2
Funktionsorienteret programmering i Scheme.

Lisp og Scheme.
Listebegrebet i Lisp.
Funktionsdefinition og lambdaudtryk.
Navnebinding.
Iteration.
Første-klasses funktioner.
Closures som klasser.
Praktisk kørsel af Scheme.
Litteratur om Scheme.

Noter

I dette kapitel vil vi introducere programmeringssproget Scheme, og dermed Lisp-familien af sprog.

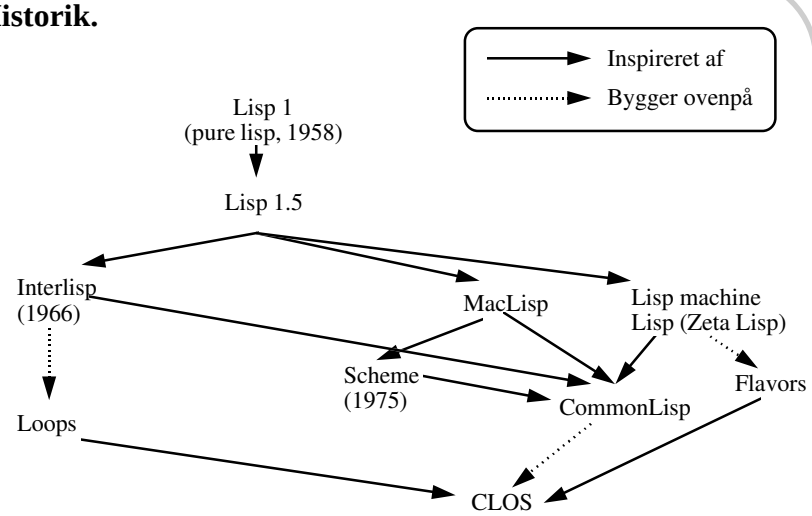
Vi vil endvidere vende tilbage til nogle af de emner, som blev diskuteret i det første kapitel, men her på en mere konkret og sprog-specifik måde.

Scheme: Et sprog i Lisp-familien.

- Lisp er det næstældste programmeringssprog efter Fortran.
- Lisp var det første funktionsorienterede programmeringssprog.
- Lisp udviklede sig hurtigt til at være et sprog, som kombinerede flere paradigmer:
  - Funktionsorienteret programmering.
  - Imperativ programmering.
  - Objekt-orienteret programmering.
- Der har de sidste 30 år været mange Lisp-dialekter, som idag må betragtes som uddøde.
- De dominerede Lisp-dialekter er idag Common Lisp og Scheme.
- Scheme kan siges at være en “Uncommon Lisp”.

Noter

Historik.



Noter

Denne slide giver kun et lille overblik over de væsentligste Lisp dialekter gennem tiden. I dette kursus ser vi først på Scheme, og senere på CLOS, som er en objekt-orienteret overbygning på Common Lisp.

Scheme er et meget overskueligt, rent og kraftfuldt sprog i Lisp familien. Common Lisp er derimod stort, og mindre rent, og det er næsten umuligt at få overblik over. I kraft af sin "rigdom" er Common Lisp dog et kraftfuldt sprog med mangfoldige funktioner og faciliteter, der gør god nytte i praktiske programudviklingssituationer.

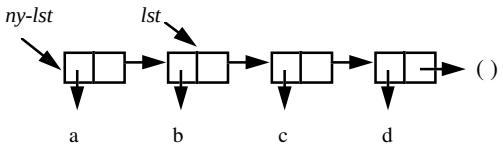
Scheme er defineret i en kort, men præcis sprograpport: *Den 4. reviderede rapport om det Algoritmiske Sprog Scheme* (titlen oversat).

Lister.

- Den originale, og den primære datastruktur i Lisp.
- Lister kan indeholde elementer af alle mulige, blandede typer.

Konstruktion:

(cons a *lst)*



Selektion:

(car ny-lst) = a

(cdr ny-lst) = lst

Egentlig liste:

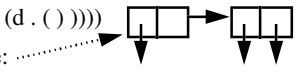
(a b c d)

Fuld dot-notation:

(a . (b . (c . (d . ( )))))

Uegentlig liste:

(a b . c)



- Et udtryk (et programfragment) er syntaktisk set selv en liste.
- Scheme slår **ikke** bro mellem sprogets listebegreb og den syntaktiske repræsentation af Scheme-programmer.

Noter

De strukturer, som kan konstrueres ved brug af cons, er i realiteten binære træer:

(cons left right).

Et sådant udtryk kaldes traditionelt et S-udtryk (symbolsk udtryk). Et S-udtryk, som via højre-grenen ender i den tomme liste ( ) opfattes som en egentlig liste (på øverste niveau). Elementerne i en egentlig liste kan naturligvis være hvad som helst, incl. atomer (tal og symboler), uegentlige lister, eller egentlige lister.

Det er også naturligt at omtale de primitive atomare typer: tal og symboler.

Scheme har et meget veludviklet talbegreb, som er overlegen i forhold til de fleste andre sprog.

Symboler er objekter, der er karakteriseret af (1) et navn og af (2) entydig identitet. Sidstnævnte betyder, at hver gang man refererer til et symbol med navn, f.eks., sym, får man fat i et bestemt objekt, hvis navn er “sym”. Dette er altså en modsætning til tekststrenge, hvor der kan eksistere vilkårlig mange forskellige objekter som er sammensat af tegnene ‘s’, ‘y’ og ‘m’. Ud over dette er symbolbegrebet i Scheme desværre ikke særlig veludviklet. I mere klassiske Lisp dialekter kan man tilknytte navngivne egenskaber (værdier) til symboler, kaldet *properties.*

Definition og anvendelse af funktioner.

- **Funktions-objekter: *Closures*.**

  (lambda (*parametre*) *body*)

  - Returnerer et funktions-objekt.
  - Lukker af over (indfanger) navnene i funktionsobjektets omgivelser:
    - Statisk navnebinding.
    - Giver en closure
      - Environment + syntaktisk form.

- **Funktionsanvendelse** (-kald):

  (e_1 e_2 ... e_n)

  - Alle udtryk i kaldet evalueres på en uniform måde.
  - Værdien af e_1 skal være et funktionsobjekt.
  - Værdierne af e_2 ... e_n kan være vilkårlige objekter.

- **Definitioner:**

  (define *name expression*)

  Formen af funktionsdefinition:

  (define *name* (lambda (*parameters*) *body*)) ~

  (define (*name parameters*) *body*)

- Globale definitioner: Binder navne i det globale environment.

PS4 - Scheme

Noter

Bemærk *closure begrebet*, som jo altså dækker over en funktion opfattet som et objekt. Dette objekt består dels af funktionens syntaktiske form, og de omgivelser (det environment = samling af navnebindinger), hvori funktionen er defineret.

Nederst til højre vises to forskellige former for funktionsdefinition:

(1) (define name (lambda (parameters) body))

og

(2) (define (name parameters) body)

Det er meget væsentligt at forstå at (1) er den oprindelige form, som er i overensstemmelse med den generelle form af define

(define name expression)

hvor 'expression' evalueres til en værdi, der bindes til navnet 'name'. Formen (2) er syntaktisk sukker for (1), og den har følgende to fordele:

Kortere og mindre indlejret.

Bestanddelen (name parameters) svarer til kaldsformen af funktionen.

Derfor bliver (2) næsten altid benyttet. Men vær sikker på at forstå, at hver gang I ser formen (2) er det blot en overfladisk omskrivning af formen (1). Scheme-systemet er fuldstændig blind for, om man bruger (1) eller (2). Så snart den ser en definition på formen (2) omformer systemet det internt til en definition af formen (1).

Navnebinding (1): Eksempel.

Navnebinding med sukker:

```
(define (solve a b c)
  (let ((det (- (* b b) (* 4 a c))))
    (cond ((> det 0) (list (/ (- (- b) (sqrt det)) (* 2 a))
                           (/ (+ (- b) (sqrt det)) (* 2 a))))
          ((= det 0) (list (/ (- b) (* 2 a))))
          (else '()))))
```

Navnebinding uden sukker (illustrerer brug af anonym funktion):

```
(define (solve a b c)
  ((lambda (det)
     (cond ((> det 0) (list (/ (- (- b) (sqrt det)) (* 2 a))
                            (/ (+ (- b) (sqrt det)) (* 2 a))))
           ((= det 0) (list (/ (- b) (* 2 a))))
           (else '())))
   (- (* b b) (* 4 a c))))
```


Noter

I eksemplet løses en andengradsligning, i forlængelse af det samme eksempel fra forrige kapitel.

Ovenstående illustrerer også **cond** og **list** funktionerne. Cond er et eksempel på et *betinget udtryk*, altså et udtryk der udvælges og beregnes på baggrund af beregning af én eller flere boolske udtryk.

Forneden ses hvordan navnebinding i bund og grund realiseres som et kald af en anonym funktion, noteret ved et lambda-udtryk.

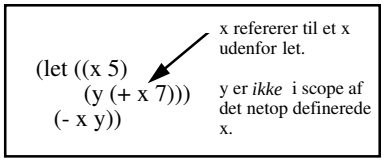
Navnebinding (2).

• Lokale bindinger af navne:

- Simultane definitioner

```
(let ((n1 v1)
      ...)
 body)        ~

((lambda (n1 ...) body) v1)
```

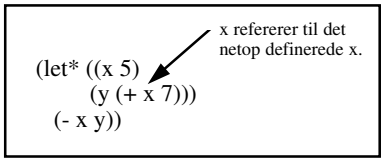
Ex



Hver navnebinding er gyldig i body

- Sekventielle definitioner

```
(let* ((n1 v1)
       (n2 v2)
       ...)
 body)              ~

(let ((n1 v1))
 (let* ((n2 v2)
        ...)
  body))
```



Hver navnebinding er gyldig "til højre" for bindingen.

Noter

Let, Let*, letrec er (ligesom) cond essentiel syntax ifølge schemerapporten. At det er "syntaks" betyder at let mv. ikke kan defineres som funktioner. I Lisp jargon kaldes sådanne også **special forms**. Man kan opfatte let mv. for syntaktiske abstraktioner. (let ((n v)) x) er således en syntaktisk abstraktion over ((lambda (n) x) v)). Syntaktiske abstraktioner kan i de fleste Scheme systemer defineres via makroer.

Navnebinding (3).

• Lokale bindinger af navne:

Gensidigt rekursive definitioner:

Eksempel:

```
(letrec  ((f1 (lambda (...) ... (f2 ...)) )
          (f2 (lambda (...) ... (f1 ...)) ))
  body)
```

Hver af navnebindingerne er gyldige i hele letrec-en.

Fortolkning af lokale 'defines':

```
(define (f ...)
  (define (f1 ...) body1)
  (define (f2 ...) body2)
 body)
```

~

```
(define (f ...)
  (letrec ((f1 (lambda (...) body1)
           (f2 (lambda (...) body2))
    body))
```


Noter

Man kan omhyggeligt overbevise sig om at der er forskel på kravet til letrec på den ene side, og let og let*, som introduceret på forrige slide.

I letrec skal f2 eksempelvis kunne refereres i kroppen af f1. Det betyder at scope af f2 ikke blot er kroppen af letrec, men også de udtryk, som det selv samt tidligere og efterfølgende navne bindes til. Derfor er letrec speciel og fundamental i den forstand at det kun vanskeligt kan realiseres af de andre former.

Iteration og hale-rekursion.

rekursiv løsning:

```
(define (list-length l)
  ; pre-betingelse: l er en egentlig liste
  (cond ((null? l) 0)
        ((pair? l) (+ 1
                      (list-length (cdr l))))))
```

halerekursiv løsning:

```
(define (list-length l)
  ; pre-betingelse: l er en egentlig liste
  (define (list-length-iter lst count)
    (if (null? lst)
        count
        (list-length-iter (cdr lst) (+ 1 count))))

  (list-length-iter l 0))
```


Noter

På denne slide viser vi et eksempel på iteration ved "almindelig" rekursion og ved halerekursion. Eksemplet er ikke det samme som i kapitel 1, men dog tilsvarende. Funktionerne ovenfor tæller antallet af elementer i en egentlig liste (jf. defintionen tidligere i dette kapitel).

I Scheme insisterer man på, at enhver implementation skal håndtere halerekursion på lager-effektiv vis.

Første klasses funktioner (1): Eksempler.

- Funktioner er af *første klasse* fordi:
  1. De kan overføres som parametre.
  2. De kan returneres som resultat.
  3. De kan lagres i datastrukturer.
- Scheme har *statisk binding af frie navne*:
  – Frie navne i en funktion bindes i funktiondefinitionens omgivelse, ikke i funktionskaldets omgivelse.

Eksempel 1:

```
(define (deriv f dx)
  (lambda (x)
    (/ (- (f (+ x dx)) (f x))
       dx)))
```

```
> (let ((res (deriv (lambda (x) (* x x x)) .0001)))
     (map res (list 1 2 3 4 5)))
(3.0003000099987354 12.000600010022566 27.00090001006572
48.00120000993502 75.00150000993244)
```

PS4 - Scheme

Noter

Generelt gælder der for alle typer af data, at disse er af 1. klasse hvis ovenstående tre krav (1—3) er opfyldt. Det er ikke overraskende at f.eks. tal er af første klasse. Men det er sikkert nyt for mange studerende, at funktioner er af 1. klasse.

Nederst ser vi et eksempel på, at vi kan returnere en funktion fra en funktion. Eksemplet illustrerer i Scheme hvordan vi numerisk kan differentiere en funktion af én reel variabel. Vi minder om, at netop dette eksempel blev diskuteret mere generelt i den første forelæsning om funktionsorienteret programmering.

Vi elaborerer videre på eksemplet på næste slide.

Første klasses funktioner (2): Eksempler.

Eksempel 2:

```
> (let ((f-list
         (map deriv (list (lambda(x) (* x x))
                          sin
                          log)
                    (list 0.001 0.001 0.001))))
    (ny-map f-list (list 2 4 6 8)))
```

Her laves en liste af funktioner.

```
((4.000999999999699 8.0010000000037 12.001000000005035 16.000999999988608)
(-0.416601415864859 -0.6532651107071796 0.9603098343597405 -0.14599468864062715)
(0.49987504165105445 0.24996875520755246 0.16665277932093048 0.12499218815120727))
```

```
(define (ny-map function-list value-list)
  ; anvend hver af funktionerne i function-list på value-list,
  ; og returner listen af resultat-lister.

  (if (null? function-list)
      '()
      (cons (map (car function-list) value-list)
            (ny-map (cdr function-list) value-list))))
```

PS4 - Scheme

Noter

I dette eksempel betegner f-list en liste af funktioner, som fremkommer ved at mappe funktionen deriv (fra forrige side) på funktionerne x*x, sin(x) og cos(x). Den udgave af map, vi anvender kalder først deriv på x*x og første element i listen af 0.001-er. Dernæst anvendes sinus på det andet af 0.001-erne, osv.

Mapning går generelt ud på at anvende en funktion på alle elementer i en liste, og at returnere listen af de resulterende funktionsværdier. Vi vil ikke diskutere emnet i større detalje her, idet mapning vil blive behandlet i stor detalje i kapitlet om højreordensfunktioner.

For at efterprøve resultatet har vi behov for at anvende hver af funktionerne i listen på hvert at tallene i en liste, her (2 4 6 8). Dertil programmerer vi en særlig funktion ny-map, som er vist i lille font forneden på denne slide.

Bemærk at det er muligt at lave en liste af funktioner. Dette illustrerer 3. punkt af kravet til 1. klasses objekter.

Lambda - Den Ultimative.

Førsteklasses funktioner med statisk navnebinding (closures) er lige så stærke som klasser.

"En klassedefinition" :

```
(define (class init-parametre)
  (let ((var1 value1)
        ...
        (varn valuen))
    (lambda (op)
      (cond ((eqv? op op1)  beregning af op1's værdi )
            ...
            ((eqv? op opm)  beregning af opm's værdi ))))
```

Indkapsling af datarepræsentation

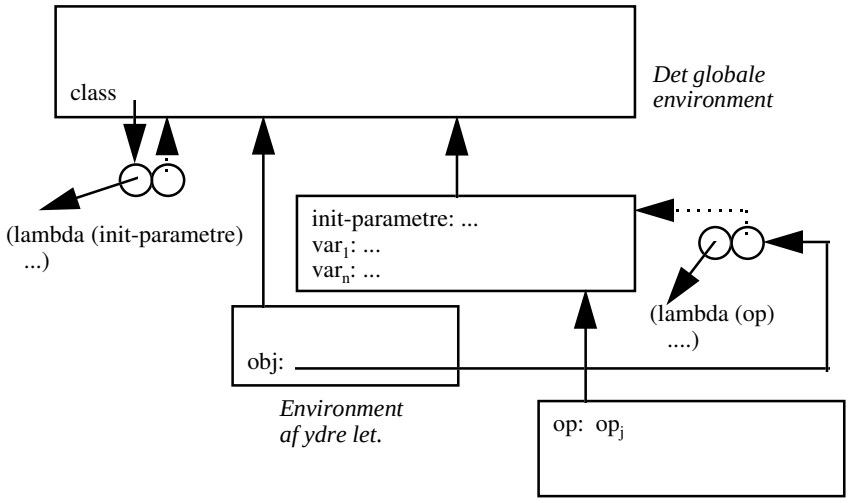
"Klasseinstantiering og operationskald" :

```
(let ((obj (class aktuelle-par)))
  (obj opj))
```

PS4 - Scheme

Noter

Nedenfor vises en model af, hvilke environments (activation records) og funktionsobjekter, der er tilstedet i det ovenstående.



Funktionen class er defineret i det globale environment. Dette er den anonyme funktion bag den ydre let også; denne kaldes, hvorved obj bindes til værdien af kaldet af class. Dette er et funktionsobjekt, som ses længst til højre. Hvert funktionsobjekt er karakteriseret af en syntaktisk del og af en reference til det environment, hvori det er defineret. (Obj opj) forårsager kaldet, som giver den nederste activation record.

Environment modellen, som vist ovenfor, er taget fra bogen "Structures and Interpretation of Computer Programs" af Abelson & Sussman, McGrawHill og MIT press.

Scheme og objekt-orienteret programmering (1).

Klasse uden nedarvning.

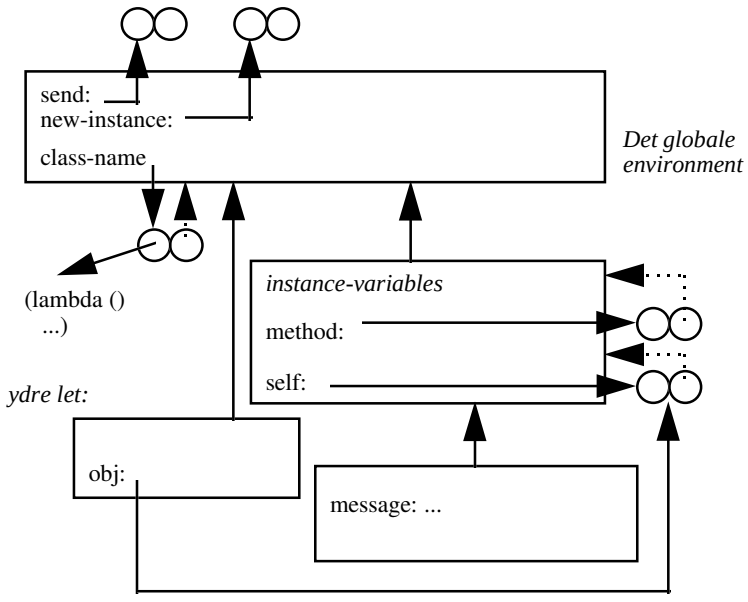
```
(define (class-name)
  (let ((instance-var init-value)
        ...)

    (define (method parameter-list)
      method-body)
    ...

    (define (self message)
      (cond ((eqv? message selector) method)
            ...

            (else (error "Undefined message" message))))

    self))
```

Anvendelse:

```
(let ((obj (new-instance class-name)))
  (send obj message parameters))
```

PS4 - Scheme

Noter



Ovenstående er et øjebliksbillede af ovenstående anvendelse af class-name. Man ser det environment som aktiveringen af selve klassen afstedkommer, og under det aktiveringen af den funktion, som "repræsenterer objektet" (self). Dernæst vil dette kald returnere én af metoderne, som vil blive kaldt af send.

Send og new-instance er vist på næste slide.

Scheme og objekt-orienteret programmering (2).

Uden nedarvning.

Instantiering:

```
(define (new-instance class)
  (class))
```

Metodeaktivering:

```
(define (send object message . args)
  (let ((method (object message)))
    (cond ((procedure? method) (apply method args))
          (else (error "Error in method lookup " method)))))
```


Noter

Her vises funktioner, som hhv. instantierer en klasse, og som kan sende en besked til et objekt. Vi kan opfatte disse som syntaktisk sukker, og som abstraktioner over de Scheme detaljer, som tillader os at opfatte funktioner og deres aktiveringer som hhv. klasser og objekter.

Primitivet procedure? returnerer hvorvidt et objekt er et procedure-objekt. Det ville have været lidt pænere at spørge ved hjælp af navnet function? men i Scheme kan man i lambdaudtryk godt lave sideeffekter -- derfor talen om procedurer.

Scheme og objekt-orienteret programmering (3).

Klasse med nedarvning.

```
(define (class-name)
  (let ((super (new-part super-class)))
    (let ((instance-variable init-value)
          ...)

      (define (method formal-parameter...)
        method-body)
      ...

      (define (dispatch message)
        (cond ((eqv? message 'selector) method)
              ...
              (else (method-lookup super message))))

      dispatch)))
```


Noter

Vi har på de forrige slides observeret at funktionsbegrebet i Scheme er stærkt nok til at simulere klasser, information hiding, instantiering, og metodeopslag samt -anvendelse. Simuleringen er meget ligefrem. Det eneste lidt kunstige element er funktionen self, som afbilder en besked i en metode; denne afbildning realiserer vi i et betinget udtryk.

Hvis man er "bidt af" de objekt-orienterede muligheder i Scheme kan man fortsætte lidt endnu. Hvis ikke, kan denne og de næste tre slides overspringes.

Vi fortsætter udforskningen af funktionsbegrebet i Scheme med henblik på at efterligne klasser fra OO programmeringssprog. Her vises en skabelon der tillader os at arve fra en superklasse. De understregede dele er nye ift. forrige skabelon.

Her vises en klasse, som instantierer et nyt objekt (en objektdel), som bindes til super.

Læg også mærke til dispatch (som også kunne kaldes self), som i denne udformning kalder et nyt "primitiv": method-lookup. Denne vises på næste side.

Scheme og objekt-orienteret programmering (4).
Med nedarvning.

```
(define (new-part class)
  (class))

(define (method-lookup object selector)
  (cond ((procedure? object) (object selector))
        (else (error "Inappropriate object in method-lookup: " object))))

(define (send object message . args)
  (let ((method (method-lookup object message)))
    (cond ((procedure? method) (apply method args))
          ((null? method) (error "Message not understood: " message))
          (else (error "Inappropriate result of method lookup: " method)))))
```

```
(define (object)
  (let ((super '())
        (self nil))

    (define (dispatch message)
      '())

    (set! self dispatch)
    self))
```


Noter

Her vises tre ny abstraktioner, med primær fokus på method-lookup. Læg mærke til, hvordan method-lookup aktiveres i send.

Nederst vises roden i klassehierarkiet, som ikke i den viste udformning har nogen metoder at tilbyde.

Scheme og objekt-orienteret programmering (5).

Med nedarvning og Smalltalk-agtig self.

Ny metode:

```
(define (set-self object-part)
   (set! self object-part)
   (send super 'set-self! object-part))
```

Instantiering:

```
(define (new-instance class)
 (let ((instance (class)))
  (virtual-operations instance)
  instance))

(define (virtual-operations object)
  (send object 'set-self! object))
```

Rod i klassehierarki:

```
(define (object)
  (let ((super '())
        (self nil))

    (define (set-self obj-part)
      (set! self obj-part))

    (define (responds-to message)
      (let ((method (method-lookup self message)))
        (if method #t #f)))

    (define (dispatch message)
      (cond ((eqv? message 'set-self!) set-self)
            ((eqv? message 'responds-to?) responds-to)
            (else '())))

    (set! self dispatch)
    self))
```

PS4 - Scheme

Noter

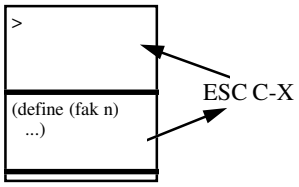
Nu udarter tingene sig. Vi ønsker at self i hvert objekt-del skal referere til "hele objektet". Dette arrangerer vi ved brug af assignments i metoden set-self, som vi antager alle klasser har. I et objekt med mange dele, starter vitual-operations kæden af assignments til self.

Hvis man vil læse mere om alt dette, kan man konsulterer rapporten "Simulation of Object-oriented Concepts and Mechanisms in Scheme" af Kurt Nørmark, R-90-01, fra AUC.

Scheme implementationer.

- ELK.
  - Hvor: /pack/elk
  - Startes hvordan: /pack/elk/bin/scheme *(udførbar fil).*
  - Dokumentation: /pack/elk/elk-2.2/doc Specielt filen kernel/kernel.ps.
- MIT Scheme (virker pt. ikke på Solaris 2 maskiner).
  - Hvor: /pack/mitscm
  - Startes hvordan: /pack/mitscm/bin/mitscm *(udførbar fil).*
  - Dokumentation: /pack/mitscm/doc (Lange og store manualer...).

Hvordan man kører Scheme gennem Emacs:



ESC C-X

(define (fak n)
...)

M-x run-elk
M-x run-mit-scheme

Disse emacs kommandoer loades fra /user/normark/public/ps4/schemeload.el

PS4 - Scheme

Noter

Det forskellige, og mere specifikke dokumentation af de to Scheme systemer findes på programmørkontoret (under Mit Scheme og under ELK -- håber jeg).

ELK er ud over at være en Scheme fortolker også et udvidelsessprog: Et Extension Language Kit. Det er muligt fra ELK at slå bro til C, og hvad deraf følger. Se dokumenterne i /pack/elk/elk-2.1/doc/ex kataloget og i /pack/elk/elk-2.1/doc/paper kataloget.

Litteratur om Scheme.

- **Definerende dokument**:
  - *Revised[4] Report on the Algorithmic Language Scheme.*
- **Bøger**:
  - Abelson & Sussman: *Structure and Interpretation of Computer Programs*. (Introducerende lærebog).
  - Springer & Friedman: *Scheme and the Art of Programming*.
  - Dybvig: *The Scheme Programming Language*.
  - Friedman & Felleisen: *The Little Lisper*.
- **Tidsskrifter**.
  - Lisp and Symbolic Computation (Kluwer).
  - Lisp Pointers (ACM).
- **Konferencer**.
  - ACM conference on Lisp and Functional Programming (hver andet år).

Noter