FÖRELÄSNING 2, TDDC74, VT2018

- Begrepp och definitioner (delvis från föreläsning 1)
- Fallanalys som problemlösningsmetod
- Rekursivt fallanalys
- Rekursiva beskrivningar och processer de kan skapa
- Rekursiva och iterativa processer och deras egenskaper

BEGREPP

Konstant, Namn

Procedur/Funktion, Parameter, Argument, Kropp, “scope”

Villkor, Logiska uttryck

Funktionsanrop, Rekursion

Substitutionsmodellen

Rekursiva definitioner

Processer: Iterativa processer, Rekursiva processer

Tid- och minnesbehov för processer

PROBLEMLÖSNING MED HJÄLP AV FALLANALYS

- Problem kan ofta delas upp
- Mycket enklare att lösa delproblem
- Vanligtvis: basfall och generella fall
- Observera den matematiska uppdelningen av fakultetsfunktionens beskrivning:
  $n! = n(n\text{-}1)!$ när $n > 0$
  $0! = 1$

PROBLEMLÖSNING MED HJÄLP AV REKURSION

- Problemuppdelning leder ofta till rekursiva fall, där beräkningar definieras delvis i termer av sig själva
- De delar som är rekursiva måste då utgöra ett enklare problem än det ursprungliga annars har uppdelningen inte uppfyllt sitt syfte
- Med hjälp av villkorliga uttrycken i Scheme representerar vi de olika fallen i ett problem

BERÄKNINGS AV FAKULTET: 0! = 1, N! = N * (N-1)!

```
(define fact
  (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```


ITERATIV FAKULTETSFUNKTION

Beskrivningar av lösningar i ett programspråk behöver inte exakt motsvara den matematiska uppdelningen. Följande program är också en lösning till fakultetsfunktionen:

```
(define fact-iter
  (lambda (n result)
    (if (= n 0)
        result
        (fact-iter (- n 1)
                   (* n result)))))
```

I den extra parametern `result` samlas delresultatet av beräkningen

ITERATIV FAKULTETSFUNKTION - NÅGOT BÄTTRE

```
(define fact
  (lambda (n)
    (fact-iter n 1)))


(define fact-iter
  (lambda (n result)
    (if (= n 0)
        result
        (fact-iter (- n 1)
                   (* n result)))))
```


SUBSTITUTIONSMODELLEN PÅ BERÄKNING

Skillnaden mellan den rekursiva och den iterativa lösningen kan visas med hjälp av substitutionsmodellen.

SUBSTITUTIONSMODELLEN FÖR (FACT 4)

Använd substitution för att beräkna `(fact 4)`:

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```


SUBSTITUTIONSMODEL

```
(fact 4)
(fact-iter 4 1)
(fact-iter (- 4 1) (* 4 1))
(fact-iter 3 4)
(fact-iter (- 3 1) (* 3 4))
(fact-iter 2 12)
(fact-iter (- 2 1) (* 2 12))
(fact-iter 1 24)
(fact-iter (- 1 1) (* 1 24))
(fact-iter 0 24)
24
```


FIBONACCITALEN

0, 1, 1, 2 , 3, 5, 8, 13, …

Fallanalys

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1)+fib(n-2)

FIBONACCITALEN

```
(define fib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2)))))))
```

FIBONACCITALEN – FALLANALYS 2

0, 1, 1, 2 , 3, 5, 8, 13, …

Fallanalys

fib(0) = 0
fib(1) = 1
fib(n) = fib-iter(n, 0, 1, 1)

fib-iter(n, f0, f1, räknare) =
f1 om (n = räknare)
fib-iter(n, f1, f0+f1, räknare+1) annars

ITERATIV FIBONACCI

```
(define fibi
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (fib-iter n 0 1 1)))))

(define fib-iter
  (lambda (n f0 f1 count)
    (if (= count n)
        f1
        (fib-iter n f1 (+ f0 f1) (+ count 1)))))
```


FIBONACCITALEN – FALLANALYS 3

0, 1, 1, 2 , 3, 5, 8, 13, …

Fallanalys

fibi(0) = 0
fibi(1) = 1
fibi(n) = iter(n, 0, 1)

iter(n, f0, f1) =
f1 om (n = 1)
iter(n-1, f1, f0+f1) annars

***Uppgift**: Skriv Schemekoden för detta fallanalys.*

FALLANALYS – RÅD

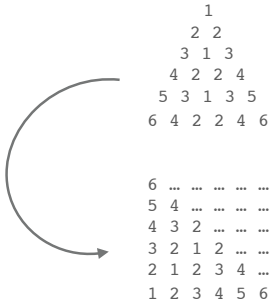
- Uppdelning av ett större problem i enklare fall kräver att man jobbar med problemet
- Sällan kommer man på alla korrekta fallen direkt, det gäller att träna upp sig, lös många uppgifter
- Ibland måste ett ofullständigt fallanalys testas för att förstå problemet bättre och sedan komplettera fallen*
- Ibland måste man studera flera alternativa fallanalys och välja den “bästa” uppdelningen

* Och det är precis därför språk som Scheme och Python passar denna typ av experimentellt problemlösning.

EXEMPEL: HUR KAN VI BERÄKNA TALEN I FÖLJANDE STRUKTUR?

```
      1
    2  2
  3  1  3
4  2  2  4
5  3  1  3  5
6  4  2  2  4  6
```


ETT PROBLEM MÅSTE IBLAND SES FRÅN OLIKA VINKLAR



ÄNTLIGEN EN STRUKTUR SOM KÄNNS IGEN – KOORDINATSYSTEMET

Fallanalys

f(i,j) = j om (i=1)
i om (j=1)
1 om (i=j)
|i-j|+1 annars

Eller?

REKURSIVA BESKRIVNINGAR OCH PROCESSER

- Vi skiljer mellan beskrivningar på program och de processer som dessa beskrivningar skapar
- En beskrivning kan vara självrefererande (dvs rekursiv) men skapa en repetition som inte uppfyller egenskaperna för en rekursiv process.

PROCEDURER OCH PROCESSER

När en procedur evalueras genereras en process. Processerna kan antingen vara

- Rekursiva, eller
- Iterativa

Rekursiva processer i sin tur kan antingen utgöra en

- linjär rekursion eller
- trädrekursion

OBS! att i både fallen kan proceduren ha definierats som en rekursiv **beskrivning** (anropar sig själv)

LINJÄR REKURSIVA PROCESSER

- Ett och samma uttryck som utgör ett resultat, innehåller ett rekursivt anrop som skall kombineras med annat
- Varje rekursivt anrop skapar en fördröjd operation, den fördröjda operationen kombinerar resultatet på det rekursiva anropet med annat, med andra ord,
- Rekursiva anropet ingår i ett resultat genererande uttryck
- Tidskomplexiteten (behovet av beräkningstid) växer linjärt relativt parameterns storlek. I ordo-notation: $O(n)$
- Rymdkomplexiteten (behovet av datorminne) växer linjärt relativt parameterns storlek. I ordo-notation: $O(n)$

ITERATIVA PROCESSER

- Ett rekursivt anrop (se `fact-iter` eller `fibi`)
- Rekursiva anropet skapar inga fördröjda beräkningar
- Rekursiva anropet är svaret (ingår inte i annat uttryck)
- Tidskomplexiteten (behovet av beräkningstid) växer linjärt relativt parameterns storlek. I ordo-notation: $O(n)$
- Rymdkomplexiteten (behovet av datorminne) är konstant och oberoende av parameterns storlek. I ordo-notation: $O(k)$ eller $O(1)$

TRÄDREKURSIVA PROCESSER

- Ett och samma uttryck som utgör ett resultat, innehåller flera rekursiva anrop som skall kombineras (se `fib`)
- Fördröjda operationer växer som träd (förgreningen är beroende av antal rekursiva anrop i ett och samma resultatuttryck, t ex, i `fib` är förgreningen 2)
- Tidskomplexiteten växer exponentiellt $O(k^n)$, där k är antalet multipla anrop, t ex, i `fib`, $k=2$
- Rymdkomplexiteten växer linjärt $O(n)$

UPPGIFT

- *f(n) = n if n<3*
- *f(n) = f(n-1)+2f(n-2)+3f(n-3) annars*

- *Skriv Schemeprocedurer som implementerar f(n) både som rekursiv- och interaktiv process.*
- *Försök följa samma mönster som lösningarna för Fibonaccitalen*