IN2040: Funksjonell Programmering

Mer om Scheme.
Rekursjon og iterasjon.

Michael Kirkedal Thomsen

Universitetet i Oslo

Uke 2 (24. august 2025)



På blokka

Forrige uke

- Introduksjon og oversikt
- Praktiske detaljer
- Funksjonell programmering
- Litt om Scheme og Lisp

I dag

- Mer om egendefinerte prosedyrer
- Substitusjonsmodellen
- Variabler og binding
- Blokkstruktur og rekkevidde
- Rekursjon og iterasjon
- Prosedyrer vs. prosesser



Repetisjon: Scheme på én slide

- Prosedyrekall i Scheme = liste.
- Første listeelement = prosedyren.
- Resten = argumentene.
- Først evalueres alle enkelt-uttrykkene, så kalles prosedyren på argumentverdiene.
  - Symboler evalueres til atomisk verdi.
  - Primitive uttrykk evalueres til seg selv.
  - Sammensatte uttrykk: Som over.
- Eneste unntak fra regelen er noen få såkalte *special forms*.
- For å binde et symbol til en verdi (f.eks. en prosedyre) brukes `define`.
- Prosedyrer lages med `lambda`.

Eksempler

```
> (+ 1 2 10 7 5)
→ 25

> (* 10 (+ 60 40))
→ 1000

> (define bar 42)

> (define foo
    (lambda (x)
      (/ x 10)))

> (foo bar)
→ 4.2
```

Repetisjon: Prosedyrer, *special forms*, og sannhet



- Prosedyrer: først evalueres alle argumentene, så anvendes prosedyren.
- Special forms: styrer selv om og når argumentene evalueres.

- `if` og `cond`: utfallet av testene avgjør hvilke uttrykk som evalueres.
- `or` og `and`: evaluerer uttrykkene én av gangen fra venstre til høyre.
- Disse er eksempler på *special forms*, ikke vanlige prosedyrer.

- Predikater er prosedyrer som returner en boolsk verdi: `#t` eller `#f`.
- Alt annet enn `#f` regnes som sant.
- Derfor vil `or` og `and` gjerne returnere andre verdier enn `#t` eller `#f`.

Evaluering: Substitusjonsmodellen

- I substitusjonsmodellen tenker vi at prosedyrekall omskrives slik:
  - Hele uttrykket erstattes av prosedyre-kroppen fra definisjonen.
  - De formelle parameterene erstattes av argumentene.
- Evalueringsstrategi:
  1. Først evalueres deluttrykkene,
  2. så kalles prosedyren på argumentverdiene.
- vi har brukt standardreglen (husk uke 1) for evaluering av uttrykk også for egendefinerte prosedyrer
- Merk: i funksjonell kode er *rekkefølgen* prosedyreargumentene evalueres i uten betydning (uspesifisert i Scheme).

Eksempel

```
> (define (square x)
    (* x x))

> (square (square 2))

⇒ (square (* 2 2))

⇒ (square 4)

⇒ (* 4 4)

⇒ 16
```

Ulike evalueringsstrategier

Applicative-order evaluation

```
> (square (square 2))
⇒ (square (* 2 2))
⇒ (square 4)
⇒ (* 4 4)
→ 16
```

- Evaluer argumentene; kall prosedyre på verdi.
- Standard i Scheme.

Normal-order evaluation

```
> (square (square 2))
⇒ (* (square 2) (square 2))
⇒⇒ (* (* 2 2) (* 2 2))
⇒⇒ (* 4 4)
→ 16
```

- Prosedyren kalles med argument*uttrykkene*; evalueres først ved behov.
- 'Først ekspander, så reduser.'

- Gir samme resultat så lenge vi holder oss til ren funksjonell kode. Vi skal komme tilbake til forskjellen mot slutten av semesteret.

Sammensatte prosedyrer

Eksempel

```
> (define (avslag p pris)
    (* (/ pris 100) p))

> (avslag 25 600) → 150

> (define (salgspris p pris)
    (- pris (avslag p pris)))

> (salgspris 25 600) → 450
```

- Å definere prosedyrer er en form for abstraksjon.
- Man kan se på hver prosedyre som en 'sort boks':
- I `salgspris` bryr vi oss ikke om hvordan `avslag` er implementert, så lenge den gir oss forventet returverdi.

- Ønsker vi å endre måten vi regner ut `avslag` på holder det å gjøre det i definisjonen; alle andre steder vi måtte bruke `avslag` kan stå uendret.
- Kan teste `avslag` isolert.
- Kortere kode gjennom gjenbruk.

Formelle parametere, og frie / bundne variabler

```
> (define (avslag p pris)
    (* (/ pris 100) p))
```

- Så lenge vi er konsekvente spiller det ingen rolle hvilke navn vi bruker på de formelle parameterene: `p` og `pris`.
- Mer generelt kan vi snakke om to typer variabler: frie og bundne.
- Parameterene er eksempler på *bundne* og *lokale* variabler.
- Rekkevidden (*scope*) for bindingen er kroppen til prosedyren.
- I kroppen til `avslag` kan symbolene `/` og `*` sees som *frie variabler*.
  - Deres bindinger er etablert utenfor denne leksikalske konteksten (i den globale omgivelsen).

Anekdote

```
> (define (avslag p pris)
    (* (/ pris 100) p))
```

- Obskurt, men lovlig:

```
> (define (avslag + -)
    (* (/ - 100) +))
```

- Men hva vil skje her:

```
> (define (avslag * -)
    (* (/ - 100) *))
```

- Unngå å bruke parameternavn som allerede er bundet til prosedyrer!
  - Vi vil da overskygge prosedyrebindingen,
  - og få problemer dersom vi ønsker å kalle prosedyren (mildt sagt . . .).

Interne definisjoner og blokkstruktur

- La oss tenke oss at vi *ikke* ønsker å bruke `avslag` andre steder.
- Kan da definere `avslag` lokalt og internt i definisjonen til `salgspris`; vi sier at de danner en blokk:

```
> (define (salgspris p pris)
    (define (avslag)
      (* (/ pris 100) p))
    (- pris (avslag)))

> (salgspris 25 600) → 450
```

- Her har vi samtidig forenklet `avslag` slik at den er helt uten parametere.
- Variablene pris og p er nå *frie* i definisjonen av `avslag`:
- Tar sin verdi fra bindingene i omsluttende definisjon av `salgspris`.

En parentes om typer

- Typer er knyttet til verdier/objekter snarere enn til variabler.
- Vi sier også at Scheme er dynamisk typet.
- `(+ 1 "hei")` gir run-time feil, ikke kompilatorfeil (eller `"1hei"`).

Føler du at det er noe som mangler?



- Verditilordning / muterbare variabler?
  - Mangler, men ikke en mangel; brukes ikke i funksjonell kode!
- Koding involverer gjerne også en eller annen form for løkke.
- I imperative språk uttrykkes dette med konstruksjoner som `while`, `for`, `do`, `repeat...until`, osv.
  - Iterativ oppdatering av lokale tilstandsvariabler.
- I fravær av muterbare variabler uttrykkes løkker på en annen måte i funksjonell programmering:
  - Rekursive prosedyrekall.
  - Løkker realisert som funksjonelle transformasjoner, ikke basert på side-effekter.

Rekursjon

- En rekursiv prosedyre anvender seg selv i sin egen definisjon.
- Klassisk eksempel: fakultetsfunksjonen.
- Kan se sirkulært ut, men er vel-definert så lenge vi har et basis-tilfelle som terminerer rekursjonen.
- Til sammenlikning: ikke-funksjonell kode uten rekursjon, med `while`-løkke og endring av lokale tilstandsvariabler.[a]

$$n! = \begin{cases} 1 & \text{hvis } n = 1 \\ n \times (n-1)! & \text{hvis } n > 1 \end{cases}$$

```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

```
def fac(n):
    f = 1
    while (n > 1):
        f = f * n
        n = n - 1
    return f
```

[a] Merk: = *her* tilsvarer `set!` i Scheme, det kommer senere.

Rekursjon (forts.)

```
> (define (fac n)
    (if (= n 1)
        1
        (* n (fac (- n 1)))))

> (fac 2) → 2

> (fac 3) → 6

> (fac 4) → 24

> (fac 5) → 120
```

- Rekursjon er mest naturlig når løsningen til et problem baseres på løsningene til reduserte instanser av samme problem.
- Vi reduserer mot basistilfellet (*base case*).
- Nøyaktig hva vi mener med å 'redusere' kommer an på problemet vi jobber med.

Noen distinksjoner

Funksjon

Prosedyre

Prosess

Prosedyre vs. prosess

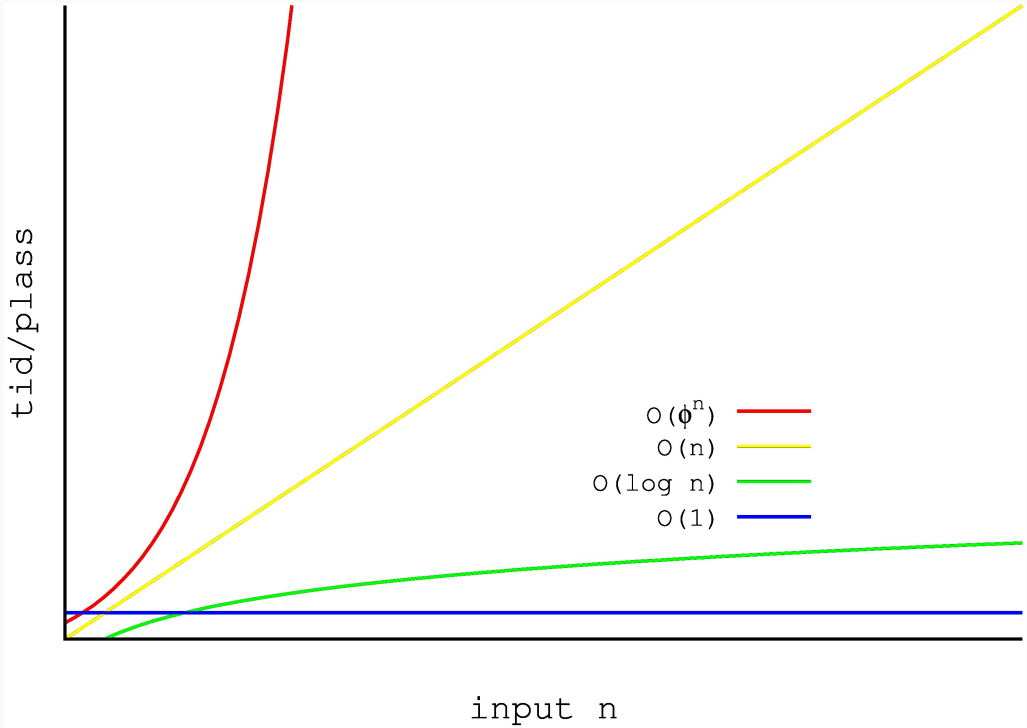
- Hver gang `fac` kaller seg selv har vi også et kall på `*` som venter på returverdien.
- Vi får en *kjede av ventende kall* som fortsetter å vokse helt til vi når basistilfellet, $n = 1$.
- Genererer en rekursiv prosess.

```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

- Jo dypere rekursjon jo mer stakkminne brukes for å utføre prosessen.
- Prosessens tids- og plassbehov vokser direkte proporsjonalt med `n`:
  - Eksempel på en *lineær* rekursiv prosess.
  - Vekstrate (*order of growth*) $= \mathrm{O}(n)$.

Vekstrate

- *Big O*-notasjon:
- Grov karakterisering av hvordan ressursbehov for en prosess vokser relativt til størrelsen på input n.
- O$(g(n))$ betyr at $g(n)$ gir det øvre taket på prosessens vekstrate.
- Eller rettere, $g(n)k$, for en positiv konstant k.



Rekursiv prosess

- Vi kan bruke substitusjonsmodellen til å omskrive et kall på `fac` for å visualisere hvordan den rekursive prosessen utvikler seg:

```
> (fac 7)
⇒ (* 7 (fac 6))
⇒ (* 7 (* 6 (fac 5)))
⇒ (* 7 (* 6 (* 5 (fac 4))))
⇒ (* 7 (* 6 (* 5 (* 4 (fac 3)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (fac 2))))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fac 1)))))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 2)))))
⇒ (* 7 (* 6 (* 5 (* 4 6))))
⇒ (* 7 (* 6 (* 5 24)))
⇒ (* 7 (* 6 120))
⇒ (* 7 720)
→ 5040
```

`fac`, versjon 2

- Vi kan også utrykke $n!$ med en annen prosedyre som genererer en annen type prosess.
- Her gjør vi rekursjonen i en (internt definert) hjelpeprosedyre.

```
(define (fac n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod)
              (+ count 1))))
  (iter 1 1))
```

- I stedet for å etterlate ventende prosedyrekall legger vi til en akkumulatorvariabel som akkumulerer produktet fortløpende. . .
- og en tellervariabel for å definere basistilfellet.
- Når vi når basistilfellet er den endelige returverdien ferdig beregnet.

`fac`, versjon 2 (forts.)

- Prosedyren er fortsatt rekursiv.
- Men den beskriver en iterativ prosess.
- Minnebruken vokser ikke; mengden informasjon som må huskes er konstant.

```
(define (fac n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod)
              (+ count 1))))
  (iter 1 1))
```

- Kan til enhver tid oppsummeres med et gitt antall tilstandsvariabler; i dette tilfellet akkumulator- og tellervariablene og input.
- Antall trinn som utføres i `fac` vokser fortsatt lineært med `n` ($\mathrm{O}(n)$).
- Men plassbehovet er *konstant* ($\mathrm{O}(1)$).

Iterativ prosess

- Igjen omskriver vi prosedyrekallene for å visualisere hvordan den komputasjonelle prosessen utvikler seg:

```
> (fac 7)
⇒ (iter 1 1)
⇒ (iter 1 2)
⇒ (iter 2 3)
⇒ (iter 6 4)
⇒ (iter 24 5)
⇒ (iter 120 6)
⇒ (iter 720 7)
⇒ (iter 5040 8)
→ 5040
```

`fac`, versjon 2 (forts.)

- Merk: det rekursive kallet står i 'haleposisjon' (det siste prosedyren utfører).
- Kan sende returverdien direkte fra det innerste kallet til stedet for det ytterste kallet: *tail call elimination*.

```
(define (fac n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod)
              (+ count 1))))
  (iter 1 1))
```

- Prosedyrens returverdi er den samme som returverdien for det siste rekursive kallet; den er halerekursiv (*tail recursive*).
- Alle Scheme-implementasjoner forutsettes å kunne oppdage når det foreligger halerekursjon og foreta *tail call elimination*.

PS: SICP reserverer termen *tail recursive* til denne siste egenskapen (som egentlig ligger hos interpreteren/kompilatoren) og beskriver ikke prosedyrer i seg selv som halerekursive.

$n!$ som rekursiv og iterativ prosess

Rekursiv prosess

```
> (define (fac n)
    (if (= n 1)
        1
        (* n (fac (- n 1)))))



> (fac 7)
⇒(* 7 (fac 6))
⇒(* 7 (* 6 (fac 5)))
⇒(* 7 (* 6 (* 5 (fac 4))))
⇒(* 7 (* 6 (* 5 (* 4 (fac 3)))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (fac 2))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fac 1)))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 2)))))
⇒(* 7 (* 6 (* 5 (* 4 6))))
⇒(* 7 (* 6 (* 5 24)))
⇒(* 7 (* 6 120))
⇒(* 7 720)
→5040
```

Iterativ prosess

```
(define (fac n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod)
              (+ count 1))))
  (iter 1 1))

> (fac 7)
⇒(iter 1 1)
⇒(iter 1 2)
⇒(iter 2 3)
⇒(iter 6 4)
⇒(iter 24 5)
⇒(iter 120 6)
⇒(iter 720 7)
⇒(iter 5040 8)
→5040
```

$n!$ som rekursiv, iterativ, og imperativ kode

Rekursiv kode

```
> (define (fac n)
    (if (= n 1)
        1
        (* n (fac (- n 1)))))
```

Iterativ kode

```
(define (fac n)
  (define (iter prod count)
    (if (> count n)
        prod
        (iter (* count prod)
              (+ count 1))))
  (iter 1 1))
```

Imperativ kode

```
def fac(n):
    f = 1
    while (n > 1):
        f = f * n
        n = n - 1
    return f
```

Fibonaccitall

- For å illustrere nok en annen type rekursiv prosess skal vi se på en prosedyre for å beregne tall i den såkalte Fibonacci-rekken:
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 \ldots$
- Bortsett fra de to første er hvert tall summen av de to foregående.
- Før vi når basistilfellene fører hvert kall på `fib` til *to* nye rekursive kall.
- Gir opphav til en såkalt trerekursiv prosess.

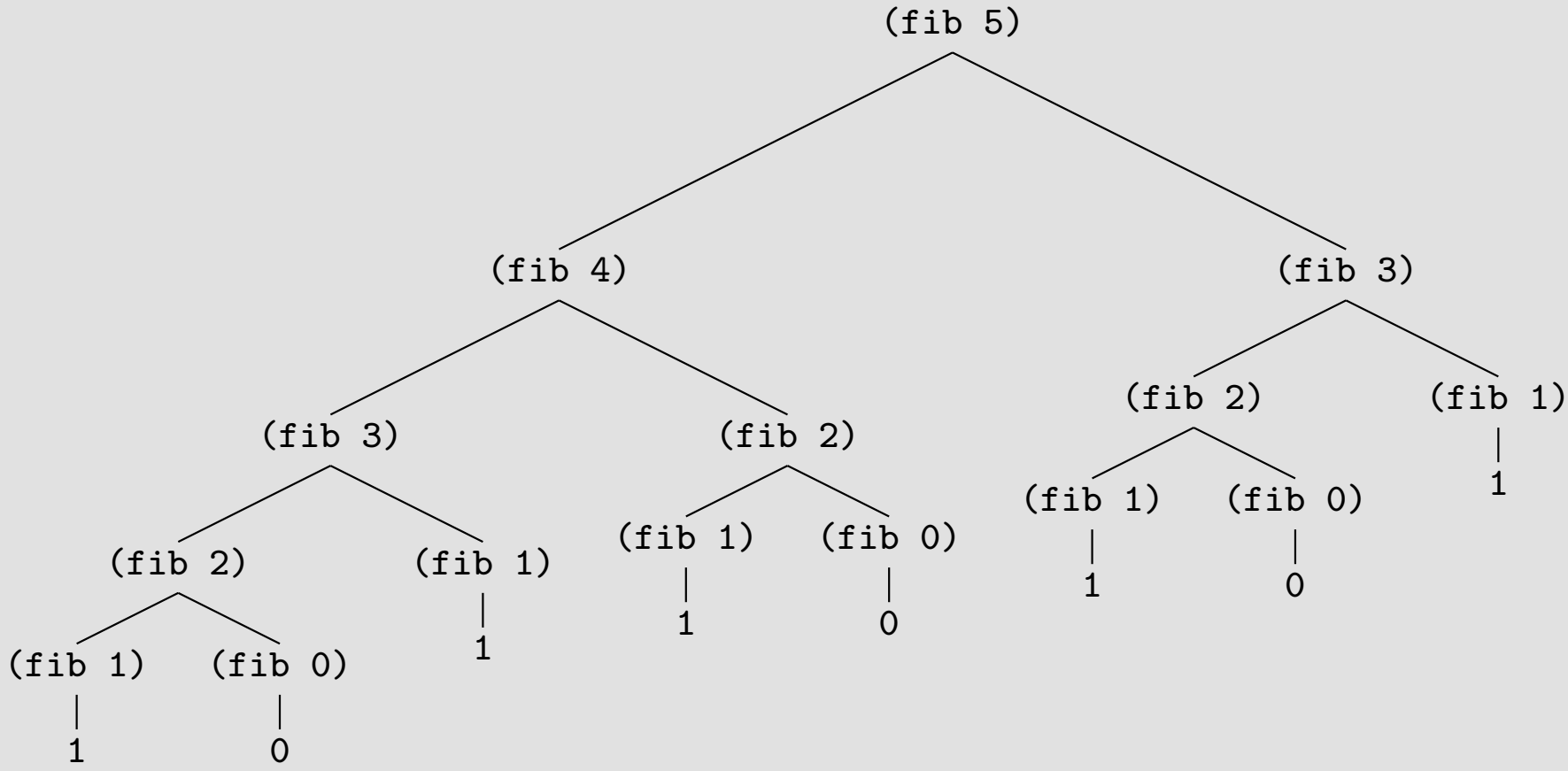
$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

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

```
> (fib 5) → 5
> (fib 6) → 8
> (fib 7) → 13
> (fib 8) → 21
> (fib 9) → 34
```

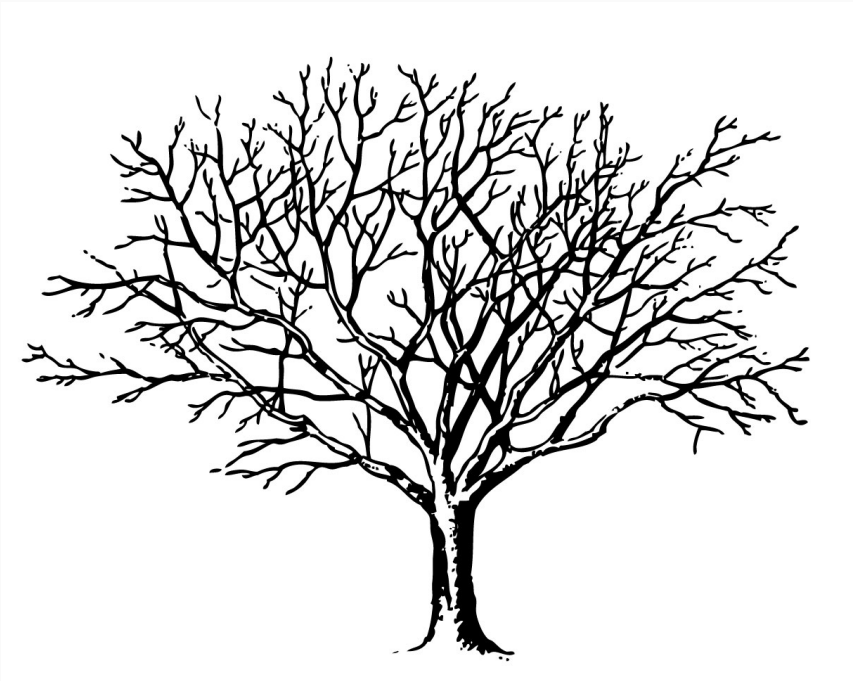
`fib` som trerekursiv prosess

- Veldig mange dupliserte operasjoner i utregningen av fib:
- Eksponensiell vekst i tidsbruk; proporsjonal med antall noder.
- Minnebruken vokser likevel lineært i n, proporsjonal med tredybden (fordi hver operasjon kun trenger å huske det som er ovenfor i treet).



Trerekursjon

- Vår første versjon av `fib` er elegant, men håpløst lite effektiv.
- Men, det betyr ikke at trerekursive prosesser ikke har sin plass:
- Når vi skal jobbe med hierarkiske data vil de være naturlige og nyttige!
- (Scheme-evaluatoren selv er basert på trerekursjon.)



`fib`, versjon 2

- Vi kan omskrive `fib` til en mer effektiv variant.
- Vi bruker to akkumulator-variabler:
- Initialiseres til fib(1) og fib(0), og oppdateres fortløpende.

```
(define (fib n)
  (define (iter a b count)
    (if (= count 0)
        b
        (iter (+ a b)
              a
              (- count 1))))
  (iter 1 0 n))
```

- En teller-variabel forteller når vi er ferdig.
- Halerekursiv: det rekursive kallet står i haleposisjon.
- Gir en iterativ prosess der tiden er lineær i n og minnet konstant:
- Prosessens tilstand kan til enhver tid oppsummeres med variablene vi sender rundt (trenger ikke bruke stack-minne for å holde rede på ventende prosedyrekall).

Rekursjon, igjen

- Eksponentialfunksjonen: vårt foreløpig siste eksempel på hvordan ulike rekursive prosedyrer kan generere ulike prosesser.
- $b^n = a \iff n = log_b(a)$

$$b^n = \begin{cases} 1 & \text{hvis } n = 0 \\ b \times b^{(n-1)} & \text{ellers} \end{cases}$$

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

- For hvert rekursive kall på `expt` etterlates et ventende kall på `*`.
- Nedtelling til basistilfellet.
- Genererer en lineær rekursiv prosess.
- Både plass- og tidsbruk er $O(n)$.

Halerekursiv versjon

- La oss omskrive prosedyren til halerekursiv form.
- Bruker også blokkstruktur og gjør rekursjonen i en intern hjelpeprosedyre.
- Samler resultatet i en akkumulatorvariabel.
- Nedtelling til basistilfellet.
- Lineær iterativ prosess.
- Tid: $O(n)$.
- Minne: $O(1)$.

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))


(define (expt b n)
  (define (iter prod count)
    (if (= count 0)
        prod
        (iter (* b prod)
              (- count 1))))
  (iter 1 n))
```

Rekursjon, igjen (og igjen)

- Men det er mulig å implementere `expt` *enda* mer effektivt.
- I begge definisjonene over har `expt` vært basert på et mønster der f.eks b^8 regnes ut som: $b \times b \times b \times b \times b \times b \times b \times b$.
- Kan omstruktureres:
  - $b^2 = b \times b$
  - $b^4 = (b^2)^2$
  - $b^8 = (b^4)^2$
- Færre operasjoner.
- Rekursiv prosess.
- Logaritmisk vekst; $\mathrm{O}(\log n)$.

$$b^n = \begin{cases} 1 & \text{hvis } n = 0 \\ (b^{n/2})^2 & \text{hvis } n \text{ er partall} \\ b \times b^{(n-1)} & \text{ellers} \end{cases}$$

```
(define (square x)
  (* x x))

(define (expt b n)
  (cond ((= n 0) 1)
        ((even? n)
         (square (expt b (/ n 2))))
        (else (* b (expt b (- n 1))))))
```

Konvensjoner og god stil

- Parenteser lukkes på samme linje.
- Maks linjelengde: 80 tegn.
- Unngå å skifte linje rett etter prefiks (med mindre det er nødvendig pga regelen over).
- Bruk små bokstaver (og bindestrek ved lange navn).
- Bruk navn som er selvforklarende, men så korte som mulig.

God stil:

```
(define (in-interval? x low high)
  (and (> x low)
       (< x high)))
```

Dårlig stil:

```
(define (InInterval? x low high)
  (and (> x low)
       (< x high)
       )
  )

(define (my_check a b c)
  (and
   (> a b)
   (< a c)))
```

Neste gang: Lister



- Datastrukturen liste.
- Listerekursjon
- LISP = LISt Processing