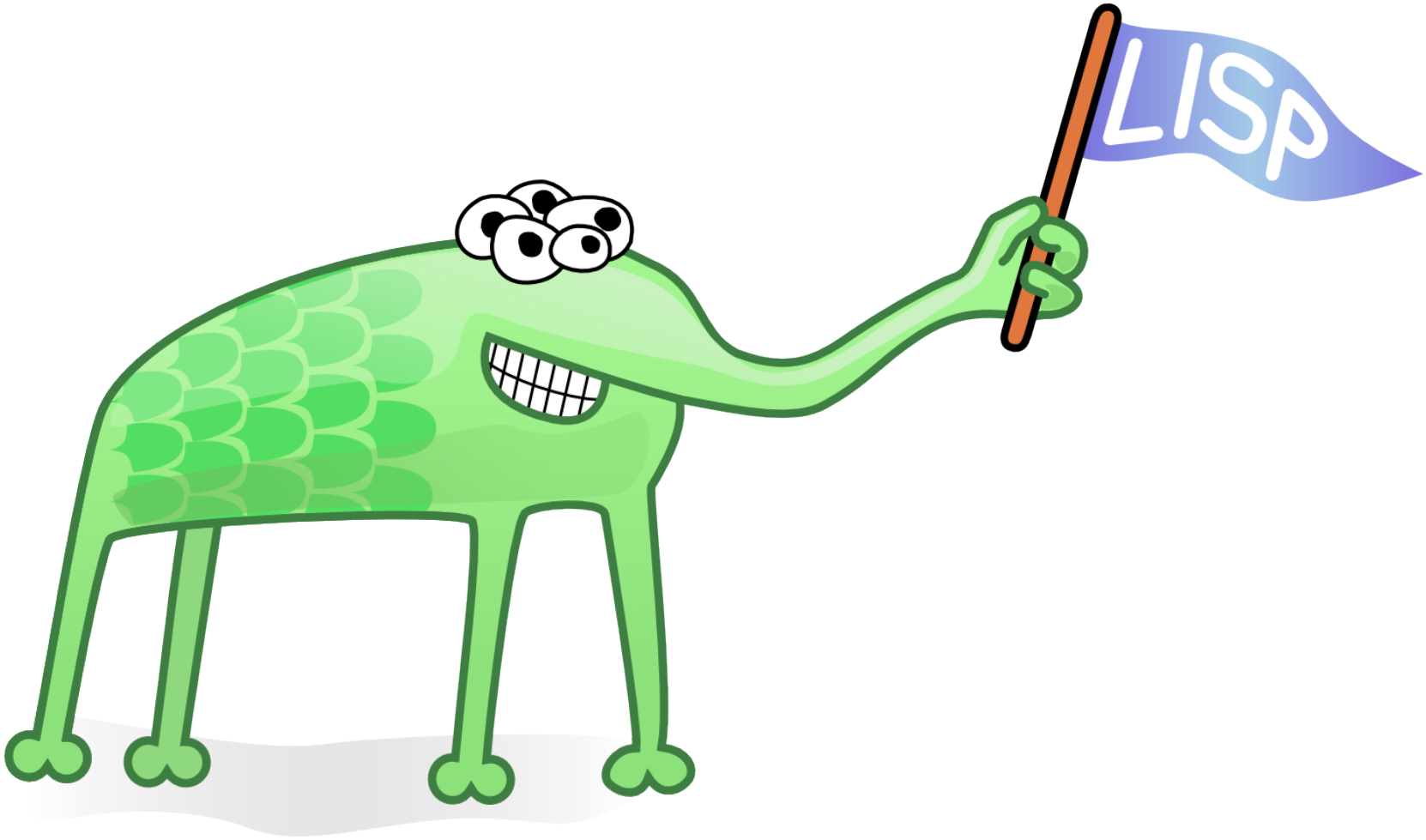
LISP



PVV-kurs 11. mars 2010

LISP has jokingly been described as "the most intelligent way to misuse a computer." I think that description a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.

— Edsger Dijkstra

Oversikt over kurset

(**Del 1**: Historie og grunnleggende Lisp)

(**Del 2**: Lister og funksjoner)

(**Del 3**: Spesielle former, verdier og variabler, imperativ programmering)

Lisp 1: Introduksjon til Lisp

Eirik Alderslyst Nygaard Øystein Ingmar Skartsæterhagen

Programvareverkstedet

11. mars 2010

(Først: Få tak i en Lisp)

(For Emacs-brukere:

```
# aptitude install sbcl slime
```

Gjør deretter `M-x slime` i Emacs)

(For ikke-Emacs-brukere:

```
# aptitude install sbcl
$ sbcl
```

)

(PVV-medlemmer kan kjøre `sbcl` eller `slime` på en PVV-maskin (f.eks. `sprint`))

(I begynnelsen)

(I begynnelsen var `NIL`)

(McCarthy sa `CONS`, og det ble lister)

(Prehistorisk tid)

(1956: IPL (Information Processing Language), assemblyspråk for listeprosessering)

(1957: Den første FORTRAN-kompilatoren)

(FLPL (FORTRAN List Processing Language))

(Begynnelsen på Lisp)

(1958: John McCarthy forsøkte å bruke FLPL til å lage et system for symbolsk derivasjon)

(Problemer med FORTRAN: Ingen rekursjon eller `IF`, tungvint å frigjøre minne)

(Høsten 1958 startet McCarthy sammen med Marvin Minsky arbeidet med LISP)

(LISP)

(1960: LISP blir beskrevet i 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)

(Struktur og tolking av Lisp-programmer)

(Et Lisp-program består av Lisp-objekter)

(Objektene i et program kan *evalueres* til andre objekter)

> *A LISP programmer knows the value of everything, but the cost of nothing.*
>
> — Alan Perlis

(Noen ting evaluerer til seg selv)

```
43 ⇝ 43
2 ⇝ 2
"hallo, verden" ⇝ "hallo, verden"
t ⇝ T
nil ⇝ NIL
```

(Pilen ⇝ står her for «evaluerer til». Den er ikke en del av Lisp)

(Så evaluering gir bare ut igjen det man putter inn? (Det virker ikke veldig nyttig))

(Aritmetikk)

```
(+ 2 2) ⇝ 4
(* 3 5) ⇝ 15
(- 12 4) ⇝ 8
(/ 12 4) ⇝ 3
```

(De aritmetiske operasjonene +, *, - og / er *funksjoner* som tar tall som argumenter)

(Vi kaller en funksjon ved å skrive funksjonsnavnet og argumentene adskilt med mellomrom innenfor et par parenteser)

(Kan funksjonene ta mer enn to argumenter?)

```
(+ 1 2 3 4 5 6 7 8 9 10) ⇝ 55
          (* 1 2 3 4 5) ⇝ 120
```

(Kan vi ha funksjonskall inni funksjonskall inni . . . ?)

```
(+ 4 5) ⇝ 9
(+ (* 2 2) 5) ⇝ 9
```

(Vi kan sette uttrykk inni uttrykk inni . . .)

```
(+ (* 2 3) (/ 10 (+ 2 2 (- 8 7))) (- 5 3)) ⇝ 10
```

(LISP står forresten for «Lots of Irritating Silly Parentheses»)

(Vi lager nye funksjoner)

(Vi kan DEfinere nye FUNksjoner med `DEFUN`)

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

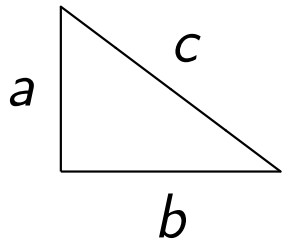
(For eksempel kan vi lage en funksjon `SQUARE` som kvadrerer et tall)

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

(Når vi evaluerer `(square 4)` blir kroppen til `SQUARE` evaluert med 4 satt inn for `X`)

```
(square 4)  ⇒  (* 4 4)
            ⇝  16
```

(En større funksjon)



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

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

(Moral: funksjoner kan ha mer enn ett parameter, og funksjonskroppen kan være et vilkårlig komplisert uttrykk)

`(hypothenuse 6 8)` $\rightsquigarrow$ 10

(Hva skjer når vi kaller HYPOTHENUSE?)

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

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

```
(hypothenuse 6 8)  ⇒  (sqrt (+ (square 6) (square 8)))
                   ⇒  (sqrt (+ (* 6 6) (* 8 8)))
                   ⇒  (sqrt (+ 36 64))
                   ⇒  (sqrt 100)
                   ⇝  10
```

(Betinget evaluering med IF)

```
(defun describe-number (n)
  (if (evenp n)
      "It is even!"
      "It is odd!"))
```

```
(describe-number 37) ⇝ "It is odd!"
(describe-number 42) ⇝ "It is even!"
```

(Eksempel: Fakultet)

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

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

(Evaluering av !)

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

$$
\begin{aligned}
\texttt{(! 0)} &\rightsquigarrow \texttt{1} \\
\texttt{(! 1)} &\Rightarrow \texttt{(* 1 (! 0))} \\
&\Rightarrow \texttt{(* 1 1)} \\
&\rightsquigarrow \texttt{1} \\
\texttt{(! 2)} &\Rightarrow \texttt{(* 2 (! 1))} \\
&\Rightarrow \texttt{(* 2 1)} \\
&\rightsquigarrow \texttt{2}
\end{aligned}
$$

(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)

(Lister er gøy)

```
(list 1 2 3) ⇝ (1 2 3)
   (list 42) ⇝ (42)
```

(`LIST` er en funksjon som tar vilkårlig mange argumenter og returnerer alle i en liste)

```
(first (list 1 2 3)) ⇝ 1
 (rest (list 1 2 3)) ⇝ (2 3)
```

(`FIRST` gir det første elementet i en liste)

(`REST` gir hele listen *unntatt* det første elementet)

(FIRST av REST av REST av ...)

```
(first (rest (list 1 2 3))) ⇝ 2
(first (rest (rest (list 1 2 3)))) ⇝ 3
```

(FIRST gir første element. For å få tak i et vilkårlig element kan vi gå nedover listen med REST vilkårlig mange ganger og så ta FIRST)

(Eksempel: Plukk ut *n*-te element)

```
(defun elem (n list)
  (if (= n 0)                  ; hvis N er 0
      (first list)             ; vil vi ha første element,
      (elem (- n 1)            ; ellers vil vi ha (N-1)-te
            (rest list))))     ; fra resten av listen
```

`(elem 2 (list 2 3 5 7 11))` $\rightsquigarrow$ `5`

(I virkeligheten vil man bruke funksjonen `NTH` for dette)

(Den tomme listen – og alle de andre)

(Den tomme listen er ikke som andre lister)

```
(list) ⇝ NIL
```

(Den tomme listen heter `NIL`)

(`NIL` er også verdien for usant. Dette er litt forvirrende)

```
(null (list 1)) ⇝ NIL
  (null (list)) ⇝ T
```

(`NULL` sjekker om en liste er den tomme listen)

(Listenavigasjon)

(`FIRST`, `REST` og `NULL` gir oss all informasjonen vi trenger for å komme oss rundt i en liste)

(`FIRST` gir oss et element (det første))

(`REST` lar oss komme videre til resten av listen)

(`NULL` forteller oss når det er på tide å stoppe)

(Ved hjelp av disse tre funksjonene kan vi skrive mange funksjoner som gjør nyttige ting med lister)

(Eksempel: Lengden til en liste)

```
(defun len (list)
  (if (null list)          ; Hvis listen er tom
      0                    ; er lengden 0,
      (+ 1                 ; ellers er den 1 pluss
         (len              ; lengden til
          (rest list))))) ; resten av listen.
```

```
(list-length (list 1 2 3)) ⇝ 3
      (list-length (list)) ⇝ 0
```

(I virkeligheten vil man bruke funksjonen `LENGTH` istedenfor å lage denne)

(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.
```

```
(sum (list 1 2 3 4 5)) ⇝ 15
          (sum (list)) ⇝ 0
```

(Sitering)

```
  (+ 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*)

(CONStruksjon av lister)

```
(cons 3 '(4 5 6)) ⇝ (3 4 5 6)
(cons '+ '(37 5)) ⇝ (+ 37 5)
(cons 'foo '(bar)) ⇝ (FOO BAR)
(cons 'foo nil) ⇝ (FOO)
```

(`CONS` gjør det motsatte av `FIRST` og `REST`: Den hekter på en ny verdi på begynnelsen av en liste)

(Eksempel: ι)

```
(defun iota (a b)
  (if (> a b)
      nil
      (cons a
            (iota (+ a 1) b))))
```

```
  (iota 7 4) ⇝ NIL
  (iota 4 7) ⇝ (4 5 6 7)
(iota 38 38) ⇝ (38)
```

(Eksempel: Slå sammen to lister)

```
(defun cat (list1 list2) ; meow
  (if (null list1)
      list2
      (cons (first list1)
            (cat (rest list1) list2))))
```

```
(cat nil '(a b c)) ⇝ (A B C)
(cat '(foo) '(bar baz)) ⇝ (FOO BAR BAZ)
```

(I virkeligheten ville man brukt `APPEND`)

(Eksempel: Sett inn element i liste)

```
(defun insert (elem i list)
  (if (= i 0)
      (cons elem list)
      (cons (first list)
            (insert elem
                    (- i 1)
                    (rest list)))))
```

```
(insert 'er 1 '(lister fine)) ⇝ (LISTER ER FINE)
```

(REPL)

(Du kan snakke med Lispen din via et REPL – en `READ-EVAL-PRINT`-loop)

(`READ`: Lisp leser det du skriver og bygger fine listestrukturer av det)

(`EVAL`: Lisp evaluerer uttrykket den leste)

(`PRINT`: Lisp presenterer stolt verdien den er kommet frem til så du skal få glede av den)

(Loop: Det hele gjentas)

(Slik ser et REPL ut)

```
CL-USER> (defun ! (n) (if (= n 0) 1 (* n (! (- n 1)))))
!
CL-USER> (! 5)
120
CL-USER> (! (! 5))
66895029134491270575881180540903725867527463331380
29810295671352301633557244962989366874165271984981
30815763789321409055253440858940812185989848111438
96500059649605212569600000000000000000000000000
CL-USER>
```

(Oppgaver)

La funksjonen $f : \mathbb{N} \to \mathbb{N}$ være definert ved

$$f(n) = \begin{cases} \frac{n}{2} & \text{hvis } n \text{ er partall} \\ 3n + 1 & \text{hvis } n \text{ er odde} \end{cases}$$

1. Skriv en funksjon `COLLATZ` som beregner f.
2. Skriv en funksjon `COLLATZ-LENGTH` som for et naturlig tall n returnerer det minste naturlige tallet k slik at $f^k(n) = 1$.
3. Vil `COLLATZ-LENGTH` alltid terminere?

```
(collatz-length 1) ⇝ 0
(collatz-length 4) ⇝ 2
(collatz-length 5) ⇝ 5
```

(Løsningsforslag)

```
(defun collatz (n)
  (if (evenp n)
      (/ n 2)
      (1+ (* n 3))))

(defun collatz-length (n)
  (if (= n 1)
      0
      (+ 1 (collatz-length (collatz n)))))
```