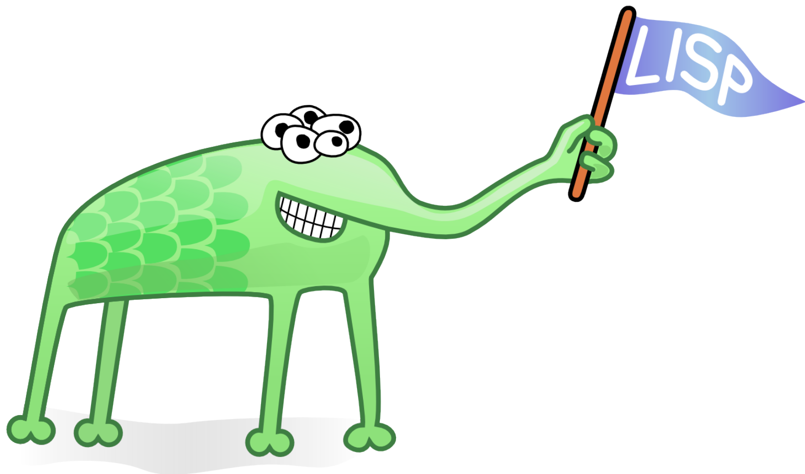
LISP



PVV-kurs 25. oktober 2012

Oversikt over kurset

Hva er Lisp?

Grunnleggende konsepter

Variabler

(Pause)

Lister

Løkker

Funksjoner

Først: Få tak i en implementasjon av Common Lisp

Mange implementasjoner å velge mellom. Noen av de mest populære:

- SBCL
- CLISP

Jeg bruker SBCL.

Bør også ha SLIME (Superior Lisp Interaction Mode for Emacs).

For Emacs-brukere på Debian/Ubuntu:

```
# aptitude install sbcl slime
```

Si deretter `M-x slime` i Emacs.

Hva er Lisp?

Generelle fakta om Lisp

- Det nest eldste høynivåspråket (det eldste ordentlige høynivåspråket?)
- Finnes mange forskjellige dialekter
- Uniform og enkel syntaks
- Automatisk minnehåndtering
- Kan både kompileres og tolkes
- Egnet til funksjonell programmering
- Lisp-kode består av Lisp-objekter

Begynnelsen

LISP ble først beskrevet i John McCarthys artikkel *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* (del 2 ble aldri skrevet).

To typer notasjon: M-uttrykk (innfiks, komplisert syntaks) for å skrive programmer, S-uttrykk (prefiks, enkel syntaks) for data og intern representasjon av programmer.

Artikkelen inneholdt en beskrivelse av `EVAL` – en funksjon som evaluerer et LISP-uttrykk.

Steve Russell lagde den første Lisp-interpreteren ved å oversette `EVAL` til maskinkode for IBM 704.

Siden har ingen sett noe til M-uttrykkene.

Dialekter

Mange varianter av Lisp ble implementert for forskjellige maskiner i løpet av 1960- og 1970-årene: Maclisp, Interlisp, Franz Lisp, ZetaLisp, ...

1975: Scheme, første dialekt med *leksikalt skop*

1984: Emacs Lisp

1984: Common Lisp, standard i form av konglomerat av tidligere dialekter

(I dag: Hovedsakelig Scheme og Common Lisp som brukes)

Grunnleggende konsepter

REPL

Vi kan kommunisere med lispen vår gjennom en REPL (read-eval-print loop).

REPL-en gjør følgende i en løkke:

- `READ`: les et uttrykk fra brukeren
- `EVAL`: evaluer uttrykket til et resultat
- `PRINT`: skriv ut resultatet til brukeren

Vi leker litt med REPL-en

Vi evaluerer følgende uttrykk:

```
245
(+ 6 15)
(+ 3 5 24 72)
(* 3 4)
(- 20 14)
(+ (* 3 4) (- 20 14))
(sqrt 16)
(sqrt 34)
```

Vi lager nye funksjoner

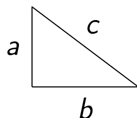
Vi kan DEfinere nye FUNksjoner med `DEFUN`.

```
(defun navn parameterliste funksjonskropp)
```

For eksempel kan vi lage en funksjon `SQUARE` som kvadrerer et tall:

```
(defun square (x) (* x x))
```

En større funksjon



$$c = \sqrt{a^2 + b^2}$$

```
(defun hypothenuse (side1 side2)
  (sqrt (+ (square side1) (square side2))))
```

SLIME

SLIME gjør Emacs til et integrert utviklingsmiljø for Common Lisp.

De viktigste kommandoene:

`M-x slime`	start SLIME
`C-M-x`	evaluer toppnivåuttrykket rundt punkt
`C-c C-l`	last inn en fil
`C-c C-z`	hopp til REPL-bufferet
`C-c C-d h`	slå opp i dokumentasjonen (HyperSpec)

IF og tester

Et IF-uttrykk ser slik ut:

```
(if test konsekvens alternativ)
```

Eksempel:

```
(defun gjett (tall)
  (if (= tall 42)
      "riktig"
      "galt"))
```

Sant og usant

Verdien `T` står for sant, og `NIL` for usant.

`NIL` brukes også som verdi for «ingenting».

`NIL` er også den tomme listen (mer om dette senere).

(For de som kan Python: `NIL` tilsvarer både `False`, `None` og `[]`.)

Eksempel: Fakultet

$$n! = \begin{cases} 1 & \text{hvis } n = 0 \\ n \cdot (n-1)! & \text{ellers} \end{cases}$$

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

Eksempel: Fibonacci

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2) \quad \text{(for } n >= 2)
\end{aligned}
$$

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

Lister

Vi kan lage lister med funksjonen `LIST`.

```
(list 1 2 3)
(list 24)
(list (list 3 5 7) (list 2 3))
(list)
```

Sitering

Vi kan *sitere* et uttrykk ved å skrive en enkelfnutt foran uttrykket.

```
 (+ 4 38) ⇝ 42
'(+ 4 38) ⇝ (+ 4 38)
       pi ⇝ 3.14159...
      'pi ⇝ pi
```

Ved å sitere et uttrykk beskytter vi det fra å bli evaluert.

Et parentetisert uttrykk i koden er en liste akkurat som dem vi lager med `LIST`.

Ord i koden (som + og `pi` over) er *symboler*.

Når er to objekter det samme?

Funksjoner for å teste om to objekter er like:

`EQ`	Tester identitet, fin for symboler
`EQUAL`	Fin for lister
`=`	For tall
`STRING=`	For strenger

(Det finnes mange flere likhetsfunksjoner.)

Formatert utskrift

Funksjonen FORMAT lar oss skrive ut verdier i henhold til en formatstreng.

```
(format t formatstreng verdi ...)
```

Eksempler:

```
(format t "Svaret er ~A.  (Men hva er spørsmålet?)~%"
        42)
(format t "Noen objekter: ~A, ~A, ~A.~%"
        '(a b c)
        50
        "foobar")
```

Variabler

Lokale variabler: LET

Operatoren LET lager lokale variabler som eksisterer inni LET-uttrykket.

```
(let ((variabel verdi) ...) kropp)
```

Eksempel:

```
(let ((x 17)
      (y 38))
  (list x y (+ x y) (* x y)))
```

Endring av variabler: SETF

```
(setf variabel verdi)
```

```
(let ((x 3)
      (y 5))
  (format t "~A = ~A, ~A = ~A~%"
          'x x 'y y)
  (setf x (+ x y))
  (setf y (list 1 2 3))
  (format t "~A = ~A, ~A = ~A~%"
          'x x 'y y))
```

Globale variabler

Vi lager globale variabler med `DEFVAR`:

```
(defvar variabelnavn startverdi)
```

```
(defvar *foo* 42)
```

Konvensjon: Globale variabler har navn som begynner og slutter med asterisk (*). Dette hindrer navnekollisjon mellom lokale og globale variabler.

Spesielle variabler

Globale variabler blir automatisk *spesielle variabler*. Slike følger ikke de vanlige reglene for skop.

Eksempel:

```
(defvar *foo* 42)

(defun print-foo ()
  (format t "foo = ~A~%" *foo*))

(defun dostuff ()
  (let ((*foo* 0))
    (print-foo)
    (setf *foo* 5)
    (print-foo))
  (print-foo))
```

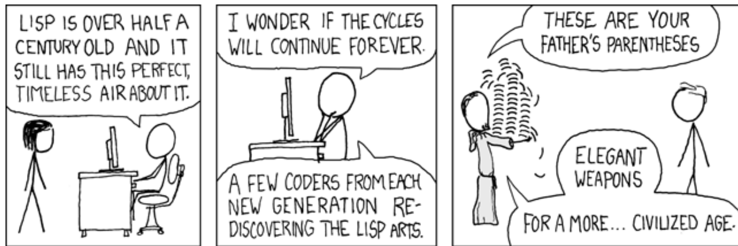
Mer om SETF

Første argument til SETF trenger ikke være en variabel.

Mange uttrykk som slår opp i datastrukturer kan brukes som første argument til SETF.

```
(defvar *the-list* (list 'foo 'bar 'baz))
(setf (first *the-list*) 'flaff)
```

Pause



http://xkcd.com/297/

Lister

Vi plukker lister fra hverandre . . .

Funksjonen `FIRST` gir første element i en liste.

Funksjonen `REST` gir en liste med de resterende elementene.

```
                (first '(a b c d)) ⇝ A
                 (rest '(a b c d)) ⇝ (B C D)
         (first (rest '(a b c d))) ⇝ B
  (first (rest (rest '(a b c d)))) ⇝ C
```

. . . og setter dem sammen igjen

Funksjonen CONS tar et element og en liste, og setter dem sammen til en ny liste.

```
(cons 'moo '(foo bar baz)) ⇝ (MOO FOO BAR BAZ)
```

Om vi kaller FIRST og REST på samme liste, og så kaller CONS på resultatene, får vi en liste som er lik den vi startet med:

```
(defvar *list* (list 'a 'b 'c 'd))
(cons (first *list*) (rest *list*))
```

Den tomme listen: NIL

Den tomme listen heter NIL.

Dette gir en liste med ett element:

```
(cons 'foo nil)
```

Og dette gir en liste med to elementer:

```
(cons 'foo (cons 'bar nil))
```

Funksjonen NULL

Funksjonen NULL tester om noe er den tomme listen.

```
      (null nil) ⇝ T
(null '(a b c)) ⇝ NIL
```

(Husk at NIL både er den tomme listen og verdien for usant.)

Eksempel: Sum av en liste med tall

```
(defun sum (list)
  (if (null list)              ; Hvis listen er tom
      0                        ; er dens sum 0,
      (+ (first list)          ; ellers første tall pluss
         (sum                  ; summen av
          (rest list)))))      ; resten av listen.
```

Eksempel: Kvadratene av alle tall i en liste

```
(defun list-of-squares (numbers)
  (if (null numbers)
      nil
      (cons (square (first numbers))
            (list-of-squares (rest numbers)))))
```

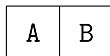
Byggesteinen for lister: Par

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

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

```
(car (cons 'a 'b)) ⇝ A
(cdr (cons 'a 'b)) ⇝ B
```

Vi tegner paret `(cons 'a 'b)` slik:



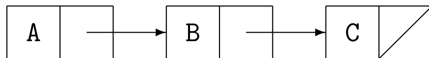
Lister er bygget opp av par

```
(list 'a 'b 'c)
```

gir samme resultat som

```
(cons 'a (cons 'b (cons 'c nil)))
```

Denne listen består altså av tre par:



Listene i Lisp er *enkeltlenkede lister*.

Løkker

Løkkeoperatorer

Common Lisp har mange operatorer for å lage løkker: `DO`, `DO*`, `DOTIMES`, `DOLIST`, `LOOP`.

Vi skal se på `LOOP`, som kan brukes til å skrive alle slags løkker.

LOOP

Et `LOOP`-uttrykk ser generelt slik ut:

```
(loop veldig mye rart ...)
```

En `LOOP` kan fungere som en for-løkke, en while-løkke, en do-while-løkke, en foreach-løkke, eller noe helt annet.

En for-løkke

```
(loop
  for i from 1 to 10
  do (format t "~A~%" i))
```

En while-løkke

```
(let ((x 1))
  (loop
     while (> x 0)
     do (format t "Square root: ~A~%" (sqrt x))
     do (format t "New value:~%")
     do (setf x (parse-integer (read-line)))))
```

En do-while-løkke

```
(loop
  do (format t "Continue?~%")
  while (string= (read-line) "yes"))
```

En foreach-løkke

```
(loop
  for x in '(foo bar baz quux)
  do (format t "Element: ~A~%" x))
```

Kombinasjon av flere typer løkker

```
(loop
  for i from 0
  for x in '(foo bar baz quux stop-here moo moo moo)
  while (not (eq x 'stop-here))
  do (format t "Element ~A: ~A~%" i x))
```

Oppsamling av verdier

```
(defun iota (n)
  (loop
     for i from 0 to n
     collect i))
```

Betingelser

```
(defun divisors (n)
  (loop
     for i from 1 to n
     if (= (mod n i) 0)
     collect i))
```

Funksjoner

Firkantsitering

Funksjoner er objekter.

For å få tak i funksjonsobjektet et funksjonsnavn er bundet til, kan vi bruke firkantsitering:

```
#'+
#'cons
#'square
```

Høyereordens funksjoner

Common Lisp har mange *høyereordens funksjoner*: funksjoner som tar andre funksjoner som argumenter.

Et eksempel er `SORT`, som tar en sammenlikningsfunksjon som andre argument:

```
(sort (list 5 2 17 8 9 24 12) #'<)
(sort (list 5 2 17 8 9 24 12) #'>)
```

MAPCAR

Funksjonen MAPCAR anvender en gitt funksjon på alle elementene i en liste, og samler alle resultatene i en liste.

```
(mapcar #'square (list 4 9 32))
```

REMOVE-IF

Funksjonen REMOVE-IF fjerner elementer fra en liste dersom en gitt funksjon returnerer sant når den anvendes på elementene.

```
(remove-if #'symbolp '(a 5 b 32 c 4))
```

Det er ofte nyttig å kombinere REMOVE-IF med MAPCAR, for å filtrere vekk uønskede elementer:

```
(mapcar #'square (remove-if #'symbolp '(a 5 b 32 c 4)))
```

Anonyme funksjoner: λ

Operatoren `LAMBDA` lager en funksjon:

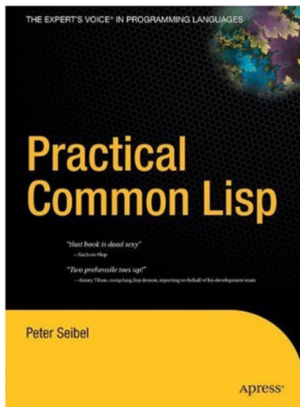
```
(lambda parameterliste funksjonskropp)
```

Når vi kaller en høyereordens funksjon, kan vi gi den en ny `LAMBDA`-funksjon istedenfor et firkantsitert navn:

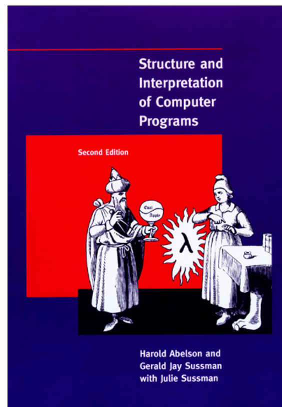
```
(mapcar (lambda (x) (list x (sqrt x)))
        (list 16 24 56))
```

Hva nå?

Anbefalte bøker



http://gigamonkeys.com/book

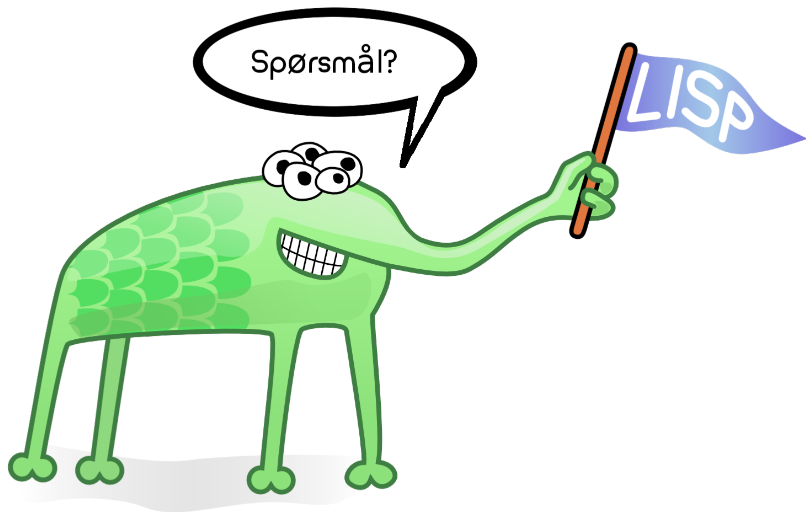


http://mitpress.mit.edu/sicp

Common Lisp HyperSpec

Common Lisp-spesifikasjonen i HTML-format.

`http://www.lispworks.com/documentation/HyperSpec/Front/index.htm`


Spørsmål?
LISP

NIL