Lisp 3: Spesielle former, variabler, imperativ programmering

Eirik Alderslyst Nygaard Øystein Ingmar Skartsæterhagen

Programvareverkstedet

11. mars 2010

(Spesielle former)

Did you ever lie awake the entire night
glowing in the dark with no sheets or light
thinking that `IF` *was something special?*

— fritt etter Bigbang

(Former)

(Et objekt som skal evalueres kalles en *form*)

(Vi har sett at vi anvender en operator ved å sette den som første element i en listeform:)

```
(+ 3 5)
(if (= n 0) 0 (/ 1 n))
```

(Vi vet at +, = og / i eksemplene over er funksjoner)

(Er `IF` en funksjon?)

(Evaluering av funksjonskall)

(Vi ser på evaluering av en enkel funksjonskallform:)

```
(+ (* 2 2) (- 10 9))
```

(Først evalueres argumentformene. Resultatene blir her 4 og 1)

(Så anvendes funksjonen på argumentene. + får altså tilsendt argumentlisten `(4 1)`. Den vet ingenting om hvor disse verdiene kommer fra. Men den *vet* hvordan tall adderes – og resultatet blir 5)

(Er IF en funksjon?)

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

(Hva skjer under evaluering av `(! 0)` hvis `IF` er en funksjon?)

(Først evalueres argumentene:)

$$\begin{aligned} \texttt{(= 0 0)} &\rightsquigarrow \texttt{T} \\ \texttt{1} &\rightsquigarrow \texttt{1} \\ \texttt{(* 0 (! (- 0 1)))} &\rightsquigarrow \cdots \end{aligned}$$

(Under evalueringen av det siste argumentet må `(! -1)` evalueres. Evalueringen av dette vil igjen kreve at `(! -2)` evalueres, og så videre)

(Spesielle former)

(En *spesiell form* er en listeform som evalueres på en annen måte enn som et funksjonskall)

(En spesiell form har navnet på en *spesiell operator* som første element istedenfor navnet på en funksjon)

(Hver spesielle operator har sine egne evalueringsregler)

(`IF` er en spesiell operator)

(De spesielle operatorene)

(Common Lisp har tjuefem spesielle operatorer)

BLOCK	LET*	RETURN-FROM
CATCH	LOAD-TIME-VALUE	SETQ
EVAL-WHEN	LOCALLY	SYMBOL-MACROLET
FLET	MACROLET	TAGBODY
FUNCTION	MULTIPLE-VALUE-CALL	THE
GO	MULTIPLE-VALUE-PROG1	THROW
IF	PROGN	UNWIND-PROTECT
LABELS	PROGV	
LET	QUOTE	

(Vi skal bare se på noen av disse)

(QUOTE)

```
'foo ⇝ FOO
(quote foo) ⇝ FOO
```

(Siteringstegnet er bare en kortform for den spesielle formen QUOTE)

(QUOTE er den enkleste spesielle operatoren: Den tar ett argument, og returnerer argumentet uten å gjøre noe med det)

(IF)

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

(Først evalueres uttrykket *test*)

(Hvis testuttrykket ga en sann verdi evalueres *konsekvens*)

(Ellers evalueres *alternativ*)

(Det er altså alltid ett deluttrykk som ikke blir evaluert)

(Dette er grunnen til at `IF` kan brukes til å få rekursjon til å stoppe)

(COND)

(COND er – som IF – en spesiell operator[1] for betinget evaluering)

```
(cond (test1 konsekvens1)
      (test2 konsekvens2)
      ...)
```

(COND tilsvarer *if—else if*-konstruksjonen i andre språk)

(IF kan alltid brukes, men gir dyp nøsting når det er mange tester)

(Det er vanlig å bruke T som siste test i en COND for å få et tilfelle å falle tilbake på hvis alle de andre testene er usanne)

[1] strengt tatt en makro, men det har ikke så mye å si for oss

(Eksempel: Fletting av sorterte lister)

(Med IF)

```
(defun braid (list1 list2) ; merge
  (if (null list1)
      list2
      (if (null list2)
          list1
          (if (<= (first list1) (first list2))
              (cons (first list1)
                    (braid (rest list1) list2))
              (cons (first list2)
                    (braid list1 (rest list2)))))))
```

(Eksempel: Fletting av sorterte lister)

(Med `COND`)

```
(defun braid* (list1 list2)
  (cond
    ((null list1) list2)
    ((null list2) list1)
    ((<= (first list1) (first list2))
     (cons (first list1)
           (braid (rest list1) list2)))
    (t
     (cons (first list2)
           (braid list1 (rest list2))))))
```

(Variabler)

One man's constant is another man's variable.

— Alan J. Perlis

(Variabler)

(En variabel knytter et navn til en verdi)

(Variabelnavnene i Lisp er symboler)

(All evaluering foregår i et *miljø*, som inneholder variabelbindinger)

(Et miljø lever typisk inni et annet miljø)

(Evaluering av et symbol: Symbolet slås opp i det lokale miljøet, deretter i det utenfor, og så videre, til en variabel med det navnet blir funnet. Resultatet av evalueringen er verdien til variabelen)

(Typer)

(Det er *verdiene* som har typer, ikke variablene)

(Noen typer vi har sett: Tall, symboler, cons-celler, nil)

(Predikater for å sjekke om et uttrykk er av en bestemt type: `NUMBERP`, `SYMBOLP`, `CONSP`)

(Alle objekter er like, men noen er likere enn andre)

(`EQ` er en funksjon som tar to argumenter og sjekker om de er samme objekt)

(Ting som ser like ut er ikke nødvendigvis samme objekt)

(`CONS` lager et nytt objekt, så

```
(eq (cons 42 nil) (cons 42 nil)) ⇝ NIL
```

selv om begge argumentene er listen `(42)`)

(For å finne ut om to lister er like (uten å nødvendigvis være samme objekt) kan vi bruke `EQUAL`)

```
(equal (cons 42 nil) (cons 42 nil)) ⇝ T
```

(Sammenligningspredikater)

Predikat	Ser på
`EQ`	Objektidentitet
`EQL`	Tall: type og verdi; ellers som `EQ`
`EQUAL`	Lister: elementvis `EQUAL`; ellers som `EQL`
`EQUALP`	Tall: verdi; ellers som `EQUAL`

(Moral: Jo flere tegn vi skriver i sammenligningspredikatet, desto mindre skal til for at argumentene skal være like)

> *Object equality is not a concept for which there is a uniquely determined correct algorithm.*
>
> — Common Lisp-spesifikasjonen

(λ lager lokale variabler)

(Parametrene til en funksjon bindes til argumentverdiene når funksjonen kalles)

(Med unntak av `DEFUN`, som lager «funksjonsvariabler» er dette den eneste måten vi har sett å lage variabler på)

(Hva om vi vil ha lokale variabler (som ikke er parametre) inni en funksjon?)

(LET)

(LET er spesiell, den oppfører seg ikke som en funksjon)

(LET introduserer lokale variabler som eksisterer inni LET-formen)

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

(Eksempel: LET)

```
(defun assoc-value (key alist &key (default nil))
  (let ((pair (assoc key alist)))
    (if pair
        (cdr pair)
        default)))
```

```
(assoc-value 'b (list (cons 'a 1) (cons 'b 2))) ⇝ 2
(assoc-value 'c (list (cons 'a 1) (cons 'b 2))) ⇝ NIL
```

(Eksempel: LET med flere variabler)

```
(defun prefix-to-infix (expr)
  (let ((func (first expr))
        (arg1 (second expr))
        (arg2 (third expr)))
    (list arg1 func arg2)))
```

```
(prefix-to-infix '(+ 1 2)) ⇝ (1 + 2)
(prefix-to-infix '(* 2 2)) ⇝ (2 * 2)
(prefix-to-infix '(- 1 2)) ⇝ (1 - 2)
```

(Globale variabler)

(Globale variabler kan lages med `DEFVAR` eller `DEFPARAMETER`)

```
(defparameter variabel verdi dokumentasjon)
(defvar variabel verdi dokumentasjon)
```

```
(defparameter +buffer-size+ 1024
  "Size of input buffer, in bytes")
(defvar *state* 'ready
  "Current state of the state machine")
```

(Funksjonell programmering)

(Alle eksempler vi har vist til nå har vært skrevet i *funksjonell* stil)

(Alt en funksjon gjør er å produsere en returverdi, og denne verdien avhenger kun av argumentene til funksjonen)

(Hvis en funksjon kalles flere ganger med samme argumenter, returnerer den det samme hver gang)

(Dette gjør programmene ryddige, men enkelte ting blir vanskelige å implementere)

(SETF)

(setf *ting verdi*)

(SETF erstatter verdien i en ting med en ny verdi)

(SETF-eksempel)

```
CL-USER> (defvar *x* 5)
*X*
CL-USER> *x*
5
CL-USER> (setf *x* (+ *x* 3))
8
CL-USER> *x*
8
CL-USER> (setf *x* (list 1 2 3))
(1 2 3)
CL-USER> (setf (first *x*) 5)
5
CL-USER> *x*
(5 2 3)
CL-USER>
```

(READ og PRINT)

(READ brukes til å lese inn et lisputtrykk. Uttrykket vil bli parset og returnert som vanlige lisp-objekter)

(PRINT skriver ut lisp-objekter i en leselig form (den samme representasjonen som READ leser inn)

```
CL-USER> (first (read))
(1 2 3)
1
CL-USER> (+ (read) (read))
40 2
42
CL-USER> (print (list 1 2 3))
(1 2 3)
(1 2 3)
CL-USER>
```

(FORMAT)

(FORMAT brukes til å skrive ut formatert tekst)

```
(format destinasjon formatstreng argumenter)
```

(Destinasjonen kan være en strøm, T eller NIL)

```
CL-USER> (format t "Hello, world!")
Hello, world!
NIL
CL-USER> (format nil "Hello, world!)
"Hello, world!"
CL-USER> (format *standard-output* "Hello, world!")
Hello, world!
NIL
CL-USER>
```

(FORMAT med enkle argumenter)

```
CL-USER> (format t "Dagens tall er: ~D~%" (random 10))
Dagens tall er: 4
NIL
CL-USER> (format t "Jeg heter ~S og er ~D år gammel~%"
"John" 82)
Jeg heter "John" og er 82 år gammel
NIL
CL-USER> (format t "Dagens gullkorn er: ~A~%"
"Sometimes the appropriate response to reality is to go
insane.")
Dagens gullkorn er: Sometimes the appropriate response to
reality is to go insane.
NIL
CL-USER>
```

(FORMAT kan skrive ut tall på forskjellige måter)

```
CL-USER> (format t "~R~%" 1501205)
one million five hundred one thousand two hundred five
NIL
CL-USER> (format t "~:R ~:R ~:R~%" 1 2 5)
first second fifth
NIL
CL-USER> (format t "~@R ~@R ~@R~%" 1 130 1259)
I CXXX MCCLIX
NIL
CL-USER>
```

(FORMAT lar deg skrive ut tall med sine engelske navn eller som det romerske tallsystemet)

(FORMAT med lister og litt mer)

```
CL-USER> (format t "~{~A, ~}" '(1 2 3))
1, 2, 3,
NIL
CL-USER> (format t "~{~A~^, ~}" '(1 2 3))
1, 2, 3
NIL
CL-USER> (format t "~{~{~10@<~:(~r~)~>~}~%~}"
                   '((1 2) (3 4) (5 6)))
One        Two
Three      Four
Five       Six
NIL
```

(FORMAT lar deg skrive ut og formatere lister som du selv vil)

(Iterasjon)

(Alt som kan uttrykkes iterativt kan også uttrykkes rekursivt)

(Har sett: Minne- og hastighetsfordelene med iterasjon kan også oppnås med halerekursjon)

(Iterasjon er altså egentlig ikke nødvendig)

(Men det finnes likevel iterasjonsoperatorer)

(Gjenta noe et antall ganger: DOTIMES)

```
CL-USER> (dotimes (i 10) (format t "~D" i))
0123456789
NIL
CL-USER>
```

```
(dotimes (variabel ganger) ting-å-gjøre)
```

(Iterer over en liste: DOLIST)

```
CL-USER> (dolist (a '(foo bar baz)) (format t "~A " a))
FOO BAR BAZ
NIL
CL-USER>
```

```
(dolist (variabel liste) ting-å-gjøre)
```

(Generell iterasjon: LOOP)

(LOOP: sveitsisk arméiterasjonsform)

(LOOP kan brukes til å skrive for-løkker, while-løkker, foreach-løkker og andre rare løkker)

(LOOP er et eget «lite» programmeringsspråk)

(LOOP FOR)

```
CL-USER> (loop for num in '(1 2 3)
               do (format t "Item: ~A~%" num))
Item: 1
Item: 2
Item: 3
NIL
CL-USER>
```

```
CL-USER> (loop for num from 1 to 3
               do (format t "Item: ~A~%" num))
Item: 1
Item: 2
Item: 3
NIL
CL-USER>
```

(Generering av verdier med LOOP)

```
CL-USER> (loop for num in '(1 2 3)
               sum num)
6
CL-USER>
```

```
CL-USER> (loop for num in '(1 5 2)
               maximize num)
5
CL-USER>
```

(Generering av lister med LOOP)

(Loop kan gjøre noe med hvert element i en liste og returnere en ny liste med resultatene)

```
(loop for num in '(1 2 3)
  collect (* 2 num))
⇝ (2 4 6)
```

(Det er også mulig å traversere flere lister samtidig)

```
(loop
  for x in '(1 2 3)
  for y in '(10 20 30)
  collect (+ x y))
⇝ (11 22 33)
```

(Betingelser i LOOP)

```
CL-USER> (loop for x from 1 to 5
           if (evenp x)
             do (format t "~A is even~%" x)
           else
             do (format t "~A is odd~%" x))
1 is odd
2 is even
3 is odd
4 is even
5 is odd
NIL
CL-USER>
```

(Vi kan få deler av løkken til å bli utført kun på visse betingelser)

(Og mye mye mye mer)

(Ikke bare lister)

(Common Lisp har tabeller (én- og flerdimensjonale) og hash-tabeller også)

(Dokumentasjon, bøker etc.)

- *Common Lisp HyperSpec*: Spesifikasjonen i hypertekstform
  `http://www.lispworks.com/documentation/HyperSpec/Front/`
- *Practical Common Lisp*
  `http://gigamonkeys.com/book/`
- *SICP*, trollmannsboken (bruker Scheme, ikke Common Lisp)
  `http://mitpress.mit.edu/sicp/`