Lisp 2: Lister og funksjoner

Eirik Alderslyst Nygaard Øystein Ingmar Skartsæterhagen

Programvareverkstedet

11. mars 2010

(Lister)

. . . lists are the heart of Lisp . . .

— Guy L. Steele Jr.

(Par)

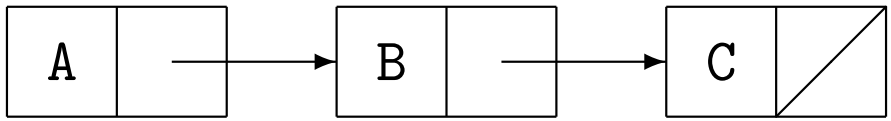
(`CONS` lager et *par* (også kalt *cons-celle*))

(Et par pakker inn to objekter. De kan plukkes ut med `CAR` og `CDR`)

$$\texttt{(car (cons } a\ b\texttt{))} \rightsquigarrow a$$
$$\texttt{(cdr (cons } a\ b\texttt{))} \rightsquigarrow b$$

(Lister er bygget opp av par)

```
(cons 'a (cons 'b (cons 'c nil))) ⇝ (A B C)
```



(Listene i Lisp er altså *enkeltlenkede lister*)

(Induktiv definisjon av lister)

(`NIL` er en liste)

((`cons` *a* *b*) er en liste hvis *b* er en liste)

(Par kan også brukes til andre ting enn lister)

(Det andre argumentet til `CONS` må ikke være en liste)

```
(cons 'a 'b) ⇝ (a . b)
(cons 'answer 42) ⇝ (answer . 42)
```

(Par kan for eksempel brukes til par av enkle verdier, som over, eller til vilkårlige trestrukturer)

(Listeoperasjoner)

(Det finnes mange innebygde funksjoner som opererer på lister)

(Her vil vi se på noen av dem)

(Medlemsskap)

```
(member 'b '(a b c)) ⇝ T
(member 'x '(a b c)) ⇝ NIL
```

(`MEMBER` sjekker om et gitt objekt er med i en liste)

(Sammensetning)

```
    (append '(1 2) '(a b c)) ⇝ (1 2 A B C)
(append '(a b) '(1 2) '(e f)) ⇝ (A B 1 2 E F)
    (append '(a b) '() '(c)) ⇝ (A B C)
```

(`APPEND` tar vilkårlig mange lister som argumenter og returnerer én ny liste bestående av alle elementene fra listene den fikk)

(Reversering)

```
(reverse '(a b c)) ⇝ (C B A)
```

(REVERSE reverserer en liste)

(Substituering)

```
(substitute 'fisk 'b '(a b c b)) ⇝ (A FISK C FISK)
```

(`SUBSTITUTE` substituerer en ny verdi for alle forekomster av en gitt verdi i en liste)

(Utplukking av elementer)

```
  (first '(a b c d e f g h i j)) ⇝ A
 (second '(a b c d e f g h i j)) ⇝ B
  (third '(a b c d e f g h i j)) ⇝ C
 (fourth '(a b c d e f g h i j)) ⇝ D
  (fifth '(a b c d e f g h i j)) ⇝ E
  (sixth '(a b c d e f g h i j)) ⇝ F
(seventh '(a b c d e f g h i j)) ⇝ G
 (eighth '(a b c d e f g h i j)) ⇝ H
  (ninth '(a b c d e f g h i j)) ⇝ I
  (tenth '(a b c d e f g h i j)) ⇝ J
```

(Utplukking av deler)

```
          (first '(a b c)) ⇝ A
           (rest '(a b c)) ⇝ (B C)
           (last '(a b c)) ⇝ (C)
        (butlast '(a b c)) ⇝ (A B)
   (nth 2 '(a b c d e f)) ⇝ C
(nthcdr 2 '(a b c d e f)) ⇝ (C D E F)
```

(`FIRST` gir første element, `REST` resten av listen)

(`LAST` gir siste element, `BUTLAST` hele listen unntatt siste)

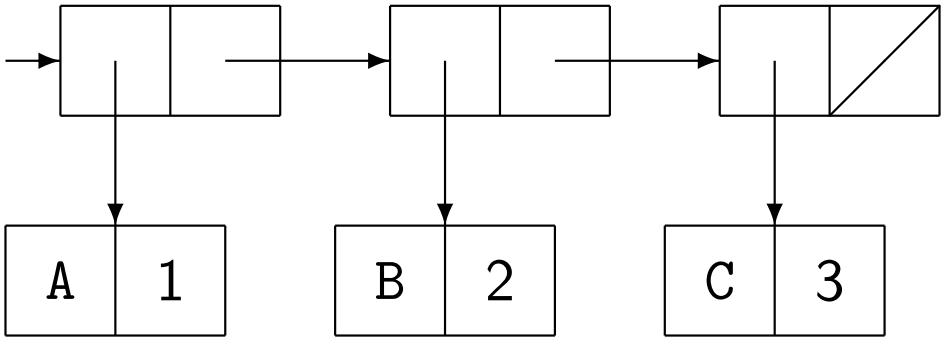
(`NTH` gir n-te element (nullindeksert) for en gitt n)

(`NTHCDR` gir alt fra element n (nullindeksert) og ut)

(Assosiasjonslister)

(En *assosiasjonsliste* (*alist*) brukes for å assosiere nøkler med verdier.)

(En alist er en liste av par, hvor hvert par består av nøkkel og verdi.)



(Assosiasjonslister)

(`ASSOC` brukes for å hente ut et element fra assosiasjonslisten.)

```
(assoc 'c
       (list (cons 'a 1)
             (cons 'b 2)
             (cons 'c 3)))
⇝ (C . 3)
```

(Du kan variere hva som er nøkkelen, og hvordan du vil gjøre testen ved å sende inn nøkkelordargumenter til `ASSOC`.)

(Funksjoner)

And we have learned, slowly and sometimes laboriously over the years, that while lists are the heart of Lisp, functions are the soul.

— Guy L. Steele Jr.

(Firkantsitering)

(Funksjoner er også objekter)

(For å få tak i funksjonsobjektet som skjuler seg bak et navn kan vi *firkantsitere* det)

```
#'cons ⇝ #<FUNCTION ...>
```

(Dette kan vi for eksempel bruke til å fortelle en sorteringsfunksjon hvilken funksjon som skal brukes for å sammenligne elementene:)

```
(sort '(2 11 1 6 3 23) #'<=) ⇝ (1 2 3 6 11 23)
(sort '(2 11 1 6 3 23) #'>=) ⇝ (23 11 6 3 2 1)
```

(λ)

```
(lambda (x) (* x x)) ⇝ #<FUNCTION ...>
```

(`LAMBDA` lager en funksjon)

(Men er det ikke `DEFUN` som lager funksjoner?)

(`DEFUN` både lager en funksjon og binder et navn til den)

(`LAMBDA` bare lager en funksjon)

(Men kan vi gjøre noe med funksjonen vi har laget?)

(APPLY og FUNCALL)

```
(apply (lambda (x) (* x x)) '(4)) ⇝ 16
(apply #'+ '(4 5 6)) ⇝ 15
```

(APPLY tar inn en funksjon og en argumentliste, og kaller funksjonen med den gitte argumentlisten)

```
(funcall (lambda (x) (* x x)) 4) ⇝ 16
(funcall #'+ 4 5 6) ⇝ 15
```

(FUNCALL er som APPLY, men tar inn argumentene til funksjonen som separate argumenter istedenfor som en liste)

(Merk at vi må bruke firkantsitering for å få tak i navngitte funksjoner)

(Funksjoner som opererer på funksjoner)

(Vi kan skrive funksjoner som tar funksjoner som argumenter og returnerer funksjoner)

```
(defun compose (f g)
  (lambda (x)
    (funcall f (funcall g x))))
```

```
(funcall (compose #'first #'rest) '(a b c)) ⇝ B
```

(Eksempel: Kall en funksjon på alle elementene i en liste)

```
(defun apply-over (fun list)
  (if (null list)
      nil
      (cons (funcall fun (first list))
            (apply-over fun (rest list)))))
```

```
  (apply-over #'square '(2 3 5 7)) ⇝ (4 9 25 49)
(apply-over #'first '((a b) (37))) ⇝ (A 37)
```

(`FUNCALL` brukes for å kalle `FUN` på hver element i innlisten)

(Resultatene fra `FUNCALL` blir brukt som elementer i den nye listen)

(I virkeligheten brukes `MAPCAR` eller `MAP`)

(Eksempel: Kollaps en liste til én verdi)

```
(defun foldr (fun initial list)
  (if (null list)
      initial
      (funcall fun
               (first list)
               (foldr fun initial (rest list)))))
```

```
(foldr #'* 1 '(1 2 3 4)) ⇝ 24
```

(I virkeligheten ville man brukt `REDUCE`)

(Reformulering av funksjoner med FOLDR)

```
(defun !* (n)
  (foldr #'* 1 (iota 1 n)))

(defun len* (list)
  (foldr (lambda (_ l) (1+ l))
         0
         list))
```

(Høyere-ordens listefunksjoner)

```
(mapcar #'1+ '(1 2 3 4 5)) ⇝ (2 3 4 5 6)
(mapcar #'cons '(a b) '(1 2)) ⇝ ((A . 1) (B . 2))
(remove-if #'evenp '(1 2 3 4 5)) ⇝ (1 3 5)
(substitute-if 42 #'evenp '(1 2 3 4 5)) ⇝ (1 42 3 42 5)
```

(Rekursjon)

(Funksjoner som kaller seg selv er *rekursive*)

(Rekursive funksjoner er vanlige i Lisp, og mange av funksjonene vi har sett på til nå har vært rekursive)

(Halekall)

(Den siste funksjonen som utføres i en annen funksjon kalles et *halekall*)

```
(defun square (x)
  (* x x)) ; halekall av *
```

(Halerekursjon)

(Når det rekursive kallet i en funksjon er hale, sier vi at funksjonen er *halerekursiv*)

```
(defun drop (n list) ; reimplementation of NTHCDR
  (if (= n 0)
      list
      (drop (- n 1) (rest list))))
```

(Fordelen med halerekursive funksjoner er at det ikke er nødvendig å holde orden på en stakk med gjenstående evaluering mens funksjonen rekurerer)

(Vi evaluerer en halerekursiv funksjon)

```
(defun drop (n list)
  (if (= n 0)
      list
      (drop (- n 1) (rest list))))
```

```
(drop 2 '(a b c))  ⇒  (if (= 2 0) '(a b c) (drop ...))
                   ⇒  (drop (- 2 1) (rest '(a b c)))
                   ⇒  (drop 1 '(b c))
                   ⇒  (if (= 1 0) '(b c) (drop ...))
                   ⇒  (drop 0 '(c))
                   ⇒  (if (= 0 0) '(c) (drop ...))
                   ⇝  (C)
```

(Fordeler med halerekursjon)

(Siden en halerekursiv funksjon bruker en konstant mengde plass på funksjonsstakken uansett hvor dyp rekursjonen er kan man skrive funksjoner som rekurerer vilkårlig lenge uten å sprenge stakken)

(En halerekursiv funksjon utføres typisk raskere enn en ikke-halerekursiv variant, siden funksjonskallene kan optimaliseres bort)

(Merk: Dette er avhengig av at kompilatoren gjør halekalloptimalisering)

(Det er ikke hale alt som rekurerer)

(Mange funksjoner kan skrives halerekursivt, men det er ikke alltid den mest naturlige måten å uttrykke dem på)

(`DROP`, som vi definerte over, ble hale på første forsøk)

(Fakultetsfunksjonen vi definerte for en stund siden er ikke halerekursiv:)

```
(defun ! (n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))
```

(Det rekursive kallet er inni kallet til `*`)

((! n) bruker $O(n)$ stakkplass)

```
(! 5)  ⇒  (* 5 (! 4))
       ⇒  (* 5 (* 4 (! 3)))
       ⇒  (* 5 (* 4 (* 3 (! 2))))
       ⇒  (* 5 (* 4 (* 3 (* 2 (! 1)))))
       ⇒  (* 5 (* 4 (* 3 (* 2 (* 1 (! 0))))))
       ⇒  (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
       ⇒  (* 5 (* 4 (* 3 (* 2 1))))
       ⇒  (* 5 (* 4 (* 3 2)))
       ⇒  (* 5 (* 4 6))
       ⇒  (* 5 24)
       ⇝  120
```

(Omskrivning til halerekursjon)

(Kan vi lage en halerekursiv fakultetsfunksjon?)

(Det vanlige halerekursivifiseringstrikset er å legge til et eller flere ekstra parametre)

(De nye parametrene brukes til å *akkumulere* verdiene som bygges opp og/eller til å holde orden på hvor langt i beregningen vi er kommet)

(Fakultet for halerekursivitet)

```
(defun ! (n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))
```

```
(defun fact-tailrec (n acc)
  (if (= n 0)
      acc
      (fact-tailrec (- n 1) (* n acc))))
```

(Den siste linjen er vrengt så det rekursive kallet kom ytterst)

(Oppstart av FACT-TAILREC)

```
(defun fact-tailrec (n acc)
  (if (= n 0)
      acc
      (fact-tailrec (- n 1) (* n acc))))
```

(For å bruke FACT-TAILREC må vi passere inn en startverdi for akkumulatoren)

(fact-tailrec 5 1) $\rightsquigarrow$ 120

```
(defun fact (n)
  (fact-tailrec n 1))
```

(Eksempel: Fibonacci (igjen))

```
(defun fib (n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

(FIB er ikke halerekursiv: Den gjør to rekursive kall, og +-kallet kan ikke utføres før begge disse har returnert)

(Eksempel: Fibonacci, halerekursiv versjon)

```
(defun fib-rec (n i fibI fibI-1)
  (if (= i n)
      fibI
      (fib-rec n (1+ i)
               (+ fibI fibI-1) fibI)))

(defun fib* (n)
  (if (< n 2)
      1
      (fib-rec n 1 1 1)))
```

(Parameterlister)

```
(defun fn (args ...
           &optional optional-args ...
           &rest rest-arg
           &key keyword-args ...)
  ...)
```

(Funksjoner kan ta inn obligatoriske argumenter, valgfrie argumenter og argumenter identifisert av nøkkelord.)

(Du kan også fange opp alle resterende argumenter ved bruk av `&rest`.)

(Nøkkelordparametre må alltid komme etter alle de andre parametrene)

(Nøkkelord og valgfrie argumenter)

```
(defun fn-key (&key name (age 20))
  (list name age))
```

```
(defun fn-optional (&optional name (age 20))
  (list name age))
```

```
(fn-key :age 41 :name "Alan") ⇝ ("Alan" 41)
(fn-optional "Alan" 41) ⇝ ("Alan" 41)
(fn-key :name "Alice") ⇝ ("Alice" 20)
(fn-optional "Alice") ⇝ ("Alice" 20)
```

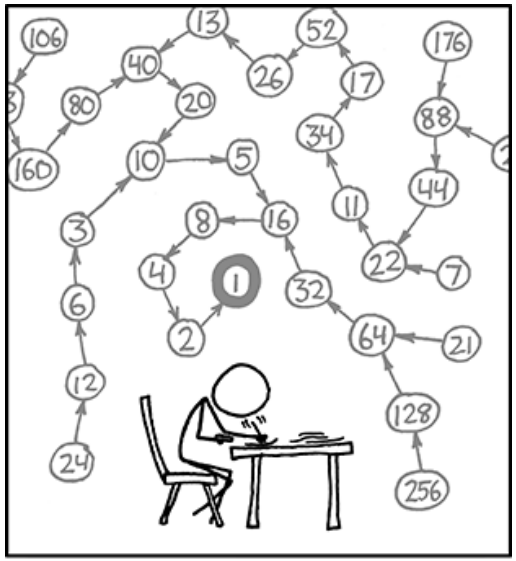
(Resterende argumenter)

```
(defun fn-rest (mom &rest children)
  (list mom children))
```

```
(fn-rest "Alice") ⇝ ("Alice" NIL)
(fn-rest "Alice" "Bob" "Eve") ⇝ ("Alice" ("Bob" "Eve"))
```

(`&rest`-parameteret bindes til en liste som inneholder alle argumenter som ikke ble fanget av tidligere parametre)

(Oppgaver)



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

http://xkcd.com/710/

1. Skriv en funksjon `FLIP` som reverserer lister:
   `(flip '(1 2 3))` ⇝ `(3 2 1)`
2. Er `FLIP`-funksjonen din halerekursiv? Hvis ikke: Kan du skrive den om til å bli det?
3. Skriv en funksjon `REPEAT` som kaller en funksjon gjentatte ganger til en gitt sluttverdi nås, og samler alle verdiene i en liste:
   `(repeat #'1+ 1 5)` ⇝ `(1 2 3 4 5)`
4. Skriv en funksjon `COLLATZ-LIST` som finner Collatz-sekvensen fra et gitt naturlig tall (sekvensen med tall man får ved gjentatt anvendelse av `COLLATZ` til man når 1):
   `(collatz-list 5)` ⇝ `(5 16 8 4 2 1)`

(Løsningsforslag: FLIP)

```
(defun flip-append (list tail)
  (if (null list)
      tail
      (flip-append (rest list)
                   (cons (first list) tail))))

(defun flip (list)
  (flip-append list nil))
```

(I virkeligheten ville man brukt REVERSE)

(Løsningsforslag: REPEAT, COLLATZ-LIST)

```
(defun repeat (fun initial end &key (test #'eql))
  (if (funcall test initial end)
      (list end)
      (cons initial
            (repeat fun (funcall fun initial) end
                    :test test))))

(defun collatz-list (n)
  (repeat #'collatz n 1))
```