1
Introduktion til Funktionsorienteret Programmering.

Dynamiske sprog og omgivelser.
Applikativ og funktionsorienteret programmering.
Eksempler på funktionsorienteret programmering.
Uafhængighed af evalueringsrækkefølge.
Referential transparency.
Substitutionsmodellen.

Noter

I dette kapitel i noterne introducerer vi funktionsorienteret programmering. Vi forsøger at motivere stilarten ved at sætte den op som en kontrast til den mere velkendte imperative stilart. Og vi vil studere nogle fundamentale egenskaber ved funktionsorienterede programmer.

Dynamiske sprog og omgivelser.

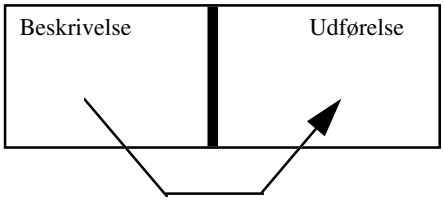
- **Typiske karakteriska ved dynamiske sprog**.
  - Applikativ stil:
    - En programmeringsstil som bruger én fundamental syntaktisk udtryksmåde: anvendelse (applikation) af *funktioner* på argumenter.
  - Typer:
    - Typen af navne, parametre og udtryk optræder ikke eksplicit og statisk i kildeprogrammer.
    - Dataobjekterne kender deres egne typer, og kan direkte udnytte typeinformation under programudførelsen.
  - Automatisk lageradministration:
    - Programmøren skal ikke bekymre sig om deallokering af dataobjekter.
- **Typiske karakteristika ved dynamiske omgivelser**.
  - Baseret på *fortolkning* eller *incrementel oversættelse*.
  - Understøtter en *incrementel* og *interaktiv* programmeringsproces, med mulighed for hurtig afprøvning og feedback.

Noter

"Dynamiske sprog og omgivelser" er en konstrast til de sprog og omgivelser, som har været fremherskende på de første dele af datalogi- og dataingeniørstudiet. Vores udgangspunkt er, at kendskab til disse sprog, og kendskab til de programmeringsteknikker, som naturligt dyrkes i disse sprog, er væsentlig i en afrundet datalogiuddannelse.

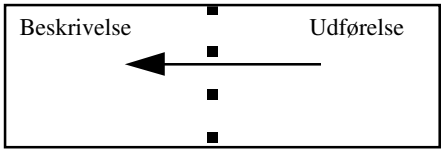
Applikativ programmering kaldes også undertiden værdi-orienteret programmering.

Statisk omgivelse.



- Information går tabt.
- Afstand mellem udførelsesbegreber og programbeskrivelsesbegreber.

Dynamisk omgivelse.



- Tilgang til programbeskrivelsen fra udførelsesomgivelsen.

I figuren til venstre er der en solid mur mellem programbeskrivelse og udførelse. Beskrivelsen transformeres til et lavere niveau, hvilket skaber afstand mellem de to niveauer. Og information går tabt, (f.eks. viden om variables navne). I nogle anvendelsessammenhænge er det nyttigt under programudførelsen at kunne reflektere om programbeskrivelsen. I langt de fleste konventionelle sprog er dette udelukket. I mange dynamiske sprog er det muligt på simpel vis. I den sidste del af kurset vil vi se mere rafinerede eksempler på dette.

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.

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 programmeingssprog, 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 dynamikse programmeringssprog.

Funktionsorienteret kontra imperativ programmering.

- **Funktionsorienteret Programmering**:
  - Funktioner er abstraktioner over udtryk.
  - Applikativ programmering.
  - Navne kan bindes til værdier, men eksisterende navnebindinger's værdier kan ikke ændres.
  - Definition og anvendelse af *højereordens funktioner*.
    - Funktioner er fuldgyldige 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 som en funktion af tiden.
  - En kommando har en målelig effekt på sine omgivelser.
  - Ærke-kommandoen: assignment

PS4 - 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 taler vi om en forandring fra (relativt) kaotiske forhold til mere ordnede forhold og "rene liner". De rene liner og de mere ordnede forhold betyder, at et program bliver lettere at forstå for folk der af én eller anden grund skal sætte sig ind i programmet, herunder naturligvis programmørerne. Det bliver også 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, 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 sideffekter, 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 funtionsorienterede 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.

Seks gode årsager til at studere funktionsorienteret programmering.

1. Undgå brugen af assignment (hvor det er muligt).
2. Programmér på et højere abstraktionsniveau.
3. Forbered dit program på parallel udførelse.
4. Vær beredt på AI (AI programmering foregår i Lisp eller Prolog).
5. Skriv udførbare specifikationer.
6. Slå bro til teoretikerne.

Noter

Ad 1: Brugen af assignment komplicerer den beregningsmæssige model, og det komplicerer ræsonnementer om et program. Det diskuterede vi i noterne til forrige slide. Det er korrekt at funktionsorienteret programmering er uden assignment, men det vil være alt for fattig at opfatte programmering uden assignment som funktionsorienteret programmering. Der hører langt mere med til historien.

Ad 2: Det er velkendt at højnelse af abstraktionsniveau for det meste gør det lettere at håndtere et problem. Vi abstraherer som regel bort fra nogle mindre væsentlige detaljer, som blot tilslører det virkelig væsentlige. Udnyttelsen af højereordens funktioner tillader os at indfange abstraktioner over udtryk, som vi ellers ikke kunne håndtere. Vi vil se mange eksempler på dette i kapitlet om højereordens funktioner senere i noterne.

Ad 3: Som vi vil se lidt senere i dette kapitel kan mange deludtryk i et funktionsorienteret program beregnes samtidig, simpelthen fordi der ikke er mulighed for at påvirke værdien af det ene sådanne udtryk fra de andre. Dette er en klar fordel, hvis man går efter mere effektivitet end der kan opnåes på uni-processor maskiner.

Ad 4: I de sidste 25-30 år er AI (kunstig intelligens) dukket op som en komet, altså med forholdsvis jævne mellemrum. I dette anvendelsesområde er dynamiske sprog særdeles anvendelige, på grund af den ekstreme fleksibilitet. Lisp plejer at spille en store rolle inden for AI.

Ad 5: Algebraiske specifikationer baserer sig på funktioner, og kan i visse tilfælde udføres direkte.

Ad 6: Det kan lade sig gøre at ræsonnere om og bevise funktionsorienterede programmer.

Eksempel: Navnebinding i stedet for assignment (1).

```
solve(a,b,c: real): root-list is
-- returner løsningerne af  ligningen ax²+bx + c = 0
local determinant: real
do
 determinant := b * b - 4 * a * c;
 if determinant > 0
 then result := ...
 elsif determinant = 0
 then result := ...
 else result :=
end
```

Misbrug af assignment.

Noter

Dette eksempel viser et (måske lidt naivt) eksempel i et imperativt program (skrevet i et sprog, som ligner Eiffel), nemlig løsning af en 2. gradsligning. Vi sætter fokus på assignment af variablen '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 variablen om igen. Det forhold, at variablens værdi aldrig ændres efter første tilskrivning retfærdiggør en anden udtryksform: Vi går fra brug af assignment til navnebinding. Dette er emnet på næste slide.

Hvad er forholdet mellem konstanter og de navnebindinger (eller assignments) vi her taler om? Konstanter i et program er størrelser, hvis værdier låses fast inden programudførelsen. Brug af konstanter anbefales ligeledes af hensyn til klarhed og læselighed af programmet, men i lige så høj grad for at sikre at konstanters værdi kan ændres (i programbeskrivelsen, vel at mærke) på en sikker måde.

Eksempel: Navnebinding i stedet for assignment (2).

Navnebinding med syntaktisk sukker

```
solve(a,b,c: real): root-list is
do
 let determinant  be  b * b - 4 * a * c
 in if determinant > 0
    then return ...
    elseif determinat = 0
    then return ...
    else return empty_list end
 end
end
```

Uden syntaktisk sukker:

```
solve(a,b,c: real): root-list is
  local-solve(determinant: real): root-list is
  do
    if determinant > 0
    then return ...
    elseif determinat = 0
    then return ...
    else return empty_list end
  end
do
  return  local-solve( b * b - 4 * a * c  )
end
```

PS4 - Funktionsorienteret Programmering

Noter

Vi arbejder her videre med eksemplet fra forrige slide. Til venstre viser vi i et fiktivt sprog, hvordan navnebinding kunne tage sig ud. I næste forelæsning (som introducrer Scheme) kommer vi til at se helt konkrete konstruktioner til navnebinding.

Til højre illustrerer vi, hvordan man kan anvende en lokal 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 local-solve) og værdien (det aktuelle parameterudtryk i kaldet af local-solve).

Man kan observere, at vi faktisk ikke har særlig meget brug for et navn til funktionen, som binder et eller flere navne for os. Hvis sproget muliggør det, vil en annonym funktion gøre fin fyldest. Dette vil blive mere konkret i næste kapitel, når vi studerer Scheme's let-konstruktion.

Bemærk at vi også ovenfor har introduceret et return udtryk, som definerer værdien af funktionen. Det siger næsten sig selv, at hvis vi bl.a. er interesseret i at komme assignment til livs skal vi ikke brug assignment til en eller anden pseudovariable for at defineret funktionens værdi. Vi definerer ikke nærmere her, om **return** *expr* afbryder funktionskroppen og returnerer umiddelbart til kalderen.

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 datastruktureres kan defineres ved at *abstrahere over bestemte rekursive mønstre* af funktionskald.

Noter

Sædvanlige kontrolstrukturer sekventierer nogle mere primive 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 funtioner kan vi indfange “rekursive mønstre”, og abstrahere over disse. Herved kan vi opbygget et bibliotek af iterations-primitiver på forskellige former for datastrukturer, som er specialdesignede til gentagne beregninger på disse strukturer. Dette er bemækelsesværdigt, og vi vil selvfølgelig vende tilbage til dette i de kommende kapitler.

Eksempel på iteration og rekursion.

En funktion, som gennemløber en liste af tal, og som returnerer en tilsvarende liste 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;
```

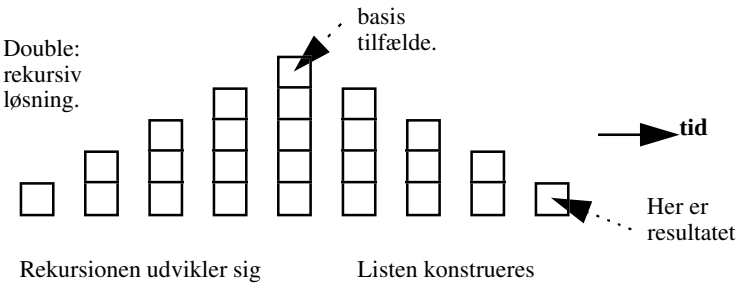
Rekursiv løsning.

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


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 fo doble. Dette kan være et problem, hvis der ikke er sat tilstrækkeligt meget lager af til køretidsstak.

Eksempel på rekursion og mapping.

Halerekursiv løsning:

```
double(lst):
  double-help(lst, empty-list)

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

Løsning via mapning:

```
double(lst):
  map(*2, lst)

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

System-defineret abstraktion over "at anvende en funktion på alle elementer i en liste og returnere listen af resultater af disse anvendelser".

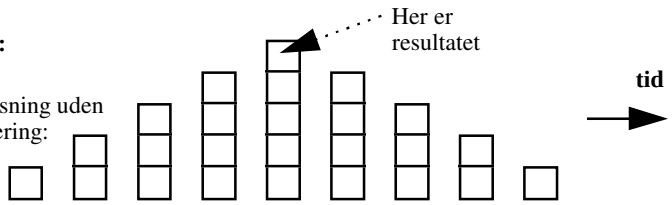
PS4 - Funktionsorienteret Programmering

Noter

På denne slide dropper vi typeangivelsen af parametre og resultatet af funktionerne.

Stakudvikling:

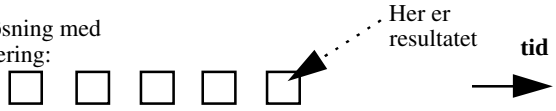
Double:
halerekursiv løsning uden
lageroptimalisering:



Rekursionen udvikler sig

Ren afstakning.
Der sker intet fornuftigt her.

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 resultatet af funktionen. I den halerekursive løsning er det kun nødvendigt med én aktivationrekord, 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 mapning eksemplet forneden kunne vi naturligvis (og burde nok) have anvendt en halerekursiv version af mapning.

Eksempel på højereordens funktion.

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

```
Function derive (f: function(real): real; dx: real): [function(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:

$(R \rightarrow R) \times R \rightarrow (R \rightarrow R)$

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 parameterværdi mon brudt, når vi f.eks. anvender deriv således

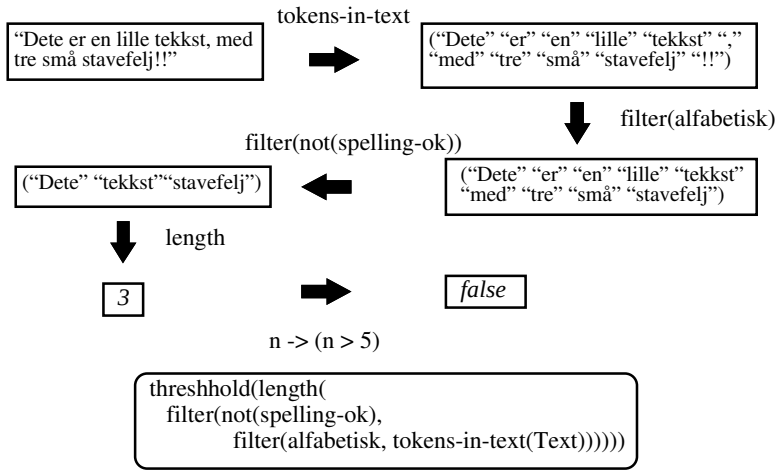
```
let g be derive(sinus, epsilon)
in
   g(2.7)
end
```

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 vandt 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 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.




```
threshhold(length(
  filter(not(spelling-ok),
         filter(alfabetisk, tokens-in-text(Text))))))
```

PS4 - 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(spellig-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.

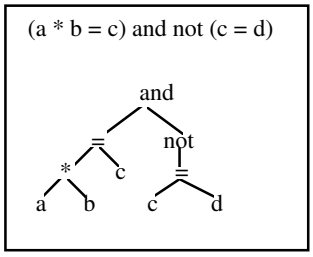
Tilsvarende kan vi opfatte filter som en funktion, der tager en boolsk-returnerende funktion som parameter, og som selv returnereren en funktion:

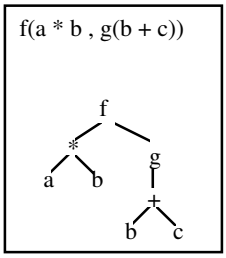
filter: (String -> Boolean) -> (String* -> String*)

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.

Noter

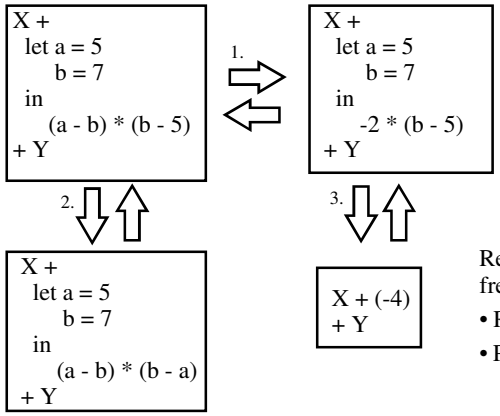
Deludtryk kan beregnes i en vilkårlig rækkefølge: Det er ikke muligt for beregningen af et deludtryk at påvirke beregningen af et deludtryk "i en anden gren" af udtrykket.

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 dog intet til hinder for at funktionen g, i f.eks. Pascal, kunne lave en sideeffekt på variablen 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 "Evalueringsrækkefølge og forsinket evaluering".

Referential transparency (1).

Et udtryk U1, som er lig med et udtryk U2 i en bestemt kontekst, kan erstatte U2 i denne kontekst.



Hvilken form for lighed?

- Samme objekt (eq).
- Strukturel lighed (equal).

Referential transparency fremmer muligheden for

- Ræsonnementer og bevis
- Programtransformationer

Noter

Diskussionen om lighed: Lighed er ikke et entydigt begreb. Vi kender mindst til "identiske objekter" (=), shalow equality (equal), og strukturel lighed (deep_equal). Ifølge Lisp terminologi svarer = (i Eiffel) til eq; deep_equal i Eiffel svarer til equal. Hvilken form for lighed mener vi ovenfor?

Den mindst diskriminerende lighed er tilstrækkelig. I termer af ovenstående er deep_equal god nok. Husk at hvis a=b er equal(a,b) også opfyldt, og hvis equal(a,b) er opfyldt er deep_equal(a,b) opfyldt (men det omvendte gælder ikke). Vi har under funktionsorienteret programmering ingen mulighed for at skelene mellem to objekter eller datastrukturer, som strukturelt set er ens. Hvis vi skulle finde ud af om to sådanne objekter O1 og O2 opfylder O1 = O2 (i Eiffel forstand, altså eq), ville vi være nødt til at prøve at ændre (mutere) den ene, og se om det havde effekt på den anden; men det er jo netop denne mulighed vi er afskåret fra i funktionsorienteret programmering.

I de viste transformationer ovenfor sker der følgende substitutioner:

1. Udtrykket a-b erstattes med tallet -2

2. Tallet 5 erstattes med a

3. Udtrykket (a-b) *(b-5) erstattes af -4.

Hvert af de fire udtryk vist ovenfor kan gøre det ud for hinanden. Der vil aldrig kunne ses forskel på resultatet af den omkringliggende beregning. Konteksten er her en addition af X, let-udtrykket og Y.

Referential transparency (2).

F er en ren funktion:

(1)
```
(3 * F(a,b) + b) * (3 * F(a,b)  - c)
```
⇨

(2)
```
let t be 3 * F(a,b)
in (t + b) *  (t - c)
end
```

Sammenlign med situationen, hvor F er en "uren funktion":

```
F(a,b: integer): integer is
do
  c := c + 1;
  result := 2 * (a + b)
end
```

Antallet af gange F kaldes er vigtig for resultatets værdi.

Rækkefølgen af beregningen af de to faktorer i (1) er også af betydning.

Noter

I et funktionsorienteret program vil de to forekomster af udtrykket

3 * F(a,b)

altid have den samme værdi. Referential transparency giver, at hvis t er lig med 3 * F(a,b), kan t erstatte begge forekomster. Dermed har vi ovenstående transformation.

På denne slide sammenligner vi med et eksemplet på en uren funktion, der laver en tilstandsforandring på sin omgivelse, her variablen c. Hvis F(a,b) kun beregnes én gang giver (1) et andet resultat end hvis F(a,b) beregnes to gange.

Endvidere er det væsentligt hvilken af faktorerne i (1) der beregnes først. Hvis den venstre faktor beregnes først vil c i højre faktor være én større, når højre faktor bliver beregnet. Dette er noget rod, og derfor anbefaler man sædvanligvis i alle sprog, der har funktioner såvel som procedurer, at funktioner ikke har "sideeffekter" (ligesom nødvendigvis, at procedurerne har).

En beregningsmodel baseret på substitution.

Resultatet af et funktionskald kan beregnes ved brug af en meget simpel model: substitutionsmodellen.

> 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).

Modellen fremkommer ved anvendelse af referential transparency og simpel, positionel parameterkorrespondance.

Eksempel:

```
f(5, g(6))
```

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

g(x):
  x 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.

Noter

Som allerede omtalt vil vi et senere kapitel diskutere forskellige rækkefølger af substitutioner. Her 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, som diskuteret ovenfor. Givet at de formelle parametre er bundet til værden af de aktuelle parametre, kan de formelle parametre i kroppen erstattes af argumenterne (eller for den sags skyl af de aktuelle parameterudtryk).

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

Funktionsorienteret stil i praktisk programmering.

- Være bevidst om fordelene ved funktionsorienteret programmeringsstil.
  - Højnelse af abstraktionsniveau.
  - Reproducerbare beregninger: letter aftestning og debugging.
- Identificere 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.

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 resulat 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 tilgåes fra et kørende program, opfatter vi ikke dette som et problem ift. funktionsorienteret programmering.