(head(lst)),
            map(f, tail(lst)))
```

Halerekursiv løsning i Scheme

```
(define (double lst)
  (reverse
   (double-help lst '())))

(define (double-help lst res)
  (if (null? lst)
      res
      (double-help
       (cdr lst)
       (cons
        (* 2 (car lst))
        res))))
```

Løsning via mapning i Scheme

```
(define (double lst)
  (map
   (lambda (x) (* 2 x))
   lst))
```

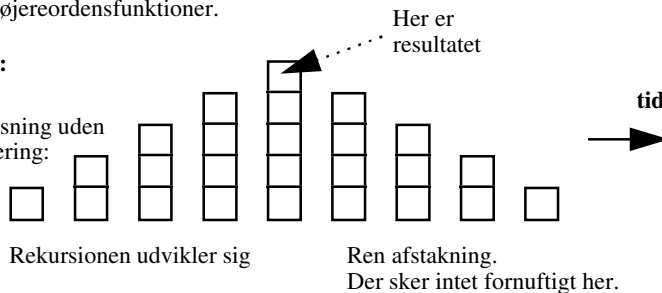
PS3 - Funktionsorienteret Programmering

Noter

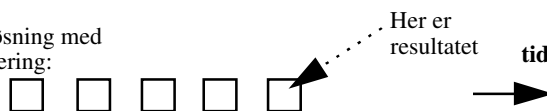
På denne slide er map en abstraktion over "at anvende en funktion på alle elementer i en liste og returnere listen af resultater af disse anvendelser". Vi vender tilbage til map i lektionen om højereordensfunktioner.

Stakudvikling:

Double:
halerekursiv løsning uden
lageroptimalisering:



Double:
halerekursiv løsning med
lageroptimalisering:



Den halerekursive løsning vist på listen (1 2 3 4):

lst	res
(1 2 3 4)	()
(2 3 4)	(2)
(3 4)	(2 4)
(4)	(2 4 6)
()	(2 4 6 8)

En funktion er halerekursiv hvis det rekursive kald direkte er resultat af funktionen. I den halerekursive løsning er det kun nødvendigt med én aktiveringrekord, som jo har lst og res som felter. Tilstanden af iterationen er helt indeholdt i disse to parametre, som derfor kan overskrives (re-assignes) undervejs i iterationen. Dette er en optimalisering, lavet af udførelsessystemet! Begrebsmæssigt bør programmøren tænke på iterationen som forløbende øverst, dog velvidende at det er meget mere lagerøkonomisk.

I Scheme versionen af den halerekursive funktion vender reverse listen om inden returnering. Derved slipper vi løbende for at "indsætte" elementer i halen af listen.

Eksempel på højereordens funktion.

- **Idé:** Vi ønsker at programmere en funktion, som returnerer en approximation af den afledte funktion som resultat.
- **Højereordens funktion:**
 - Funktion som parameter.
 - Funktion som resultat.

```
Function derive (f: real -> real; dx: real): [real -> real] is
do
  return function(x: real): real is
    do
      return ( f(x + dx) - f(x) ) / dx
    end
end
```

- Store udfordringer til de fleste imperative sprog, på trods af, at disse understøtter funktionsbegrebet.
- Muligt i funktionsorienterede sprog.

Signatur for derive:

$(\mathbb{R} \rightarrow \mathbb{R}) \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$

PS3 - Funktionsorienteret Programmering

Noter

I ovenstående eksempel refererer den indre funktion, der returneres som resultat, til parameteren f af den ydre funktion. Bliver denne binding af f til en aktuel parameter værdi mon brudt, når vi f.eks. anvender `deriv` således

```
let g be derive(sinus, epsilon)
in
  g(2.7)
end
```

Spørgsmålet kan omformuleres til “er det statisk eller dynamisk navnebinding vi ønsker”. Svaret er som næsten altid: statisk navnebinding. Af hensyn til sikkerhed og gennemskuelighed skal navne bindes i definitionens omegn, og ikke i anvendelsens omegn. Herved opstår der imidlertid et problem ved funktionsreturnering, som vi ikke er vant til. Vi kan ikke blot glemme alt om aktiveringen af funktionen `deriv`, idet resultatet af denne beregning stadig holder fast i (er afhængig) af de navnebindinger, der blev etableret i forbindelse med kaldet af `derive` (her parametrene).

Eksempel på højereordens funktion i Scheme.

```
(define (derive f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x))
        dx)))
```

```
1> (let ((res (derive (lambda (x) (* x x x)) .0001)))
      (map res (list 1 2 3 4 5)))
(3.0003000099987354 12.000600010022566 27.00090001006572
48.00120000993502 75.00150000993244)

2> (let ((f-list (map derive (list (lambda(x) (* x x)) sin log)
                              (list 0.001 0.001 0.001))))
      (ny-map f-list (list 2 4 6 8)))
((4.000999999999699 8.00100000000037 ...)
 (-0.416601415864859 -0.6532651107071796 ...)
 (0.49987504165105445 0.24996875520755246 ...))
```

Funktioner i Scheme er af *første klasse* fordi:

1. De kan overføres som parametre.
2. De kan returneres som resultat.
3. De kan lagres i datastrukturer.

PS3 - Funktionsorienteret Programmering

Noter

Vi viser her `derive` i Scheme. Generelt gælder der for alle typer af data, at disse er af 1. klasse hvis ovenstående tre krav (1—3) er opfyldt. Det er ikke overraskende at f.eks. tal er af første klasse. Men det er sikkert nyt for mange studerende, at funktioner er af 1. klasse.

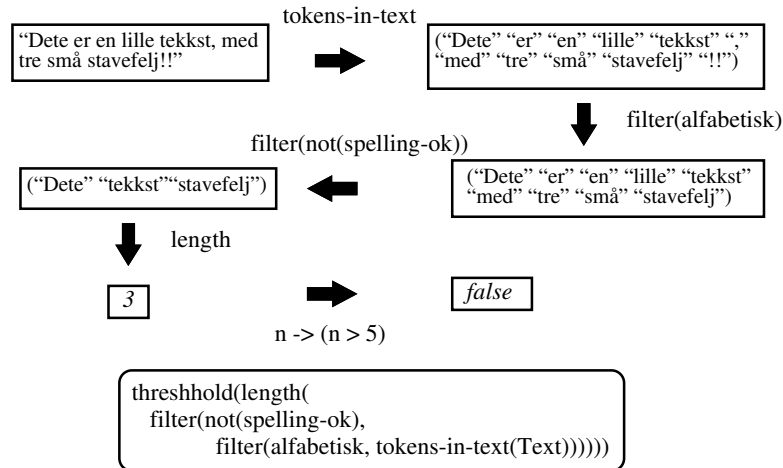
Vi viser også to eksempler på anvendelse af `derive`. Eksempel 2 er mest kompliceret. Læg her mærke til, at vi kan lave en liste af funktioner, som vi mapper `derive` henover. Dette kræver - måske lidt kunstigt - en liste af småtal af tilsvarende længde som listen af funktioner, der ønskes afledt. Vi ser altså, at `map` kan tage mere end to parametre. `F-list` bliver en liste af afledte funktioner. Vi prøver disse af ved at anvende dem på en række tal. Denne anvendelse af en liste af funktioner på en liste af tal kræver en ny slags `map`, som vi programmerer til lejligheden:

```
(define (ny-map function-list value-list)
  (if (null? function-list)
      '()
      (cons (map (car function-list) value-list)
            (ny-map (cdr function-list) value-list))))
```

Eksempel på funktionsorienteret programmering.

En funktions-orienteret løsning på en realistisk opgave.

Opgave: Givet en tekst(streng) med danske ord, find ud af om der er mere end 5 stavefejl i teksten.



PS3 - Funktionsorienteret Programmering

Noter

For at ovenstående skal virke, skal der naturligvis være defineret passende funktioner. Nogle af funktionerne er meget specifikke, mens de andre er generelt anvendelige.

Et ord om funktionen `not(spelling-ok)`. Vi antager, at `spelling-ok` er en funktion:

`spelling-ok: ord -> bool`

som fortæller hvorvidt parameteren er korrekt stavet. Vi har brug for negeringen af denne. Vi kunne opfatte `not(spelling-ok)` som pseudo notation for denne funktion; det er dog mere spændende at opfatte `not` som en funktion, der tager en boolsk funktion som parameter og returnerer en anden boolsk funktion som parameter, negeringen af parameterfunktionen:

`not: (X -> bool) -> (X->bool)`.

Med andre ord, vi opfatter `not` som en højereordens funktion. X vil konkret være typen `String`.

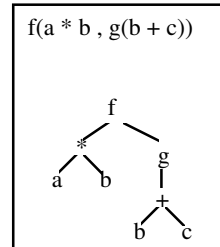
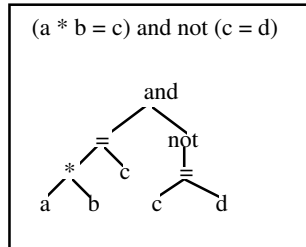
Tilsvarende kan vi opfatte `filter` som en funktion, der tager en boolsk-returerende funktion som parameter, og som selv returnerer en funktion:

`filter: (String -> Boolean) -> (String* -> String*)`

`Filter(not(spelling-ok))` er således en funktion, der givet en liste af strenge returnerer listen af de strenge, der ikke er korrekt stavede. Vi vil studere `filter` i større detalje i kapitlet om højereordens funktioner.

Uafhængighed af evalueringsrækkefølge.

- I et funktionsorienteret program beregnes et *udtryk*, hvilket resulterer i en *værdi*.
- Et udtryk er *hierarkisk struktureret* i *deludtryk*.



- Deludtryk kan beregnes i en vilkårlig rækkefølge såfremt hele udtrykket “har en værdi”.
 - Såfremt der ikke opstår fejl i deludtryk.
 - Såfremt beregningen af alle deludtryk terminerer.
- Deludtryk kan beregnes parallelt.

PS3 - Funktionsorienteret Programmering

Noter

Deludtryk kan beregnes i en vilkårlig rækkefølge: Det er ikke muligt for beregningen af et deludtryk at påvirke beregningen af et deludtryk “i en anden gren” af udtrykket.

I et procedure- eller funktionskald i et “sædvanligt” imperativt programmeringssprog er det ikke god stil at gøre antagelser om, at parametrene beregnes i en bestemt rækkefølge (f.eks. venstre mod højre). Der er nemlig intet til hinder for at funktionen g , i f.eks. Pascal, kunne lave en sideeffekt på variabelen b , således at det ville blive af afgørende betydning, i hvilken rækkefølge de to deltryk af f blev beregnet.

Rækkefølgen af evaluering er et dybere emne end det antydes på denne slide. Kan det f.eks. under nogen omstændigheder være af betydning om vi starter fra toppen eller fra bunden af evalueringstræet. Med andre ord, skal en funktion beregne alle sine parametre inden funktionen kaldes (start fra bunden), eller skal vi beregne parametrene når der bliver brug for dem (start fra toppen). Vi vender tilbage til denne problematik i kapitlet “Evalueringssækkefølge og forsinket evaluering”.

En beregningsmodel baseret på substitution.

Resultatet af et funktionskald kan beregnes ved brug af en meget simpel "substitutionsmodel":

En funktion anvendes på et antal aktuelle parametre ved at beregne dets "krop" hvori alle formelle parametre er substitueret med værdien af de aktuelle parametre (argumenterne).

Eksempel:

$f(5, g(6))$

$f(x, y):$
 $(x + y) * (x - y)$

$g(x):$
 $x \text{ div } 2$

- Det er naturligt (men ikke nødvendigt) at starte indefra og ud: beregn først ved brug af modellen $g(6)$, hvilket giver 3.
- Substituer x med 5, og y med 3 i definitionen af f . Resultat: 16.

PS3 - Funktionsorienteret Programmering

Noter

Vi vil i en senere lektion diskutere forskellige rækkefølger af substitutioner. I denne kommende lektion vil vi se, at vi ikke nødvendigvis skal beregne $g(6)$ før vi beregner $(x+y) * (x-y)$ i kroppen af f .

Substitutionsmodellen er dybest set en konsekvens af *referential transparency*. Givet at de formelle parametre er bundet til værdien af de aktuelle parametre, kan de formelle parametre i kroppen erstattes af argumenterne (eller for den sags skyld af de aktuelle parameterudtryk).

Husk at vi taler om et *argumenter* som er værdien af et *aktuelt parameterudtryk*.

Lambda - Den Ultimate.

Førsteklasses funktioner med statisk navnebinding (closures) er lige så stærke som klasser.

“En klassedefinition” :

```
(define (class init-parametre)
  (let ((var1 value1)
        ...
        (varn valuen))
    (lambda (op)
      (cond ((eqv? op op1) metode for op1 )
            ...
            ((eqv? op opm) metode for opm ))) )
```

Indkapsling af datarepræsentation

Returværdien er en “dispatch funktion” til klassens metoder

“Rå klasseinstantering og operationskald” :

```
(let ((obj (class aktuelle-par)))
  (obj opj))
```

PS3 - Funktionsorienteret Programmering

Noter

På denne slide illustrerer vi, at funktionsbegrebet (i Scheme) har vokset sig så stærkt, at vi umiddelbart kan simulere klasser, objekter, metoder og beskeder.

Vi definerer ovenfor en funktion, som hedder class. Funktionen tager et antal simple parametre, som spiller rollerne af initialværdier af instansvariablene var₁ ... var_n.

Et kald af funktionen class returnerer et funktionsobjekt, som spiller rollen af et objekt. Objektet (dvs funktionen bag objektet) kan kaldes, hvorved vi kan aftvinge det en bestemt metode. Bemærk at denne metode netop er placeret i scope af den indkapslede objekttilstand!

På næste slide viser vi et helt konkret eksempel på ovenstående.

Et eksempel på en klasse.

```
(define (point x y)

  (define (x-cord)
    x)

  (define (y-cord)
    y)

  (define (move dx dy)
    (new-instance point (+ x dx) (+ y dy)))

  (define (self message)
    (cond ((eqv? message 'x-cord) x-cord)
          ((eqv? message 'y-cord) y-cord)
          ((eqv? message 'move) move)
          (else (error "Cannot find method"))))

  self)
```

PS3 - Funktionsorienteret Programmering

Noter

Vi definerer klassen point. Til forskel fra skabelonen på forrige side udnytter vi her "klassens" parametre som instansvariable. Vi kunne uden problemer dublere disse (ved at introducere en let konstruktion med egentlige instansvariable ala skabelonen - men dette ville være redundant).

Bemærk at vi ovenfor definerer navngivne metoder, og en navngiven dispatcher, som vi kalder self. Bemærk også, at det er self der returneres. Dette svarer helt til skabelonen på forrige side.

Vi vælger at programmere klassen point i det funktions-orienterede paradigme. Det ytrer sig ved, at move returnerer et nyt punkt i stedet for at mutere et eksisterende punkt.

New-instance er funktion, som instantierer en klasse. På næste side viser vi new-instance mv.

Nye OO “primitiver”.

```
(define (new-instance class . args)
  (apply class args))

(define (send object message . args)
  (let ((method (method-lookup object message)))
    (if (procedure? method)
        (apply method args)
        (error "Error in method lookup"))))

(define (method-lookup obj mes)
  (obj mes))
```

Eksempler på anvendelse:

```
1> (define p
    (new-instance point 3 4))
p
2> (send p 'x-cord)
3

3> (define q (send p 'move 1 2))
q
4> (send q 'x-cord)
4
5> (send q 'y-cord)
6
```

PS3 - Funktionsorienteret Programmering

Noter

Ovenfor programmerer vi tre simple “primitiver” til hhv. instantiering af en klasse, afsendelse af en besked, og opslag af en metode. Bemærk at disse primitiver alle er funktioner. Vi viser også hvorledes vi laver punkter p og q.

Både new-instance og send har lidt specielle parameterlister. Dotten før sidste formelle parameter betyder, at listen af resterende aktuelle parametre bindes til det formelle parameternavn efter dotten. Dette er Scheme-teknikken til håndtering af “vilkårlig mange parametre”. Bemærk den pæne måde dette noteres på i forhold til dot-notationen, som vi studerende i lektionen om Lisp-kulturen.

Funktionsorienteret stil i praktisk programmering.

- Vær bevidst om fordelene ved funktionsorienteret programmeringsstil.
 - Højnelse af abstraktionsniveau med højereordensfunktioner.
 - Reproducerbare beregninger: letter aftestning og debugging.
- Identificer delproblemer, som lader sig løse i funktionsorienteret stil.

Programmeringsomgivelsen for applikative sprog:

- Read-eval-print "shell": Manifestationen af den interaktive og incrementelle arbejdsform.
 - Evaluering af funktioner (opnåelse af resultater).
 - Definition og redefinition af globale værdier (funktioner).
 - Imperative islæt i den måde omgivelsen tillader os at redefinere funktioner.
 - For dyb blokstruktur er upraktisk.
 - Også anvendelig for andre paradigmer end det funktionsorienterede.
- I praksis slår man bro mellem en teksteditor og read-eval-print shellen.

PS3 - Funktionsorienteret Programmering

Noter

Lad os knytte en bemærkning til det, at der er imperative islæt ved den måde programmeringsomgivelsen tillader os at definere og re-definere navne på funktioner og variable. Det er naturligvis nødvendigt i en programudviklingsproces at have mulighed for at re-definere et navn til en anden "værdi", såsom en modificeret funktion. Dette er typisk et resultat af en fejlfindingsproces. Er dette i konflikt med funktionsorienteret programudvikling? Svaret er nej, idet denne redefinition ikke kan programmeres. I det næste kapitel om Scheme vil det f.eks. være helt meningsløst at lave betingede define's, eller iterative define's. Definitioner og redefinitioner fyres af af en programmør, gennem interaktiv dialog med omgivelsen.

Programbeskrivelsen kan dermed ændre sig over tiden. Dette er en nødvendighed, selv i funktionsorienteret programmering. Selv om programbeskrivelsen ofte kan tilgås fra et kørende program, opfatter vi ikke dette som et problem ift. funktionsorienteret programmering.