Edellä annettiin ymmärtää että lupauksen $p =$ `(delay` e`)` lausekkeelle e laskettaisiin arvo a *uudelleen joka kerran* kun p lunastetaan operaatiolla `(force` p`)`.

Oikeasti a lasketaan *vain kerran* ensimmäisen lunastuksen yhteydessä, talletetaan silloin muistiin ja katsotaan muistista seuraavilla lunastuskerroilla laskematta sitä uudelleen.

Laiska ohjelmointi on siis *melko* tehokasta ja funktionaalista:

- Arvoja a ei lasketa moneen kertaan...
- ...mutta kirjanpito vie aikaa ja tilaa.
- Mahdollistaa *inkrementaaliset* funktionaaliset ohjelmat...
- ...mutta `delay` päivittää muistia eikä siis ole itse toteutettavissa funktionaalisesti.

Eräs huono puoli ohjelmoijan vastuulle jäävässä laiskuudessa on, että hän joutuu kirjoittamaan samasta funktiosta (kuten `filter`) erilliset ahke laiskat versiot.

Toinen mahdollisuus olisi jättää laiskuus ohjelmointikielen toteutuksen vastuulle, jolloin s olisi ohjelmoijalle näkymätöntä. Esimerkiksi Has [T99] on tällainen kieli. Myös Scheme-tulkki oli mahdollista toteuttaa laiskana [AS96,§4.2].

Tällöin lausekkeet olisivat näkymättömien `delay`-operaatioiden sisällä, joihin ohjelmaa suoritettaessa kohdistettaisiin näkymättömiä `force`-operaatioita *mutta vasta silloin kun niide arvoa tarvitaan*.

Suoritusmekanismissa varsinkin kalvojen II.5.3 *funktiokutsu* muuttuisi:

Ennen kutsun (lauseke$_0$ `...` lauseke$_k$) *jokainen* lauseke laskettiin arvoonsa.

Nyt vain *ensimmäinen* (eli lauseke$_0$) `force`-laskettaisiin (jotta selviäisi kutsuttava fun Muut lausekkeet (sen parametrit) `delay`-viivytettäisiin tehtävässä kopiossa.

Perusfunktiot (`car`, `cdr`, `cons`, `+`, `-`,...) `force`-las parametrinsa voidakseen muodostaa tuloksensa

Kolme suoritusmekanismia:

Ahkera (eager, strict, applicative-order, call-by-value) on se mitä Scheme (ja useimmat muut kielet) käyttää: argumenttien arvot lasketaan ennen niiden lähettämistä kutsuttavaan funktioon.

Laiska (lazy, non-strict, call-by-need) luonnosteltiin edellä: argumentit lähetetään lupauksina pystyä laskemaan niiden arvot jos kutsuja tarvitsee.

Arvot lasketaan (korkeintaan) kerran `delay`-muistikirjanpidon avulla.

Kanoninen (canonical, normal-order, call-by-name) on muuten kuin laiska, mutta muistikirjanpidon sijasta `delay` tulkitaan `lambda`-lyhenteeksi, eli argumentit lähetetään lupauksina, mutta jokainen lunastus aiheuttaa uuden suorituksen.

Tämä mekanismi on kalvojen II.6.5 lambdakalkyylin "oikea" eli "ulkoa sisään" -sievennysjärjestys.

Schemellä jokainen näistä on kohtuullisen vaivatonta.

Tarkastellaan ohjelmointiesimerkkinä kalvojen II
"ulkoa sisään" -sieventäjää `beta`.

Siinä on käytetty

- näiden kalvojen laiskan suorituksen ideaa
- toteutettuna näiden kalvojen `delay/force`-mekanismilla.
- kalvojen II.5.5 sanakirjaideaa
- toteutettuna kalvojen II.8.7 assosiaatiolistoil
- kalvojen II.8.6 ideaa symboleista tunnuksina niminä
- toteutettuna erillisinä apurutiineina.
- poistumista virhetilanteessa kesken rekursion
- toteutettuna kalvojen II.7.5 järjestelmäjatke

(Listaus jatkuu yli kalvorajojen.)

```
; Tyypittömän lambda-laskennan beta-redusoija.
; 30.09.2002  Matti Nykänen
;
; Lambda-laskennan <termi> on (pitkässä muodossa):
; * muuttuja (VAR <symboli>).
;   Ulkoasu (lyhyessä muodossa) on pelkkä <symboli>, ei kuitenkaan '^'.
; * vakio (CON <datum>).
;   Ulkoasuna ovat <numero>t, <merkkijono>t ja ne <symboli>t joita ei ole
;   abstrahoitu ja jotka eivät ole (tulostukseen varattua) muotoa
;   'x<indeksi>'.
; * abstraktio (ABS <symboli> <termi>).
;   Ulkoasu on (^ <symboli_1> ... <symboli_n> <termi>).
; * sovellus (APP <termi_1> <termi_2>).
;   Ulkoasu on (<termi_1> <termi_2> <termi_3> .... <termi_n>).
;   Myös ylimääräiset sulut eli (<termi>) sallitaan ulkoasuna.
; * funktioarvo (FUN <symboli> <termi> <ympäristö>):
;   * <symboli> on <symboli> joka nimeää parametrinä toimivan muuttujan
;   * <termi> on funktion määrittelevä lauseke jossa <symboli> on vapaa
;   * <ympäristö> antaa vastineet <termi>n muille vapaille muuttujille.
;   Funktioarvoja ei syötetä eikä tulosteta, joten niillä ei ole omaa
;   ulkoasua.
;
; Tmämä <ympäristö> taas on assosiaatiolista <symboli>selta
; muuttujannimeltä vastaavalle <arvo>lle. Tämä <arvo> on <lupaus>
; yrittää sieventää vastaava <termi> funktioarvoksi.
;
; Jos sievennysyritys epäonnistuu, eikä saada haluttua funktioarvoa,
; niin palautetaan (pitkässä muodossa) kesken jäänyt sovellus
; (<muu kuin funktioarvo> <termi_1'> ... <termi_k'>)
; missä kukin <termi_i'> on vastaava argumentti<termi_i> mutta täysin
; normalisoituna.
; Tämä voidaan sitten tulostaa (lyhyessä muodossa).

(define (beta syote-termi)
  (define (tee-vakio datum)
    (list 'CON datum))
  (define (on-vakio? termi)
    (eq? (car termi) 'CON))
  (define vakio-datum cadr)
  (define (tee-muuttuja symboli)
    (list 'VAR symboli))
  (define (on-muuttuja? termi)
    (eq? (car termi) 'VAR))
  (define muuttuja-symboli cadr)
  (define (tee-abstraktio symboli termi)
    (list 'ABS symboli termi))
  (define (on-abstraktio? termi)
    (eq? (car termi) 'ABS))
  (define abstraktio-symboli cadr)
  (define abstraktio-termi caddr)
  (define (tee-sovellus termi_1 termi_2)
    (list 'APP termi_1 termi_2))
  (define (on-sovellus? termi)
    (eq? (car termi) 'APP))
  (define sovellus-termi_1 cadr)
  (define sovellus-termi_2 caddr)
  (define (tee-funktioarvo symboli termi ymparisto)
    (list 'FUN symboli termi ymparisto))
  (define (on-funktioarvo? termi)
    (eq? (car termi) 'FUN))
  (define funktioarvo-symboli cadr)
  (define funktioarvo-termi caddr)
  (define funktioarvo-ymparisto cadddr)
  (define ymparisto-tyhja '())
  (define (ymparisto-lisaa symboli lupaus ymparisto)
    (cons (cons symboli
                lupaus)
          ymparisto))
  (define (ymparisto-nouda symboli ymparisto)
    (cdr (assq symboli ymparisto)))
  (define (funktioarvoksi termi
                          ymparisto)
    (cond
      ((on-vakio? termi)
       termi)
      ((on-muuttuja? termi)
       (force (ymparisto-nouda (muuttuja-symboli termi)
                               ymparisto)))
      ((on-abstraktio? termi)
       (tee-funktioarvo (abstraktio-symboli termi)
                        (abstraktio-termi termi)
                        ymparisto))
      ((on-sovellus? termi)
       (let ((funktio_1 (funktioarvoksi (sovellus-termi_1 termi)
                                        ymparisto)))
         (if (on-funktioarvo? funktio_1)
             (funktioarvoksi
              (funktioarvo-termi funktio_1)
              (ymparisto-lisaa (funktioarvo-symboli funktio_1)
                               (delay (funktioarvoksi
                                       (sovellus-termi_2 termi)
                                       ymparisto))
                               (funktioarvo-ymparisto funktio_1)))
             (tee-sovellus
              funktio_1
              (normaalimuotoon (funktioarvoksi (sovellus-termi_2 t
```

```
                                        ymparisto))))))
    ((on-funktioarvo? termi)
     termi)))
(define (normaalimuotoon termi)
  (cond
    ((on-vakio? termi)
     termi)
    ((on-sovellus? termi)
     (tee-sovellus (normaalimuotoon (sovellus-termi_1 termi))
                   (normaalimuotoon (sovellus-termi_2 termi))))
    ((on-funktioarvo? termi)
     ; Hack: totetaan uusi <vakio> vanhan <muuttuja>n tilalle, koska
     ; niiden lopullisissa tulostusasuissa ei ole eroa.
     (let ((alias-symboli (uusi-symboli)))
       (tee-abstraktio
        alias-symboli
        (normaalimuotoon (funktioarvoksi
                          (tee-sovellus termi
                                        (tee-vakio alias-symboli))
                          ymparisto-tyhja)))))))
(define uusi-symboli
  ; Tämä (ja vain tämä) funktio on tilaperustainen.
  (let ((indeksi -1))
    (lambda ()
      (set! indeksi (+ indeksi 1))
      (string->symbol (string-append "x" (number->string indeksi))))))
(define (ulkoinen-muuttuja? datum)
  (and (symbol? datum)
       (not (eq? datum '^))))
(define (pitka-termi lyhyt-termi)
  (call-with-current-continuation
   (lambda (virhe)
     (let pidenna ((lyhyt lyhyt-termi)
                   (muuttujat '()))
       (cond
         ((ulkoinen-muuttuja? lyhyt)
          (let ((alias (assq lyhyt muuttujat)))
            (cond
              (alias
               (tee-muuttuja (cdr alias)))
              ((let ((tekstina (symbol->string lyhyt)))
                 (and (char=? (string-ref tekstina 0)
                              #\x)
                      (string->number
                       (substring tekstina
                                  1
                                  (string-length tekstina)))))
               (virhe
                "Kirjoita <symboli>set vakiot muuten kuin 'x<numero>'!"))
              (else
               (tee-vakio lyhyt)))))
         ((and (list? lyhyt)
               (not (null? lyhyt)))
          (cond
            ((null? (cdr lyhyt))
             (pidenna (car lyhyt) muuttujat))
            ((eq? (car lyhyt) '^)
             (if (null? (cddr lyhyt))
                 (virhe "Ei <abstraktori>a ilman <muuttuja>a!")
                 (let abstrahoi ((lyhyet (cdr lyhyt))
                                 (muuttuneet muuttujat))
                   (cond
                     ((null? (cdr lyhyet))
                      (pidenna lyhyet muuttuneet))
                     ((ulkoinen-muuttuja? (car lyhyet))
                      (let ((alias (uusi-symboli)))
                        (tee-abstraktio alias
                                        (abstrahoi (cdr lyhyet)
                                                   (cons
                                                    (cons (car l
                                                          alias)
                                                    muuttuneet))
                     (else
                      (virhe
                       "Abstrahoida voi vain <symboli>sia muuttu
            (else
             (let sovella ((lyhyet (cdr lyhyt))
                           (pitkat (pidenna (car lyhyt)
                                            muuttujat)))
               (if (null? (cdr lyhyet))
                   (tee-sovellus pitkat
                                 (pidenna lyhyet muuttujat))
                   (sovella (cdr lyhyet)
                            (tee-sovellus pitkat
                                          (pidenna (car lyhyet)
                                                   muuttujat))))
         ((or (number? lyhyt)
              (string? lyhyt))
          (tee-vakio lyhyt))
         (else
          (virhe
           "Anna <vakio>na <numero>, <merkkijono> tai muu <symbo
(define (lyhyt-termi pitka-termi)
  (cond
    ((on-muuttuja? pitka-termi)
     (muuttuja-symboli pitka-termi))
    ((on-vakio? pitka-termi)
     (vakio-datum pitka-termi))
```

```
    ((on-sovellus? pitka-termi)
     (let sovella ((pitka (sovellus-termi_1 pitka-termi))
                   (lyhyt (list (lyhyt-termi
                                 (sovellus-termi_2 pitka-termi)))))
       (if (on-sovellus? pitka)
           (sovella (sovellus-termi_1 pitka)
                    (cons (lyhyt-termi (sovellus-termi_2 pitka))
                          lyhyt))
           (cons (lyhyt-termi pitka)
                 lyhyt))))
    ((on-abstraktio? pitka-termi)
     (cons '^
           (let abstrahoi ((pitka pitka-termi))
             (if (on-abstraktio? pitka)
                 (cons (abstraktio-symboli pitka)
                       (abstrahoi (abstraktio-termi pitka)))
                 (list (lyhyt-termi pitka)))))))
(let ((syotetty (pitka-termi syote-termi)))
  (if (string? syotetty)
      syotetty
      (lyhyt-termi (normaalimuotoon (funktioarvoksi syotetty
                                                    ymparisto-tyhja)))))
```

(Listaus jatkui yli kalvorajojen.)

II.8.9 Parametrilistan otto

- `(lambda` (x_1 ... x_n) ...) antaa funktion joka vaatii *tasan* n argumenttia kuten kalvoilla II kerrottiin.

- `(lambda` (x_1 ... x_n . y) ...) antaa funktion vaatii *ainakin* n argumenttia.

  Kutsussa

  `((lambda` (x_1 ... x_n . y) ...) a_1 ... a_m

  (missä siis $m \geq n$) kukin parametri x_i saa ed arvokseen argumentin a_i. Lisäksi parametri y arvokseen loput parametrit listana (a_{n+1} ..

- `(lambda` y ...) antaa funktion joka hyväksyy *kuinka monta argumenttia hyvänsä*; pakollis parametreja x_i ei enää ole (eli $n = 0$), on va jäännösparametri y.

- `(lambda () ...)` antaa funktion joka *ei hyväksy yhtään argumenttia*.

  Matemaattisesti parametritön funktio on pelkkä vakio.

  Ohjelmoinnissa sillä voi luoda suorituskelpoisen ohjelmanosan (ns. *thunk*) jonka voi (esim.) välittää parametrina suoritettavaksi myöhemmin.

Nämä parametrilistanotaatiot voi ajatella *sovitusoperaationa*: funktiota kutsuttaessa sen parametrilistaa ja sille lähtevien argumenttien listaa verrataan toisiinsa siten, että

- `()` on "ei enää argumentteja"
- symboli y on "kaikki loput argumentit"
- pari $(x_i \ . \ P)$ on "ensimmäinen argumentti olkoon nimeltään x_i; jatka sitten seuraavien argumenttien sovittamista loppuihin parametreihin P".

(Parametri- ja argumenttilistojen ei tarvitse olla todellisia fyysisiä listoja.)

II.8.10 Parametrilistan anto

Meillä on (i) funktio f ja (ii) lista $L = (a_1 \ \ldots a_m)$ haluamme tehdä kutsun $(f \ a_1 \ \ldots a_m)$.

(Kutsu $(f \ L)$ olisi siis väärin koska siinä f saisi ainoana parametrinaan *listan* L, vaikka se halua parametreinään listan L *alkiot* $a_1, \ldots, a_m$.)

Kirjastofunktiokutsu `(apply` $f \ L$`)` tekee mitä halutaan: kutsuu funktiota f argumentteina list L alkiot.

Esimerkiksi `o1` määrittelee funktioiden yhdistäm $f \circ g$ kun g hyväksyy yhden argumentin, `o` taas kaikille g:

```
(define (o1 f g)
  (lambda (x1)
    (f (g x1))))

(define (o f g)
  (lambda y
    (f (apply g y))))
```

II.9 Tilaperustaiset piirteet

Reclaimer, spare that tree!
Take not a single bit!
It used to point to me,
Now I'm protecting it.
It was the reader's CONS
That made it, paired by dot;
Now, GC, for the nonce,
Thou shalt reclaim it not. `/usr/games/fortune`

Tähän asti on pysytelty Schemen funktionaalisesti puhtaassa osassa, jossa voitiin pitää

- aliohjelmia matemaattisina funktioina ja
- ohjelman suoritusta lausekkeen sievennyksenä.

Nyt jatketaan lisäämällä piirteitä joilla voidaan ilmaista laskentaprosesseja myös muuten kuin funktionaalisesti.

Puhtaasta osasta puuttuu joitakin piirteitä:

- Kalvoilla II.8.8 kerrottiin `delay`-kirjanpidosta tallettaa lupausten arvoja muistiin, mutta ei kerrottu miten.

  Muistin käyttöön viitattiin myös kalvoilla II.8 II.8.5.

  Muisti ja sen muokkaaminen on siis lisättävä

- Ulkomaailman kanssa kommunikointi on muutakin kuin "parametrit sisään, arvo ulos

  Syöte ja tulostus on siis lisättävä.

Mukaan tulee

- laskulaitteen muistin
- ympäröivän maailman

tilan käsite.

Sievennysmallissa ei ollut ohjelman muokattavis olevaa tilaa, oli vain sievennyksen nykyinen väli

II.9.1 Ympäristöoppia

Aloitetaan muistin mallintaminen tarkentamalla sitä tapaa, miten `define`- tai `lambda`-määriteltyjen muuttujien arvoista pidetään kirjaa [AS96,§3.2].

Kehys (frame) varastoi äärellisen kuvauksen muuttujilta arvoille:

- Kuvaus on määritelty vain äärellisen monelle muuttujalle.
- Arvo jolle muuttuja kuvataan on joko nuoli kalvojen II.8.2 laatikkoon tai kalvojen II.6.4 `lambda`-lausekkeen arvona saatu funktio (josta enemmän kalvoilla II.9.2).

Näitä kuvauksia voi ajatella kalvon II.8.7 `assq`-assosiaatiolistaksi.

Lisäksi kehyksessä on nuoli *ympäröivään* kehykseen (ellei tämä kehys ole se *globaali*).

Intuitiivisesti kuvaus kertoo *paikallisten* muuttujanmäärittelyiden vastaavat arvot, ja ympäröivät kehykset muiden.

Ympäristö (environment) on nuoli kehykseen, pikemminkin siitä alkava *pino jaettuja kehyk*

Jokaiseen suorituskelpoiseen lausekkeeseen e liittyy ympäristö joka antaa arvon jokaiselle muuttujan esiintymälle s silloin kun lauseket suoritetaan [AS96,kuva 3.1]:

1. Ensin katsotaan onko ensimmäisen kehyk kuvaus määritelty muuttujalle s. Jos on, saadaan haluttu arvo.
2. Muuten jatketaan sitä ympäröivään kehy ja yritetään uudelleen.
3. Jos kehykset loppuvat eikä arvoa löytyny kyseessä virhe: muuttuja s ei olekaan määritelty lausekkeessa e.

Kalvojen II.5 käsitteillä

- kehys on se jono pareja $\langle nimi, arvo \rangle$ jonka *yk* funktionkutsu lisää sanakirjaan
- ympäristö on sanakirja, paitsi että sama keh voi olla jaettu usean eri sanakirjan kesken.

(Itse asiassa toteutuksen ei tarvitse selata ympäristöjä suoritusaikana: käännösaikana voida laskea etukäteen kuinka kaukaa arvo löytyy.)

II.9.2 Laskentasäännöt

Scheme-lausekkeiden arvot lasketaan eli *evaluoidaan* seuraavasti kalvojen II.9.1 ympäristöissä.

Evaluointi tapahtuu aina *nykyisessä* ympäristössä, ja voi muuttaa tai vaihtaa sitä.

Tulkin komentorivin suoritus tapahtuu globaalissa ympäristössä – eli siinä jolla ei ole ympäröivää. Kaikki kirjastofunktiot on määritelty siinä ympäristössä (mutta aluksi ei muuta).

Lambda-lauseke kalvolta II.6.4 – eli kaikki funktionmäärittelyt – evaluoidaan perustyypin `procedure` alkioksi eli *proseduuriksi*.

Proseduurissa on kaksi osaa [AS96,kuva 3.2]:

- Nuoli nykyiseen ympäristöön jossa se luotiin.

  Proseduurin mukana kulkee siis aina viite sen synty-ympäristöön. Näin toteutuvat kalvojen II.6.3 leksikaaliset näkyvyyssäännöt.

- Viite proseduurin parametreihin ja runkoon.

Kalvojen II.8.5 funktioiden `eq?`-vertailu testaa (yksinkertaisimmillaan) näitä osia.

Määrittely globaalissa ympäristössä (`define`

- lisää tämän globaalin ympäristön kuvauks uuden muuttujan s

- ja sille arvon a joka saadaan evaluoimalla e tässä globaalissa ympäristössä

Jos s oli jo kuvauksessa, niin sen arvoksi vaihdetaan a. [AS96,kuvat 3.2 ja 3.4]

Proseduurikutsu (e_0 ... e_k) suoritetaan nykyi ympäristössä p seuraavasti:

1. Evaluoidaan lausekkeille $e_0, \ldots, e_k$ arvot $a_0, \ldots, a_k$ nykyisessä ympäristössä p.

2. Tarkistetaan että a_0 on proseduuri jolla o parametrit $x_1, \ldots, x_k$ (tai jokin muu kalvo II.8.9 muoto). Olkoon sen synty-ympärist ja runko α.

3. Luodaan uusi kehys r jonka ympäröijä on jonka kuvaus on $x_i \mapsto a_i$. Evaluoidaan α ympäristössä r ja raportoidaan tulos sitä odottavaan ympäristöön p. [AS96,kuvat ja 3.5]

Loppukutsuille kalvoilta II.7.1 odottava p poistetaan odottajien ketjusta *ennen* evaluointia, ja raportoidaan vastaus suoraan näin löytyneeseen ympäristöön, eli siihen joka odotti ympäristöä p.

Järjestelmäjatke kalvoilta II.7.5 on juuri tämä evaluoitavien lausekkeiden α "kuka odottaa arvoa keneltä" -ketju.

Määrittely paikallisessa ympäristössä kalvoilta II.6.2 voidaan suorittaa kuten globaali, paitsi että

- lisäykset tehdään siihen kehykseen r jossa paikalliset määrittelyt sisältävä runko α evaluoidaan ja
- samaa muuttujaa s ei saa lisätä toistamiseen.

Toinen tapa tehdä paikallisia määrittelyjä on kalvojen II.6.4 `let`-rakenteen rekursiivinen versio `letrec` [D96,§4.3;KCR98,§4.2.2].

- Parhaillaan luotavan paikallisen muuttujan alustuslausekkeessa *voi mainita* samaa ja m parhaillaan luotavia muuttujia.

  Jopa vuorottainenkin rekursio on siis mahdo
- Niiden arvoja *ei voi käyttää* koska niiden alustuslausekkeita ei vielä ole suoritettu. [KCR98,§5.2.2]
- Siis mainintojen on oltava sellaisissa paikois jotka suoritetaan vasta, kun kaikki alustukse suoritettu:

  Kalvojen II.6.4 `lambdan` tai kalvojen II.8.8 `de` sisällä.
- Kalvojen II.6.2 paikalliset määritelmät ovat vaihtoehtoista syntaksia `letrec`-rakenteelle.

Ehtolause kalvolta II.5.4 evaluoidaan vastaavasti kuin ennen: ensin ehto evaluoidaan nykyisessä ympäristössä näillä uusilla säännöillä, sitten tuloksen perusteella valitaan haara josta jatketaan evaluointia.

Heittomerkki kalvolta II.8.4 tuottaa edelleen (rakenteisen) vakion.

Scheme-ohjelmien syntaksi on samanlainen kuin rakenteisen datan. Dataa voidaankin suorittaa ohjelmana kirjastokutsulla [KCR98,§6.5]

(`eval` d p)

joka yrittää suorittaa rakenteisen data-alkion kuten

```
'(lambda (f x) (f x x))
```

ympäristössä p.

Mahdollisia ympäristöjä p ovat:

`(scheme-report-environment 5)` eli 5. standardi.

`(null-environment 5)` eli pelkkä kieli, ei kirjastoja.

`(interaction-environment)` eli nykyisen toteutuksen komentoriviympäristö (standardin ehdottama muttei vaatima).

II.9.3 Arvon vaihtaminen

Nyt kun kalvot II.9.2 kertovat missä kalvojen II. ympäristöistä kulloinkin ollaan, voidaan esitellä Scheme-kielen operaatio muuttujalle annetun ar muuttamiseen [AS96,§3.1;D96,§4.5;KCR98,§4.1

Kun *sijoitusoperaatio*

(set! s e)

suoritetaan nykyisessä ympäristössä p, niin:

- Evaluoidaan e ympäristössä p arvoonsa a.
- Haetaan se kuvaus l josta muuttujalle s katsottaisiin arvo nykyisessä ympäristössä p.

  Sijoituksen kohdetta s *ei* evaluoida: sen on s oltava sijoitettavan muuttujan nimi eikä mit muuta.
- Päivitetään kuvausta l siten, että s kuvautu nyt arvolle a.

  Muutos näkyy juuri niihin evaluointiaskeleisii joissa viitataan muuttujan s arvoon kuvauks l.

Operaatio `set!` palauttaa *määrittelemättömän arvon*; se tehdään *muistia muuttavan sivuvaikutuksensa* vuoksi, ei minkään tuloksen laskemiseksi.

Konventio: Muistia muuttavien operaatioiden nimet päättyvät huutomerkkiin!

Määrittelemättömällä "arvolla" ei voi laskea eteenpäin. Siksi kalvojen II.6.4 `lambda`-lausekkeen runkoon saakin kirjoittaa useita suoritettavia lausekkeita peräkkäin, kuten kalvolla II.7.1 jo vihjattiin. Nämä lausekkeet evaluoidaan kirjoitusjärjestyksessä: viimeinen funktion arvon selvittämiseksi, sitä edeltävät vain sivuvaikutustensa tähden.

Jos halutaan kirjoittaa useita lausekkeita peräkkäin, niin tarvitaan muista lohkorakenteisista kielistä tuttua rakennetta `begin` t_1`;...;`t_n `end`.

Sen Scheme-muoto on lauseke (`begin` t_1 `...` t_n) mikä on pelkkä järjestelmän tarjoama lyhennysmerkintä lausekkeelle

((`lambda ()` t_1 `...` t_n))

missä lausekkeista tuotetaan ensin kalvojen II.8.9 "thunk" jota kutsutaan heti sen jälkeen.

Esimerkki [AS96,§3.2.3]:

`withdraw` nostaa *globaalilta* pankkitililtä `balance` summan `amount` [AS96,§3.1.1].

`new-withdraw` tekee pankkitilistä *paikallisen*: `lamb` evaluoidaan ympäristössä johon `let` on luo tilin paikallisena muuttujana [AS96,§3.1.1].

`make-withdraw` *konstruoi* uusia paikallisia pankki sisempi (näkyvä) `lambda` evaluoidaan siinä ympäristössä joka luotiin ulompaa (piilevää kutsuttaessa [AS96,kuvat 3.6–3.10].

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))

(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))

(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

Kirjallisuutta

AS96 H.Abelson&G.J.Sussman: *Structure and Interpretation of Computer Programs. Second Edition.* MIT Press, 1996.

AHV95 S.Abiteboul,R.Hull,V.Vianu: *Foundations of Databases.* Addison Wesley, 1995.

AHU83 A.V.Aho,J.E.Hopcroft&J.D.Ullman: *Data Structures and Algorithms.* Addison Wesley, 1983.

B01 I.Bratko: *Prolog Programming for Artificial Intelligence. Third Edition.* Addison Wesley, 2001.

CM87 W.F.Clocksin&C.S.Mellish: *Programming in Prolog. Third, Revised and Extended Edition.* Springer-Verlag, 1987.

CM98 G.Cousineau&M.Mauny: *The Functional Approach to Programming.* Cambridge University Press, 1998.

DD98 C.J.Date&H.Darwen: *Foundation for Object/Relational Databases: The Third Manifesto.* Addison Wesley, 1998.

DEC96 P.Deransart,A.Ed-Dbali&L.Cervoni: *Prolo The Standard. Reference Manual.* Springer-Verlag, 1996.

D96 R.K.Dybvig: *The Scheme Programming Language: ANSI Scheme. Second Edition.* Prentice Hall, 1996.

H92 D.Harel: *Algorithmics. Second Edition.* Ac Wesley, 1992.

H97 J.R.Hindley: *Basic Simple Type Theory.* Cambridge University Press, 1997.

KCR98 R.Kelsey, W.Clinger&J.Rees (toim.): Revi Report on the Algorithmic Language Sche *Higher-Order and Symbolic Computation* 11(1), 1998 **tai** *ACM SIGPLAN Notices* 3 1998.

L87 J.W.Lloyd: *Foundations of Logic Programming. Second, Extended Edition.* Springer-Verlag, 1987.

P02 B.C.Pierce: *Type Theory and Programming Languages.* MIT Press, 2002.

S98 S.Slade: *Object-Oriented Common Lisp*. Prentice Hall, 1998.

SS86 L.Sterling&E.Shapiro: *The Art of Prolog.* MIT Press, 1986.

S97 B.Stroustrup. *The C++ Programming Language. Third Edition.* Addison-Wesley, 1997.

T99 S.Thompson: *Haskell: The Craft of Functional Programming. Second Edition.* Addison-Wesley, 1999.