II.9.4 Parin osien muuttaminen

Kalvoilla II.9.3 annettiin operaatio `set!` muuttujan arvon muuttamiseen.

Myös kalvojen II.8.2 pareille on vastaavat operaatiot [AS96,§3.3.1;D96,§6.2;KCR98,§6.3.2]:

`set-car!` edelliselle [AS96,kuvat 3.12–3.13] ja

`set-cdr!` jälkimmäiselle lokerolle [AS96,kuva 3.15].

Nämä ovat tavallisia kirjastofunktioita:
(`set-car!` e_1 e_2) evaluoi molemmat argumenttinsa e_1 ja e_2 arvoihinsa a_1 ja a_2 tarkistaa että a_1 on (nuoli) pari(in), ja korvaa sen `car`-lokeron arvo(on vievällä nuole)lla a_2.

Jos esimerkiksi kalvojen II.9.1 ympäristöt olisivat kalvojen II.8.7 assosiaatiolistoja niin (`set!` s e) olisi lyhenne lausekkeelle (`set-cdr!` (`assq` 's p) e).

Näillä funktioilla saa aikaan yllätyksiä:

```
(define a '(x))
(define b a)
(define c '(x))
(define (surprise)
  (set-car! b 'y))
(define (sour-price)
  (set-cdr! c c))
```

Kutsun (`surprise`) jälkeen muuttujan `b` arvo onl
(`y`) — mutta niin on myös muuttujan `a` koska r
jakoivat saman muistirakenteen!

Kutsun (`sour-prize`) jälkeen muuttujan `c` arvona
kehällinen pari (`x . (x . (x . ...))`)!

Muistiin kirjoittaminen aiheuttaa sellaisia piiloriippuvuuksia, joiden jälkeen Scheme-ohjelm toimintaa ei enää voi järkeillä yhtälö- vaan tilaperustaisesti.

II.9.5 Yksiulotteiset taulukot

Kalvolla II.9.4 annettiin funktiot parin lokeroiden sisältöjen muuttamiseksi.

Silloin pari on *yksiulotteinen taulukko* jossa on 2 paikkaa nimeltä `car` ja `cdr`.

Schemessä on mahdollista muodostaa myös pidempiä taulukkoja eli *vektoreita* (`vector?`) joille on seuraavanlaisia kirjastofunktioita [D96,§6.6;KCR98,§6.3.6]:

`(vector` e_1 `...`e_n`)` palauttaa vektorin v, jossa on n alkiota: i. alkio on lausekkeen e_i arvo.

`(vector-length` v`)` palauttaa montako alkiota vektorissa v on: yllä siis n.

`(make-vector` n e`)` palauttaa vektorin, jossa on n alkiota: jokainen on lausekkeen e (kerran laskettu) arvo.

Alustaja e voidaan myös jättää pois: silloin arvot jäävät määrittelemättömiksi.

`(vector-ref` v i`)` palauttaa vektorista v sen paik
i olevan alkion: paikkojen numerointi alkaa
arvosta 0 (ja päättyy siis arvoon
`(vector-length` v`)` -1).

`(vector-set!` v i e`)` muuttaa vektorissa v sen p
i (osoittamaksi) alkioksi lausekkeen e arvo

`(vector->list` v`)` palauttaa listan vektorin
sisältämistä alkioista samassa järjestyksess
ensimmäisenä paikan 0 alkio, toisena paika
jne.

`(list->vector` l`)` palauttaa vektorin listan l
sisältämistä alkioista samassa järjestyksess
paikassa 0 ensimmäinen alkio, paikassa 1 t
alkio, jne.

`(vector-fill!` v e`)` muuttaa vektorin v kaikkien
paikkojen sisällöksi lausekkeen e (kerran
lasketun) arvon.

Vektorivakiot voidaan kirjoittaa kuten kalvojen
listavakiot kunhan eteen lisätään merkki `#`: `'#(a`
jne.

II.9.6 Merkkijonot ja merkit

Olemme kohdanneet merkkijonotyypin (`string?`) [D96,§6.5;KCR98,§6.3.5] vakioita kalvojen II.8.6 ja II.9.3 virheilmoituksissa. Ne ovat (tuttuun tapaan) tekstiä kaksinkertaisten lainausmerkkien välissä, siis `"juuri niin"`.

Jos merkkijonovakioon haluaa kaksinkertaisen lainausmerkin, on sen eteen laitettava takakenoviiva: \".

Jos taas takakenoviivan, on se kahdennettava: \\.

Merkkijonoja voi käsitellä kuten kalvojen II.9.5 vektoreita, paitsi että ne voivat sisältää vain yksittäisiä *merkkejä*: `string`, `make-string`, `string-length`, `string-ref`, `string-set!`, `string-fill!`, `string->list` ja `list->string`.

Uusia kirjastofunktioita ovat:

(`string-copy` v) palauttaa uuden erillisen kopion merkkijonosta v.

(`string-append` v_1 ...v_k) palauttaa merkkijonon, joka on saatu yhdistämällä jonot $v_1, \ldots, v_k$ peräkkäin.

(`substring` v p q) palauttaa merkkijonon, joka sisältää merkkijonon v merkit paikasta p al paikkaan $q-1$ saakka.

(`number->string` n b) palauttaa kalvon II.4.2 numeron n kirjoitusasun merkkijonona.

Kantaluku b voi olla 2, 8, 10 tai 16; sen puuttuessa oletetaan tuttu 10.

(`string->number` v b) tekee saman toisin päin, t palauttaa `#f` ellei onnistu.

(`symbol->string` s) palauttaa symbolin s tulostusasun merkkijonona.

(`string->symbol` v) tekee saman toisin päin, eli merkkijonoa v vastaavan symbolin.

Tämän avulla voidaan luoda sellaisia (esim erikoismerkkejä sisältäviä) symboleita joita kukaan ei ole voinut (vahingossakaan) kirjo

Yksittäiset merkit ovat myös oma tyyppinsä (`char?`) [D96,§6.4;KCR98,§6.3.4].

Merkkityypin vakiot alkavat yhdistelmällä `#\` joten `#\a` on merkki 'a' kun taas pelkkä `a` on samannimisen symbolin esiintymä.

Välilyöntimerkki kirjoitetaan `#\space` ja rivinvaihtomerkki `#\newline`.

Kalvojen II.8.5 funktiolla `eq?` ei voi luotettavasti vertailla merkkejä (koska sama merkki voi olla numeroiden tapaan useassa eri kalvojen II.8.2 laatikossa): sen sijaan niille on funktiot `char`o`?` missä o on vertailu `=`, `<`, `>`, `<=` tai `>=`.

Vertailusta voi tehdä myös riippumattoman SUURISTA ja pienistä kirjaimista (kuten 'A' vs. 'a') lisäämällä `-ci` ennen vertailua o.

Samat vertailut löytyvät myös merkkijonojen sanakirjajärjestyksille. Samoin kalvojen II.8.5 `equal?` toimii myös merkeille ja merkkijonoille.

Muita merkkien luokittelupredikaatteja ovat `char-alphabetic?`, `-numeric?`, `-whitespace?` sekä `-upper/lower-case?`.

Muunnosfunktioita ovat `char-upcase`, `char-downcase`, `char->integer` ja `integer->char`.

II.9.7 Tulostus

*Perus*tulostukseen on 4 kirjastorutiinia [D96,§7.2;KCR98,§6.6.3]:

`(write` d`)` tulostaa tietoalkion d sellaisena että syötteenlukurutiini `read` tuloksen lukisi, se muodostaisi tulosteesta sellaisen Scheme-a d', joka olisi mahdollisimman paljon alkupe alkion d kaltainen.

Tätä pyrkimystä kutsutaan `read-write-`*invarianssiksi*.

Se merkitsee, että `write` jättää merkkijonovakioiden ympärille lainausmerki merkkien eteen `#\`, jne.

`(display` d`)` tulostaa tietoalkion d ihmiselle miellyttävämmässä muodossa piilottamalla `read`-ohjausinformaation.

`(newline)` vaihtaa riviä tulostuksessa.

`(write-char` c`)` tulostaa merkin c. Esimerkiksi `(write-char #\newline)` vaihtaa sekin riviä.

Perustulostus tapahtuu *oletustulosporttiin* (yleensä `stdout`). Sen voi kuitenkin ohjata myös tiedostoon:

`(with-output-to-file` v `(lambda () ` e`))` yrittää avata kirjoittamista varten tiedoston jonka nimenä on merkkijono v. Jos se onnistuu, niin sitten suoritetaan e siten, että em. tulostusoperaatiot kohdistuvatkin avattuun tiedostoon oletusportin sijaan.

Jos lausekkeesta e palataan normaalisti arvolla a, niin avattu tiedosto suljetaan, ja koko lauseke palauttaa saman arvon a.

Jos lausekkeesta paettiin kalvon II.7.5 `call/cc`-kutsulla, niin tiedosto jää auki odottamaan mahdollista paluuta, ainakin siihen saakka kunnes paluu ei ole enää mahdollista.

`(call-with-output-file` v `(lambda (`g`) ` e`))` tekee saman operaation siten, että voidaan tulostaa sekä oletusporttiin että tiedostoon: lauseketta e suoritettaessa g on portti avattuun tiedostoon.

Lausekkeessa e voidaan tulostaa porttiin g (oletusportin sijasta) kutsuilla `(write` d g`)` jne., eli antamalla portti viimeisenä lisäparametrina.

`(current-output-port)` palauttaa tämänhetkisen oletusportin.

Porttiparametrin g puuttuessa käytetään si tätä arvoa, jota `with-output-to-file` vaihta

`(output-port?` g`)` kertoo, onko g tulostusportti

`(open-output-file` v`)` on alimman abstraktiotas liittymä: se (yrittää avata ja) palauttaa tiedostoon nimeltä v tulostusportin g, jonk sulkeminen jää ohjelmoijan vastuulle.

`(close-output-port` g`)` sulkee edellisen kutsun palauttaman avoimen tulostusportin g.

II.9.8 Syöte

Syötteenkäsittelykirjastorutiinit muistuttavat kalvojen II.9.7 tulostusrutiineja [D96,§7.1;KCR98,§6.6.2]:

`(read)` lukee ja palauttaa (tai ainakin yrittää) syötteestä seuraavan kokonaisen Scheme-tietoalkion d.

Symboleiden ISO- ja pikkukirjaimia ei erotella.

`(eof-object?` d`)` kertoo onko luettu tietoalkio d *syötteenloppualkio* eli loppuiko syöte ennen kuin mitään "oikeaa" d tavattiin.

"Syötettä ei ole *enää* lisää."

`(read-char)` lukee ja palauttaa (tai ainakin yrittää) syötteestä seuraavan merkin (tai syötteenloppualkion).

`(peek-char)` *kurkistaa* ja palauttaa (tai ainakin yrittää) syötteestä seuraavan merkin (tai syötteenloppualkion): kurkistettu merkki *jää* syötteeseen odottamaan seuraavaa varsinaista `read`-lukuoperaatiota.

`(char-ready?)` kertoo onko syötteessä seuraava merkki (tai syötteenloppualkio) *valmiina* luettavaksi tai kurkistettavaksi.

Jos ei, niin luku- tai kurkistusyritys jäisi *odottamaan*.

"Syötettä ei ole *vielä* lisää."

Oletussyöteportinkin (yleensä `stdin`) voi ohjata tiedostoon kalvojen II.9.7 tapaan:

`(with-input-from-file` v `(lambda () ` e`))` on kuin `with-output-to-file`.

`(call-with-input-file` v `(lambda (`g`) ` e`))` on kuin `call-with-output-file`.

`(current-input-port)` on kuin `current-output-po`

`(input-port?` g`)` on kuin `output-port?`.

`(open-input-file` v`)` on kuin `open-output-file`.

`(close-output-port` g`)` on kuin `close-output-por`

II.9.9 Ohjelmien lataaminen

Kalvojen II.9.8 kirjastorutiineilla voidaan lukea *dataa ohjelmiin*. Miten luetaan suoritettavia *ohjelmia muistiin*?

Tai toisin sanoen, niillä voidaan lukea tavaraa joka on "näkymättömän heittomerkin" (kalvo II.8.4) sisällä; mutta ohjelmat pitäisi lukea heittomerkin ulkopuolella jotta ne suoritettaisiin.

Scheme-standardi ehdottaa (muttei vaadi) tähän kirjastofunktiota [D96,§7.3;KCR98,§6.6.4]

(`load` v)

joka avaa Scheme-lähdekielisen tiedoston nimeltä v, ja suorittaa kirjoitusjärjestyksessä sen sisältämät `define`-määritelmät ja lausekkeet.

Eri toteutuksissa se saattaa pystyä muuhunkin, esimerkiksi lukemaan *ennalta käännetyn* Scheme-objektikooditiedoston.

DrScheme-toteutus sisältää myös *projektinhallintatyökalun* jolla voi välttää tarvittavien tiedostojen latailun käsin.

III Esimerkki: sääntöjärjesteln

Toteutetaan Scheme-kielellä epätriviaali symbol ohjelmoinnin esimerkki: *tulkki yksinkertaisille sääntöjärjestelmille*.

(Sellainen on oppikirjassakin [AS96,§4.4] mutta tehtynä hieman toisin.)

Tehdään toisin sanoen ohjelma joka saa – tiete symboliyhdistelmin esitettyinä – syötteinä

- joukon ongelmanratkaisusääntöjä ja
- ongelman

ja kertoo voiko ongelman ratkaista näillä säänn

Valitaan sääntökieli niin, että sillä voidaan esitt ratkaista vaikkapa kalvojen I.1 tyyppisiä päättelytehtäviä, joissa säännöt ovat muotoa

jos fakta$_1$ **ja** ... **ja** fakta$_k$ **niin myös** uusfak

missä kukin *fakta* on yksinkertainen väitelause.

III.1 Termien esitys

Sääntökielemme perusrakenne olkoot *termit* joita on kahta lajia:

Muuttujatermit joita tarvitaan koska haluamme kirjoittaa *yleisiä* sääntöjä kuten

jos x on ihminen **niin** x on kuolevainen *kaikilla* x.

Esitetään muuttujatermit Scheme-symboleilla.

Rakennetermit joita tarvitaan koska haluamme kirjoittaa *ominaisuuksista* joita muuttujilla on kuten

$$\text{ihminen}(x).$$

Esitetään rakennetermit Scheme-listoilla

$$(\,\text{funktori}\ \underbrace{\text{termi}_1 \ldots \text{termi}_k}_{\text{alitermit}}\,)$$

missä ominaisuuden nimi eli *funktori* on symboli.

Erityisesti *vakiot* kuten 'Sokrates' esitetään siten että *paikkaluku (ariteetti, arity)* $k = 0$: `(sokrates)`.

```
; A <term> is either of:
;  * <variable> represented as a symbol
;  * <compound> represented as list
;    (<functor> <term> ... <term>)
;    where <functor> is a symbol.
; (In particular, #f is NOT a <term>.)

; Is this <term> a <variable>?
(define term:variable? symbol?)

; Are these two <variable>s the same one?
(define term:variable=? eq?)

; Is this <term> a <compound>?
; Not a complete test:
; list (term:subterms <this>) is not checked.
(define term:compound?
  (lambda (<term>)
    (and (pair? <term>)
         (symbol? (car <term>)))))

; The <functor> of this <compound>.
(define term:functor car)

; Are these two <functor>s the same one?
(define term:functor=? eq?)

; The sub<term> list of this <compound>.
(define term:subterms cdr)

; Create a <compound>
; from a <functor> and a sub<term> list.
(define term:combine cons)

; Create a previously unknown <variable>.
(define term:new-variable
  (let ((index -1))
    (lambda ()
      (set! index (+ index 1))
      (string->symbol
       (string-append "X"
                      (number->string index))))))
```

III.2 Samastus

Kalvojen III.1 termien keskeinen operaatio on tunnistaa *miten kaksi termiä saadaan samoiksi asettamalla niiden muuttujatermeille arvoiksi sopivat termit*.

Esimerkiksi termi

(ihminen (sokrates))

on termin

(ihminen x)

erikoistapaus asettamalla $x =$ (sokrates).

Silloin saadaan seurata vastaavaa "jos... niin..." -sääntöä ja tuottaa termistä

(kuolevainen x)

haluttu tulostermi

(kuolevainen (sokrates)).

On siis pidettävä kirjaa muuttujatermien arvoista. Tähän sopii kalvojen II.8.7 assosiaatiolista.

(Listaus jatkuu yli kalv

```
; A <binding> is a finite mapping from <variable>s to <term>s
; represented as an association list.

; The empty <binding>.
(define binding:empty '())

; Create a new <binding> which is as given
; but in addition maps <variable> to <term>.
(define binding:add
  (lambda (<variable>
           <term>
           <binding>)
    (cons (cons <variable>
                <term>)
          <binding>)))

; Return the <term> which is the value of this <variable>
; in this <binding>, or #f if there is none.
(define binding:get
  (lambda (<variable>
           <binding>)
    (let ((bound (assoc <variable>
                        <binding>)))
      (if bound
          (cdr bound)
          #f))))

; Expand this <term> in full by substituting each <variable> occur
; in it with the corresponding value in <binding>. We must keep
; <binding>s ACYCLIC so that this result remains finite.
(define binding:expand
  (lambda (<term>
           <binding>)
    (let expand ((this <term>))
      (cond ((term:variable? this)
             (let ((value (binding:get this
                                       <binding>)))
               (if value
                   (expand value)
                   this)))
            ((term:compound? this)
             (term:combine (term:functor this)
                           (map expand
                                (term:subterms this))))))))

; Test whether this <variable> (which does not have a value in <bi
; would occurr in (binding:expand <compound> <binding>). (The expa
```

```
; is NOT carried out for space efficiency.) If it would, then
; binding <variable> to <compound> in <binding> is forbidden, since it
; would violate the aforementioned acyclity condition.
(define binding:occurs?
  (lambda (<variable>
           <compound>
           <binding>)
    (letrec ((check-term
              (lambda (<term>)
                (cond ((term:variable? <term>)
                       (let ((value (binding:get <term>
                                                 <binding>)))
                         (if value
                             (check-term value)
                             (term:variable=? <variable>
                                              <term>))))
                      ((term:compound? <term>)
                       (check-compound <term>)))))
             (check-compound
              (lambda (<compound>)
                (check-term-list (term:subterms <compound>))))
             (check-term-list
              (lambda (term-list)
                (and (not (null? term-list))
                     (or (check-term (car term-list))
                         (check-term-list (cdr term-list)))))))
      (check-compound <compound>))))
```

(Listaus jatkui yli kalvorajan.)

Sokrates-esimerkissämme sovitettiin yhteen muuttujatonta vakiotermiä (`ihminen (sokrates)`) muuttujalliseen sääntötermiin (`ihminen` x).

Yleisessä tapauksessa *molemmat* sovitettavat t voivat sisältää muuttujia: "x on y:n esi-isä jos

1. x on y:n isä tai

2. x on y:n isän esi-isä"

muuntuu säännöiksi

1. **jos** (`isa` x y) **niin** (`esi-isa` x y) ja

2. **jos** (`isa` y z) **ja** (`esi-isa` x z) **niin** (`esi-isa`

joista jälkimmäistä käytettäessä y:n isä z ei löyd uusfaktaa (`esi-isa` x y) samastamalla, joten jo samastus aloitetaan siitä niin seuraaviin jää z.

Lisäksi halutaan *yleisin* mahdollinen muuttujien sidonta: (`ihminen` x) ja (`ihminen` y) tulevat vähimmillään samoiksi sidonnalla $x = y$ kun taas $x = y =$ (`sokrates`) olisi turhan yksityiskohtainen

Tämä *samastusoperaatio* (unification) on keske osa monia (varsinkin logiikkaan liittyviä) symbolinkäsittelytehtäviä. Sen tulos on *yleisin samastin* (most general unifier, mgu).

(Listaus jatkuu yli kalvorajojen.)

```
; Unify takes two <term>s and a <binding>,
; and tries to extend minimally it into a <binding'> such that
; (equal? (binding:expand <term1> <binding'>)
;         (binding:expand <term2> <binding'>)) yields #t.
(define unify
  ; The unification algorithm is developed as a set of mutually recursive
  ; subroutines (uxy <term1> <term2> <binding>) where
  ; x (y, respectively) tells what is known about current <term1>
  ; (<term2>, respecively):
  ;  v: a <variable> without value in <binding>
  ;  c: a <compound>
  ;  t: either kind of term.
  (letrec (; The main routine is for two terms of any kind:
           ; Determine what kind of term the first one is.
           (utt
            (lambda (<term1>
                     <term2>
                     <binding>)
              (cond ((term:variable? <term1>)
                     (let ((value (binding:get <term1>
                                               <binding>)))
                       (if value
                           (utt value
                                <term2>
                                <binding>)
                           (uvt <term1>
                                <term2>
                                <binding>))))
                    ((term:compound? <term1>)
                     (uct <term1>
                          <term2>
                          <binding>)))))
           ; Term 1 is a variable, determine what term 2 is:
           (uvt
            (lambda (<variable>
                     <term>
                     <binding>)
              (cond ((term:variable? <term>)
                     (let ((value (binding:get <term>
                                               <binding>)))
                       (if value
                           (uvt <variable>
                                value
                                <binding>)
                           (uvv <variable>
                                <term>
                                <binding>))))
                    ((term:compound? <term>)
                     (uvc <variable>
                          <term>
                          <binding>)))))
           ; Term 1 a compound, determine what term 2 is:
           (uct
            (lambda (<compound>
                     <term>
                     <binding>)
              (cond ((term:variable? <term>)
                     (let ((value (binding:get <term>
                                               <binding>)))
                       (if value
                           (uct <compound>
                                value
                                <binding>)
                           (uvc <term>
                                <compound>
                                <binding>))))
                    ((term:compound? <term>)
                     (ucc <compound>
                          <term>
                          <binding>)))))
           ; Both terms are variables, bind them together if neces
           (uvv
            (lambda (<variable1>
                     <variable2>
                     <binding>)
              (if (term:variable=? <variable1>
                                   <variable2>)
                  <binding>
                  (binding:add <variable1>
                               <variable2>
                               <binding>))))
           ; Try to bind a variable into a compound:
           (uvc
            (lambda (<variable>
                     <compound>
                     <binding>)
              (if (binding:occurs? <variable>
                                   <compound>
                                   <binding>)
                  #f
                  (binding:add <variable>
                               <compound>
                               <binding>))))
           ; Both terms are compounds, so try to unify their subte
           ; recursively:
           (ucc
```

```
        (lambda (<compound1>
                 <compound2>
                 <binding>)
          (and (term:functor=? (term:functor <compound1>)
                               (term:functor <compound2>))
               (uccs (term:subterms <compound1>)
                     (term:subterms <compound2>)
                     <binding>))))
      (uccs
       (lambda (subterms1
                subterms2
                binding)
         (cond ((null? subterms1)
                (if (null? subterms2)
                    binding
                    #f))
               ((null? subterms2)
                #f)
               (else
                (let ((car-binding (unify (car subterms1)
                                          (car subterms2)
                                          binding)))
                  (if car-binding
                      (uccs (cdr subterms1)
                            (cdr subterms2)
                            car-binding)
                      #f)))))))
utt))
```

(Listaus jatkui yli kalvorajojen.)

Kirjallisuutta

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

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

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

B01 I.Bratko: *Prolog Programming for Artificia Intelligence. Third Edition.* Addison Wesle 2001.

CM87 W.F.Clocksin&C.S.Mellish: *Programming Prolog. Third, Revised and Extended Edit* 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: *Prolog: 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.* Addison Wesley, 1992.

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

KCR98 R.Kelsey, W.Clinger&J.Rees (toim.): Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11(1), 1998 **tai** *ACM SIGPLAN Notices* 33(9), 1998.

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

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

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

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

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

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