II.8 Listat

Tähän mennessä olemme käsitelleet sellaisia Scheme-tyyppejä, joiden arvot ovat osiin jakamattomia: totuusarvoja, numeroita ja funktioita.

Lisp (List processing) -kieliperheessä (ja koko funktionaalisessa ohjelmoinnissa) *listat* ovat keskeinen rakenteinen tyyppi, Scheme-kielessä jopa miltei ainoa [AS96,§2.2;KCR98,§6.3.2;D96,§6.2].

Piilevästi tyypitettynä kielenä Schemen listat ovat *heterogeenisia*: sama lista voi sisältää *alkio*inaan eri tyyppisiä arvoja.

Common Lisp: sisältää Scheme-kieltä rikkaamman kokoelman rakenteisia tyyppejä, kuten tietueet ja oliot.

II.8.1 Tyhjä lista

Yksinkertaisin lista on *tyhjä* eli sellainen joka ei sisällä yhtään alkiota. Se on vielä itse jakamato vakio, mutta siitä aloitetaan listojen induktiivine määrittely.

Schemessä tyhjää listaa merkitään `'()`.

Jo kalvoilla II.5 nähdyn heittomerkin `''` tarkoitukseen palataan kalvoilla II.8.4. (DrScheme-toteutuksen kielitaso "full scheme" näyttää kuitenkin sallivan sen jättämisen pois.)

Testi `null?` tutkii onko sen argumentti tyhjä list jotakin muuta.

Common Lisp: Tyhjällä listalla on myös nimi `n`

Tyhjä lista vastaa vakiota `#f` eli on ainoa "epätosi" arvo.

Lainausmerkki ei ole pakollinen.

II.8.2 Parinmuodostaja

Kirjastofunktio

`(cons` uusialkio vanhalista`)`

konstruoi *uuden* listan joka on saatu liittämällä annetun vanhan listan alkuun annettu uusi alkio.

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))     (‡)
```

tuottaa "sisältä ulos" luettuna

1. 1-alkioisen listan liittämällä tyhjän listan alkuun alkion `4`

2. 2-alkioisen listan alkioista `3` ja `4` liittämällä askeleessa 1 tehdyn listan alkuun alkion `3`

3. listan alkioinaan `2`, `3` ja `4`

4. ja lopuksi listan alkioinaan `1`, `2`, `3` ja `4`.

Kirjastofunktiokutsu `(list 1 2 3 4)` tuottaa saman tuloksen ja on selkeämmän näköistä. Sille voi antaa *halutun määrän parametreja;* mekanismiin palataan kalvoilla II.8.9.

Listoja voi visualisoida *laatikko-nuoli-notaatiolla* (box-and-pointer notation):

- Jokainen jakamaton tietoalkio on omassa laatikossaan.

- Kutsu `(cons` a l`)` luo uuden laatikon jossa o *kaksi lokeroa*. Sitä kutsutaankin *pariksi* ja se perustietotyyppi (`pair?`).

- Kutsu asettaa edelliseen/vasemmanpuoleise lokeroon nuolen, joka osoittaa uuteen lisätty alkioon a.

  Jos a on itsekin lista, niin nuoli osoittaa siih laatikkoon, jolla a alkaa. Silloin lista a on syntyneen listan alkio.

- Kutsu asettaa jäkimmäiseen/oikeanpuoleise lokeroon nuolen, joka osoittaa siihen laatikk jolla kasvatettava lista l alkaa.

  Jos l on tyhjä lista, niin lokeron merkitäänki jääneen tyhjäksi.

Esimerkiksi lauseke (‡) luo ensin parin jonka vas jäsen osoittaa vakioon 4 ja oikea on tyhjä, sitte parin jonka vasen jäsen on 3 ja oikea osoittaa edelliseen pariin,. . . [AS96,kuva 2.4].

Parin voi muodostaa vaikka jälkimmäinen argumentti ei olisikaan lista: `(cons 1 2)` luo parin jonka edellinen jäsen on `1` ja jälkimmäinen `2` [AS96,kuva 2.2]. Tulosta kutsutaan *pisteytetyksi pariksi* (dotted pair).

Aitoja (proper) listoja ovat (tyhjä lista itse ja) ne parirykelmät joiden jälkimmäisiä nuolia seuraamalla päädytään lopulta tyhjään listaan.

Esimerkiksi lauseke (‡) tuottaa aidon listan.

Kirjastofunktio `list?` tutkii onko sen argumentti aito lista.

Aidon n-alkioisen listan *tulostusasu* on

$$\texttt{(alkio}_1\ \texttt{alkio}_2\ \texttt{alkio}_3\ \texttt{...alkio}_n\texttt{)}$$

eli sulkuihin välilyönnein eroteltuina.

Epäaitoja (improper) listoja ovat puolestaan ne joissa päädytään lopulta johonkin muuhun jakamattomaan vakioon.

Lauseke `(cons 1 2)` tuotti epäaidon listan.

Epäaidon n-alkioisen listan tulostusasu on

$$\texttt{(alkio}_1\ \texttt{alkio}_2\ \texttt{alkio}_3\ \texttt{...alkio}_{n-1}\ \texttt{.}\ \texttt{alkio}_n\texttt{)}$$

eli päättävän parin piste tulee näkyviin.

II.8.3 Parin osat

Kalvoilla II.8.2 synnytetyn parin sisältöä voi kurl kirjastofunktiolla

`car` edelliselle lokerolle (joka osoitti ensimmäisee alkioon)

`cdr` jälkimmäiselle lokerolle (joka osoitti loppulistaan).

Esimerkiksi `(car (cdr` l`))` palauttaa arvonaan lis *toisen* alkion: `cdr` palauttaa listan l ilman sen ensimmäistä alkiota, ja `car` ottaa siitä vuorosta ensimmäisen alkion.

Koska tällaisia listankaivuuoperaatioita käytetää usein, on vakiokirjastossa

`(define (c`$s_1 \ldots s_k$`r x) (c`s_1`r (c`s_2`r ...(c`s_k`r x`

missä kukin suunta s_i on joko `a` ("alas") tai `d` ("oikealle") ja $1 \leq k \leq 4$.

Erityisesti n. alkion kaivaa `cad`^{n}`r`.

Nyt meillä on (minimi)välineet *lista*rekursioon.

Tehdään esimerkkinä alkiot kahdentava listakuvaus

$$(a_1\ a_2 \ldots a_n) \mapsto (a_1\ a_1\ a_2\ a_2\ \ldots a_n\ a_n).$$

Aitojen listojen määritelmässä oli kaksi tapausta:

Perustapauksena tyhjä lista.

Nyt ()↦().

Induktiivisena tapauksena (pisteytetty) pari jossa on ensimmäinen alkio a ja loppulista l.

Nyt $(a\ .\ l) \mapsto (a\ .\ (a\ .\ l'))$ missä l' on induktiivisesti kahdennettu loppulista l.

Tapaukset muuntuvat suoraan Scheme-ohjelmaksi:

```
(define (dup l)
  (if (null? l)
      '()
      (cons (car l)
            (cons (car l)
                  (dup (cdr l))))))
```

Vastaavanlainen rekursio toistuu usein listoja käsiteltäessä.

II.8.4 Listavakiot

Kalvojen II.8.3 listarekursioesimerkkiä `dup` testat
on työlästä kirjoittaa argumentteja kuten `(1 2 3`
kalvojen II.8.2 `cons`-operaatioilla kuten
lausekkeella (‡).

Kunpa ne voisi myös *kirjoittaa* tulostusasussaan

Juuri näin voikin tehdä *jos laittaa eteen heittomerkin* `''`!

(Vertaa kalvot II.8.1–II.8.2.)

Heittomerkki siis kertoo että seuraava lauseke o
vakio tulostusasussaan.

Kalvojen II.4.1 ja II.4.2 totuusarvo- ja
numerovakioihin ei vielä tarvittu heittomerkkiä
(mutta se sallitaan).

Kalvojen II.4.3 symbolityypin vakioihin heittome
kuitenkin jo tarvittiin erottamaan ne kalvon II.5
muuttujista.

Heittomerkki siis pysäyttää arvon laskennan siih
vakioon, joka löytyy heittomerkin takaa. Se vak
olla rakenteinenkin.

Ilmaus `'`e on itse asiassa lyhenne ilmaukselle `(quote` e`)`, missä `quote` ei olekaan funktio, koska funktiona se laskisi ensin argumenttinsa e arvon; sen sijaan tämän erikoisilmauksen arvoksi otetaankin e sellaisenaan.

Esimerkiksi siis funktiossa

```
(define (lainaan x)
  (list 'x
        x))
```

ylempi `'x` on symbolinen vakio `x`

alempi `x` on parametrin `x` arvo

joten kutsun `(lainaan 5)` arvo on lista `(x 5)`.

Näin e on *"lainausmerkkien sisällä"* eikä sitä pidetä enää suoritettavana Scheme-ohjelmana vaan rakenteisena datana.

Joskus on rakennettava lista, joka koostuu osin vakioista ja osin normaalisti lasketuista arvoista Tätä varten on *heittomerkki "nurinpäin"* eli `` ``e (lyhenteenä ilmaukselle `(quasiquote` e`)`).

Nurin väännetyn heittomerkin sisällä *pilkku* lausekkeen e edessä eli `,`e (lyhenteenä ilmauksel `(unquote` e`)`) *pakenee* lainausmerkkien sisältä lausekkeen e ajaksi, eli laskee sen arvon ja liittä synnytettävän listan (tms.) alkioksi tähän kohta

Yhdistelmä `,@`e (lyhenteenä ilmaukselle `(unquote-splicing` e`)`) toimii samoin, paitsi että lausekkeen e arvona saatava lista lisätään alkio alkiolta eikä alilistana.

Siis `` `(1 ,x ,@y) `` synnyttää listan jonka ensimmä alkiona on `1`, toisena muuttujan `x` arvo ja kolma neljäs, viides,. . . alkio saadaan muuttujan `y` arvo olevasta listasta.

Heittomerkillä nurinpäin on kätevää ilmaista sel rakenteita, jotka ovat melkein mutta ei ihan vak

Kirjastokutsuilla `cons` ja `list` saa kuitenkin selkeämpää ohjelmakoodia heti kun laskettavia osasia on useita.

II.8.5 Samansisältöisyys ja samuus

Lukujen yhtäsuuruuden eli samuuden tutkimiseen oli operaatio =.

Rakenteisille arvoille samansisältöisyys ja samuus ovat eri käsitteitä. Siksi niille on eri operaatiot:

`(equal?` a b`)` palauttaa toden intuitiivisesti silloin kun lausekkeiden a ja b arvot ovat *tulostusasultaan* samat.

Se siis tutkii ovatko rakenteiset arvot *sisällöiltään yhteneväiset*.

Esimerkiksi määrittelyillä

```
(define eka '(1))
(define toka (cons 1 '()))
(define vika toka)
```

antaa `(equal? eka toka)` tuloksen `#t`.

`(eq?` a b`)` palauttaa "intuitiivisesti" toden silloin lausekkeiden a ja b yhteisenä arvona on *sam elementti* kalvojen II.8.2 (muistinkäyttöä kuvaavassa) laatikko-nuoli-piirroksessa.

Se siis tutkii onko kyseessä *yksi ja sama rak*

- Vakio `()` eli "katkaistu nuoli" on yksikäsitteinen (vaikka siitä onkin useita kopioita).

  Siis `(eq? '() '())` on `#t`.

- Vakioilla `#t` ja `#f` on yksikäsitteiset omat erilliset järjestelmän varaamat laatikot.

  Siis `(eq? #t #t)` on `#t`.

- Operaatio `cons` synnyttää aina uuden laa joten `eq?` kertoo pareista ovatko ne synty samalla kertaa.

  Siis `(eq? eka toka)` on `#f` mutta `(eq? toka vika)` on `#t`.

`(eqv?` a b`)` on sellainen `eq?` joka toimii järkevästi kalvojen II.4.2 numerovakioilla (ja myöhemn selitettävillä merkkivakioilla): sama vakio ku saattaa sijaita useassa eri laatikossa.

Funktio `equal?` voidaan määritellä funktion `eqv?` taas perusfunktion `eq?` avulla.

Yleensä `equal?` lienee luonnollisin vaihtoehto.

Kalvojen II.4.3 symbolityypin data-alkioiden erikoispiirre oli, että ne ovat erilliset täsmälleen silloin kun niiden (tulostus- eli) kirjoitusasut ovat erilaiset:

- `(eq? snafu fubar)` laskee ensin muuttujien `snafu` ja `fubar` arvot, ja vertaa sitten niitä.
- `(eq? 'snafu 'snafu)` on *aina* `#t`.
- `(eq? 'snafu 'fubar)` on *aina* `#f`.

Kalvojen II.8.2 laatikko-nuoli-notaatiossa jokaisella symbolilla on siis oma yksikäsitteinen laatikkonsa.

Tämä erotteleva käyttäytyminen periytyy myös vertailuihin `eqv?` ja `equal?`.

Funktioilla (eli `lambda`-lausekkeiden arvoilla) ei o

- (toteutusriippumatonta) tulostusasua
- (vielä) vastinetta laatikko-nuoli-notaatiossa

joten miten niitä verrataan toisiinsa?

Yleensä ohjelmointikielissä ei mitenkään, mutta koska nyt niiden halutaan olevan arvoja siinä m muunkin datan, täytyy tähänkin (algoritmisesti ratkeamattomaan!) ongelmaan ottaa *jokin* kant

Matematiikassa funktiot ovat samat jos ne sa samoilla argumenteilla samat arvot.

Ohjelmoinnissa funktiot ovat samat jos ne *käyttäytyvät* samoin laskennan kuluessa.

Schemessä jokainen `lambda`-suoritus luo uuden funktio(laatiko)n ja kahta funktiota pidetään sa (ainakin) silloin kun kyseessä on sama luomus (vertaa `cons`): määritelmillä

```
(define foo (lambda (x) x))
(define bar (lambda (x) x))
(define baz bar)
```

antaa `(eq? bar baz)` arvon `#t` mutta `(eq? foo ba` määrittelemätön (yleensä `#f`). Samoin `eqv?` ja `e`

II.8.6 Symbolit datassa

Kalvojen II.8.4 heittomerkki jakoi Scheme-lausekkeen kahteen osaan:

Ulkopuolella oleva lausekkeen osa suoritetaan koko lausekkeen lausekkeen arvon laskemiseksi.

Tällä puolella esiintyy kalvojen II.5.1 muuttujia.

Sisäpuolella oleva lausekkeen osa otetaan (rakenteisena) datana koko lausekkeen arvoa laskettaessa.

Tällä puolen esiintyy kalvojen II.4.3 symbolityypin vakioita.

On harmillista että (Lisp-historiallisista syistä)

- datan symboleilla on sama syntaksi kuin suoritettavan lausekkeen muuttujilla
- datan listoilla on sama syntaksi kuin suoritettavilla lausekkeilla

vaikka ne ovat eri käsitteitä.

Kalvojen I.3 derivointiongelman ratkaisevassa Scheme-ohjelmassa [AS94,§2.3.2] symboleita (+
käytetään

tunnuksina erottamaan eri tyyppiset lausekkee
toisistaan

niminä kohdealueen käsitteille.

Tällöin ohjelmakoodi voidaan edelleen lukea reseptinä vastaavalle symbolinmanipulointitehtä

"Syötelauseke `exp` on muotoa

vakio numerona

muuttuja symbolina

yhteenlasku `(+ addend augend)` missä `addend` ja `augend` ovat myös muotoa `exp`

kertolasku `(* multiplier multiplicand)` missä `multiplier` ja `multiplicand` ovat myös muotoa `exp`

ja palasten käsittely on eristetty omiin apurutiineihinsa." (Tosin Scheme ei ole yhtä "läpinäkyvää" kuin (esim.) kalvojen I.3 Prolog.)

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type -- DERIV" exp))))

(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2) (list '+ a1 a2))

(define (make-product m1 m2) (list '* m1 m2))

(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))

(define (addend s) (cadr s))

(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))
```


II.8.7 Assosiaatiolistat

Joskus halutaan ohjelmassa liittää datassa tava
(syöte)symbolille jokin "arvo" jota ei kalvojen I
mukaisesti järjestelmä tee. Tämä kirjanpidon vo
ohjelmoija hoitaa suoraan listalla pareja

$$L = ((\text{avain}_1 \ . \ \text{arvo}_1) \ \ldots (\text{avain}_n \ . \ \text{arvo}_n)$$

jota kutsutaan *assosiaatiolistaksi* [KCR98,§6.3.2

Syötesymbolia `key` listan L mukaan vastaava ar
löytyy kirjastokutsulla (`assq 'key` L) joka palau
ensimmäisen parin (`key .` arvo) jos sellainen löy
ja muuten `#f`. Arvon antamiseksi tai päivittämis
riittää siis uuden parin lisääminen listan alkuun.

`assq` vertaa hakuavainta listan avaimiin kalvojen
testillä `eq?`, joten se on luotettava vain
symboliavaimille.

`assv` vertaa testillä `eqv?`, joten se on luotettava
(merkki- ja) numeroavaimille.

`assoc` vertaa testillä `equal?`, joten se on luotetta
kaikille (jopa rakenteisille) avaimille.

Kalvojen II.5.5 sanakirjat olivat assosiaatiolistoj

II.8.8 Laiskat listat

Sift the twos and sift the threes,
The sieve of Eratosthenes.
When the composites sublime
The numbers that remain are Prime. *Anon.*

Tutustutaan Scheme-ohjelmien *laiskaan* (lazy, non-strict) suoritukseen [AS96,§3.5] esimerkkinä *Eratostheneen seula* alkulukujen tunnistamiseen.

Olkoon tehtävänä "listaa kaikki alkuluvut annettuun rajaan $n \in \mathbb{N}$ saakka".

1. Tee järjestetty lista $2, 3, 4, \ldots, n$ kaikista seulottavista luvuista.

2. Listan ensimmäinen alkio on seuraava alkuluku; poista se ja kaikki sen monikerrat listasta. Jatka kunnes lista on tyhjä.

   Siis 2 on ensimmäinen alkuluku; listaan jäävät vain parittomat luvut $3, 5, 7, 9, 11, \ldots$.

   Seuraavalla kierroksella löytyy toiseksi alkuluvuksi 3; listaan jäävät $5, 7, 11, \ldots$.


```
(define (filter pred unfiltered)
  (cond ((null? unfiltered)
         '())
        ((pred (car unfiltered))
         (cons (car unfiltered)
               (filter pred
                       (cdr unfiltered))))
        (else
         (filter pred
                 (cdr unfiltered)))))

(define (sieve numbers)
  (if (null? numbers)
      '()
      (cons (car numbers)
            (sieve (filter (lambda (number)
                             (not (zero? (remainder number
                                                    (car numbers))
                             (cdr numbers))))))

(define (numbers-between low high)
  (if (> low high)
      '()
      (cons low
            (numbers-between (+ low 1)
                             high))))

(define (primes-to limit)
  (sieve (numbers-between 2 limit)))
```

Vaan entäpä jos tehtävänä onkin "listaa n ensimmäistä alkulukua"?

Sama menetelmä kävisi jos tietäisimme etukäte kuinka pitkän seulalistan tarvitsisimme jotta siih jäisi lopuksi (ainakin) n lukua.

Tai jos *jatkaisimme seulalistaa tarpeen mukaan*

Laiska lista on

joko tyhjä lista `()` (kuten tavallista)

tai pari (**alkio . lupaus**) missä

alkio on epätyhjän laiskan listan ensimmäinen alkio (kuten tavallista) mutta

lupaus (promise) [D96,§5.7;KCR98,§4.2.5] onkin parametritön *funktio* joka kutsuttaessa palauttaa seuraavan palasen listaa – joko seuraavan alkio/lupaus-parin tai loppumerkkinä tyhjän listan.

Erikoisilmaus `(delay` e`)` tekee lausekkeesta e lupauksen (eikä siis laske sen arvoa nyt) jonka lunastaminen myöhemmin aiheuttaa lausekkeen e suorituksen.

Kirjastofunktio `(force` p`)` lunastaa nyt lupauksen p eli suorittaa sen lausekkeen e josta p aikanaan tehtiin.

Näin aluksi voi ajatella että `(delay` e`)` on lyhenne kalvojen II.8.9 lausekkeelle `(lambda () ` e`)` ja että `(define (force` p`) (`p`))`; tätä tarkennetaan myöhemmin.

Laiskoilla listoilla ohjelmointikuri on seuraava:

Alkion lisääminen (eli "`cons`") hoituu siten, et a on lisättävä alkio ja e lauseke joka tuottai loppulistan, niin `(cons` a `(delay` e`))`.

Tätä varten määritellään usein makro (`cons-stream` [AS96,§3.5.1]).

Loppulistan ottaminen (eli "`cdr`") parista q h siten, että `(force (cdr` q`))`.

Tätä varten määritellään usein funktio [AS96,§3.5.1]

```
(define (stream-cdr p) (force (cdr p)))
```

Ensimmäisen alkion ottaminen (eli "`car`") parista q hoituu operaatiolla `car` kuten enne – sitähän ei ole "luvattu".

Tätä varten määritellään silti usein funktio [AS96,§3.5.1]

```
(define stream-car car).
```

Koska nyt seulalistaamme pidennetään vain tarpeen mukaan, voimme abstrahoida ylärajan kokonaan pois ja ohjelmoida *äärettömän pitkillä* numerolistoilla $2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \ldots$; $3, 5, 7, 9, 11, \ldots$; $5, 7, 11, \ldots; \ldots$

```
(define (filter pred unfiltered)
  (if (pred (car unfiltered))
      (cons (car unfiltered)
            (delay (filter pred
                           (force (cdr unfiltered)))))
      (filter pred
              (force (cdr unfiltered)))))

(define (sieve numbers)
  (cons (car numbers)
        (delay (sieve (filter (lambda (number)
                                (not (zero? (remainder number
                                                       (car numbers)))))
                              (force (cdr numbers)))))))

(define (numbers-from low)
  (cons low
        (delay (numbers-from (+ low 1)))))

(define primes
  (sieve (numbers-from 2)))

(define (take n stream)
  (if (zero? n)
      '()
      (cons (car stream)
            (take (- n 1)
                  (force (cdr stream))))))
```

Kyllä ääretönkin lista (tms.) tietokoneeseen mahtuu, kunhan (i) sillä on äärellinen kuvaus ja (ii) siitä luetaan kerrallaan vain äärellinen pätkä!

- `(numbers-from low)` palauttaa parin

  $\texttt{from}_{\texttt{low}}$ = `(low .` lupaus parista $\texttt{from}_{\texttt{low+}}$

  eli äärettömän listan `(low low+1 low+2 ...`

- `(filter pred unfiltered)` palauttaa parin jo

  `car` on listan `unfiltered` ensimmäinen sellai alkio joka toteutti ehdon `pred` ja

  `cdr` on lupaus suodattaa loputkin listasta jo tarpeen.

  Ensimmäisen alkion löytämiseksi saatetaan joutua lunastamaan listan `unfiltered` lupauk

- `(sieve numbers)` palauttaa parin jonka

  `car` on seuraava löydetty alkuluku ja

  `cdr` on lupaus poistaa sen monikerrat listast `numbers` ja seuloa tulosta edelleen, jos halutaan vielä lisää alkulukuja.

- Muuttuja `primes` saa arvokseen äärettömän alkulukulistan, joka alkaa luvulla 2 ja jatkuu lupauksena seuloa seuraavatkin jos on pakk

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.* Ad 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.