4. Tekoälyohjelmoinnista [CR91][AS96, luvuista 1–3]

Tehdään lyhyt katsaus luvussa 1.5 mainitun tekoälyohjelmoinnin valtakielen *Lisp* kurssilla käytettyyn *pienempään ja ortogonaalisempaan* murteeseen *Scheme*, joka siis soveltuu paremmin (itse)opiskeluun.

Scheme poikkeaa tutummista lausekielistä kuten C(++), Java, ... esimerkiksi seuraavissa suhteissa:

1. Siinä on jo kielen tasolla erillinen tietotyyppi *symboleille* tavallisten numeeristen ja merkki(jono)tyyppien lisäksi.
2. Siinä on jo kielen tasolla työkalut *listojen* mahdollisimman joustavaan käsittelyyn.
3. Scheme-ohjelman suoritus voidaan mallintaa pelkkänä *laskutoimituksena* vetoamatta tietokonesanastoon kuten "muisti" ym.

Esimerkiksi luvun 1.1.3 Aristotelinen syllogismi (1)–(3) voitaisiin kirjoittaa "Scheme-tyyliin"

`(ihminen sokrates)` (12)

`(jos (ihminen (? . x)) (kuolevainen (? . x)))` (13)

`(kuolevainen sokrates)` (14)

ja ohjelmoida kone tekemään seuraavaa: "Jos muistissa on ilmaukset A ja `(jos` B C`)`, ja jos ilmaukset A ja B saadaan samoiksi valitsemalla sopivat arvot joillekin niiden aukkopaikoille `(? .` z`)`, niin tuo muistiin myös ilmauksen C kopio, jossa nämä aukkopaikat on täytetty samoilla valinnoilla."

Näin kone olisi ohjelmoitu tekemään yksinkertaisia päätelmiä — kuten esimerkkimme valitsemalla aukon `(? . x)` täytteeksi `sokrates`.

Huomaa, että `jos`-rakenne on dataa, ei ohjelmalause!

Scheme poikkeaa muissakin suhteissa tavallisemmista lausekielistä:

- Ohjelmat eivät koostu *peräkkäisistä käskyjonoista* vaan *sisäkkäisistä funktionkutsuista*.

  Suoritus ei siis etene *ohjelmariviltä seuraavalle* vaan *sisältä ulos*: sisemmät funktionkutsut palauttavat tuloksena ulommalle, joka jatkaa ne saatuaan.

- Nämä funktiot ovat samassa asemassa kuin muukin tieto: niistä voidaan luoda suoritusaikana uusia kopioita, joita voidaan lähettää parametreina, palauttaa tuloksina, tallettaa muuttujien arvoiksi, ...

  Tämä helpottaa *korkeamman kertaluvun (higher-order)* ohjelmointia, jossa kirjoitetaan sellaisia funktioita, jotka saavat parameterinään toisia funktioita, joihin ne lisäävät uusia toimintoja.

- Kielessä on *implisiittinen tyypitys*: muuttujien tyyppejä ei esitellä, vaan jokainen muuttuja voi saada arvokseen minkä tahansa tyyppisen Scheme-tietoalkion.

  Arvojen tyyppien sopivuus tarkistetaan vasta niiden käyttöhetkellä: (+ x y) johtaa *suoritusaikaiseen virheeseen* jos tällä hetkellä muuttujien x ja y arvot eivät ole numeerisia.

  (*Täysin* tyypittömässä kielessä etsittäisiin silloinkin jokin "järkevä" tulkinta tälle operaatiolle.)

- Korkeamman kertaluvun ohjelmoinnin helppouden ja implisiittisen tyypityksen vuoksi Scheme soveltuu *kokeilevaan (exploratory)* ohjelmointiin, jossa kokeillaan nopeasti prototyyppejä erilaisista ideoista muokkaamalla olemassaolevaa ohjelmakoodia.

  Tämä lienee merkittävä syy Lisp-tyyppisten kielten suosioon tekoälyohjelmoinnissa: uutta ideaa pääsee kokeilemaan vähemmällä vaivalla, jolloin sen osoittautuminen kehnoksi ei merkitse niin suurta hukkaan heitettyä työmäärää.

  Kokeilevassa ohjelmoinnissa on luontevinta käyttää *tulkattuja (interpreted)* kieliä niiden interaktiivisuuden vuoksi. Näin teemme tälläkin kurssilla käyttäen Scheme-tulkkia `gsi`. Valmiit ohjelmat voidaan sitten kääntää nopeammaksi konekoodiksi (esimerkiksi kääntäjällä `gsc`).

- Scheme-kielessä jää muistinhallinta suoritusmekanismin huoleksi: ohjelmoija voi (ja hänen täytyy) elää siinä (harha)luulossa, että koneessa on muistia aina "tarpeeksi".

4.1. Symbolinen data

Scheme-kielessä on oma tietotyyppi nimeltään *symboli*, jolla on seuraavat ominaisuudet:

- Symboleita on (periaatteessa) *ääretön* määrä.

  Käytännössä tämä tarkoittaa, että Scheme-toteutus pitää kirjaa kaikista kohtaamistaan symboleista, esiintyivätpä ne sitten ohjelmassa tai syötteessä.

  Syötesymboleita kuten `sokrates`, `ihminen` ja `kuolevainen` ei siis tarvitse erikseen esitellä etukäteen.
- Jokainen symboli on *uusi jakamaton tietoalkio*, joka eroaa kaikista muista tietoalkioista.

  Symboli `sokrates` *ei* ole sama asia kuin *merkkijono* `"sokrates"`: esimerkiksi jälkimmäinen alkaa merkillä `#\s`, kun taas edellinen ei koostu lainkaan merkeistä.

  Symboli `sokrates` ei myöskään ole sama kuin mikään toinen symboli, vaikkapa `ihminen`.
- Jokaisella symbolilla on *arvo*.

  Näin symbolit ajavat saman asian kuin muiden ohjelmointikielten muuttujat, muun muassa.

  Symboli voidaan esitellä ja antaa sille (alku)arvo lausekkeella `(define` *`symboli arvo`*`)`.

  Kahden eri symbolin arvot voivat olla samat, esimerkiksi `(define sokrates 0)` ja `(define ihminen 0)`.

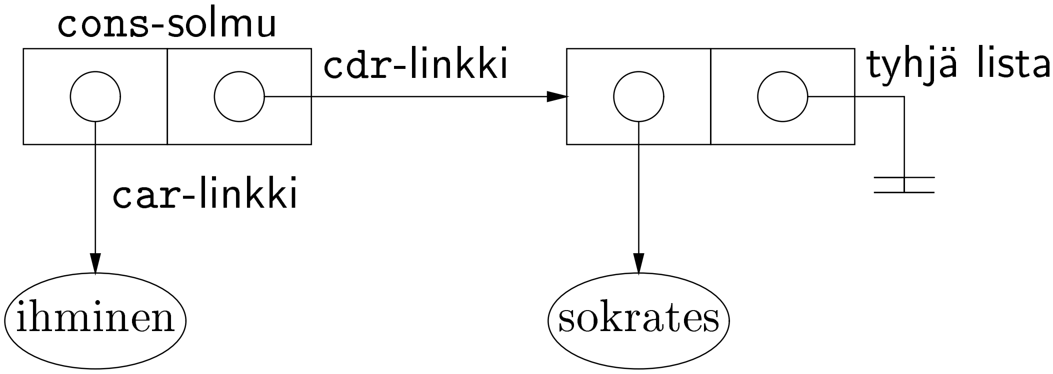
Symboleiden voidaan ajatella olevan John Searlen kiinalaisen huoneen [Sea90] kirjoitusmerkkejä, joilla ei ole ”merkitystä” niitä mekaanisesti manipuloivalle tietokoneelle. Kunkin symbolin arvo saadaan puolestaan katsomalla huoneessa olevasta ”sanakirjasta”, ja on

- joko toinen symboli tai symbolien monimutkaisempi yhdistelmä, joka ojennetaan huoneesta ulos vastauksena,
- tai epäsuora ohje, miten tuo vastaus pitää muodostaa.

4.2. Listat ja niiden käsittely

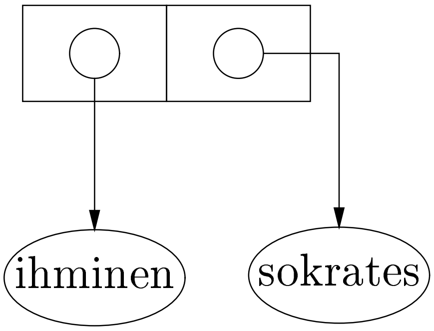
Yksittäisiä symboleita voidaan yhdistellä mutkikkaammikksi rakenteiksi *pariuttamalla* niitä toistensa kanssa *listoiksi*.

Listan (ihminen sokrates) esitys solmuina ja linkkeinä on seuraava:



Kuva 29: Scheme-listan esitys.

Listoja ovat siis *tyhjä lista* `()` ja sellaiset `cons`-solmut, joiden `cdr`-linkki osoittaa listaan. Muuten kysessä on *pisteytetty pari (dotted pair)* (ihminen . sokrates):



Kuva 30: Scheme-parin esitys.

Listoja ja pareja käsitellään vastaavilla perusoperaatioilla:

- (`car` c) hakee *cons*-solmun c `car`-linkin.
- (`cdr` c) hakee *cons*-solmun c `cdr`-linkin.
- (`cons` a d) luo uuden `cons`-solmun, jonka `car`-linkki osoittaa solmuun (symboliin, vakioon, . . .) a, ja `cdr`-linkki solmuun d.
- (`null?` l) kertoo, onko lista l tyhjä vai ei.

Silloin kuvan 29 lista saataisiin Scheme-lausekkeella

```
(cons 'ihminen (cons 'sokrates '()))
```

ja kuvan 30 pari taas lausekkeella

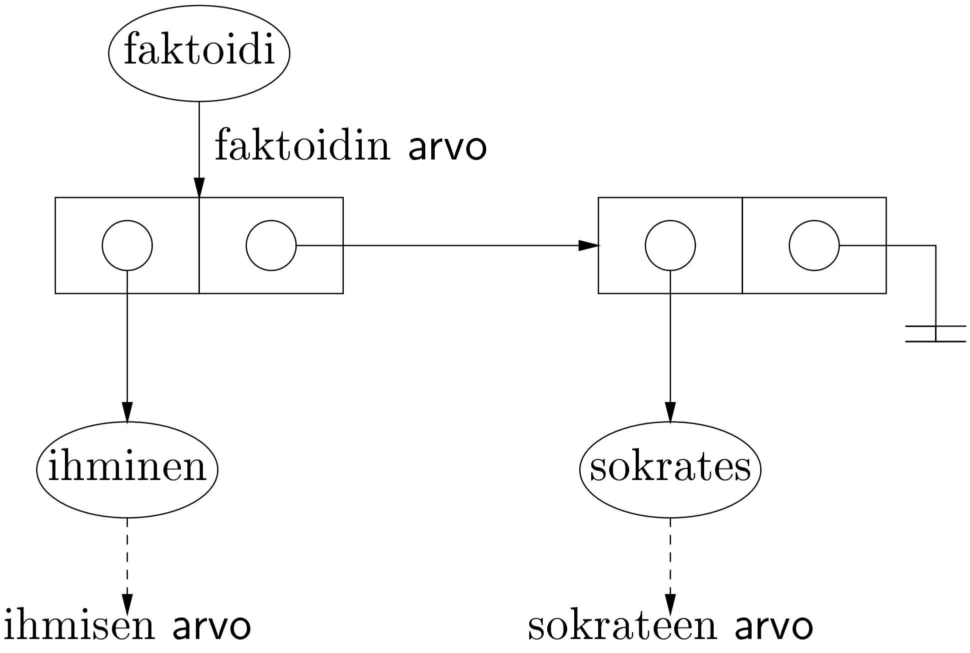
```
(cons 'ihminen 'sokrates)
```

joiden suoritusta selittää luku 4.3.

Jos siis suoritetaan lauseke

(define *faktoidi* (cons 'ihminen (cons 'sokrates '())))

niin tulos on seuraava:



Kuva 31: Lista symbolin arvona.

Myös Scheme-funktioiden kutsut kirjoitetaan listanotaatiolla:

$$(\text{nimi arg}_1 \ldots \text{arg}_n)\,.$$

Kielen muutkin rakenteet kirjoitetaan listoina, joisen ensimmäinen jäsen kertoo mistä rakenteesta nyt on kyse.

4.3. Ohjelman suoritus sieventämisenä

Scheme-kielinen funktionmäärittely kuten

```
(define (len l)
  (if (null? l)
      0
      (+ 1
         (len (cdr l)))))
```

voidaan tulkita *sievennyssääntönä* "kukin ilmaus `(len` X`)` voidaan sieventää ilmaukseksi `(if (null?` X`) 0 (+ 1 (len (cdr` X`))))`" jota sitten sievennetään edelleen.

Esimerkiksi `(len '(ihminen sokrates))` sievenee seuraavasti:

1. Sievennykset suoritetaan *sisältä ulos*, eli ensin kaikki argumentit, nyt ainoastaan `'(ihminen sokrates)`. (Scheme-kielessä on myös `delay`, jolla sievennysjärjestystä voi itse ohjailla.)
   Lainausmerkki `'` sanoo, että sen jäljessä oleva ilmaus on dataa, eli sievenee omaksi itsekseen: ilmaus `(ihminen sokrates)` sievenee siis *arvoonsa* listaksi $(\mathrm{ihminen\ sokrates})$ (eikä siis funktion `ihminen` kutsuksi argumenttinaan symbolin `sokrates` arvo).
   Kun argumentin arvo

   $$X = (\mathrm{ihminen\ sokrates})$$

   on näin selvitetty, käytetään `len`-sievennyssääntöä, ja saadaan

   `(if (null?` $(\mathrm{ihminen\ sokrates})$`)`
   `    0`
   `    (+ 1`
   `       (len (cdr` $(\mathrm{ihminen\ sokrates})$`))))`.

2. Ilmaukset `(if` E T F`)` sievennetään hieman yleisperiaatteesta poiketen: ensin sievennetäänkin vain *ehto* E, ja sen arvon perusteella vain toinen *haaroista* T tai F.

Aloitetaan siis sieventämällä ehto

$$E = \texttt{(null? } (\text{ihminen sokrates})\texttt{)}.$$

Predikaatti `null?` kuuluu Scheme-standardiin, ja kertoo onko sen argumenttina saama lista tyhjä (luku 4.2). Nyt $(\text{ihminen sokrates})$ ei sitä ole, joten E sievenee epätodeksi `#f`.

Jatketaan siis sieventämällä haara

$$F = \texttt{(+ 1 (len (cdr } (\text{ihminen sokrates})\texttt{)))}.$$

Sen ensimmäinen argumentti `1` sievenee vakioksi 1. Scheme-standardin mukaan `+` tarkoittaa yhteenlaskuoperaatiota. Sievennystulos saadaan siis laskemalla

$$1 + \texttt{(len (cdr } (\text{ihminen sokrates})\texttt{))},$$

kunhan tuo jälkimmäinen operandi vielä sievennetään.

3. Kyseessä on jälleen `len`-funktion kutsu, joten aloitetaan sieventämällä sen argumentti

   $$X = \texttt{(cdr } (\text{ihminen sokrates})\texttt{)}.$$

   Luvun 4.2 nojalla sen arvo on (sokrates).

   Sovelletaan taas `len`-sievennyssääntöä, ja saadaan kokonaistilanteeksi

   $$1 + \texttt{(if (null? } (\text{sokrates})\texttt{) 0 (+ 1 (len (cdr } (\text{sokrates})\texttt{))))}.$$

4. Sievennys jatkuu kuten askeleessa 2, ja johtaa lopulta tulokseen

   $$1 + (1 + \texttt{(if (null? } ()\texttt{) 0 (+ 1 (len (cdr } ()\texttt{)))))}).$$

   Nyt kuitenkin `(null?` $()$`)` sieveneekin toteen arvoon `#t`, joten `if`-sievennys jatkuukin haarasta $T =$ `0`, joka sievenee vakioksi 0.

   Sievennys saadaan siis pisteeseen, jossa jäljellä ovat enää laskutoimitukset, joiden operandit on nyt sievennetty:

   $$1 + (1 + 0) = 2$$

Sievennysmallilla on kaksi etua:

- Ohjelman `len` suorittaminen voitiin selittää vetoamatta lainkaan mihinkään (olemassaolevaan tai hypoteettiseen) tietokoneeseen tai laskulaitteeseen.

  Sen sijaan vedottiin *symbolien ja niiden yhdistelmien mekaaniseen manipulointiin*.

- Ohjelmien toiminnan matemaattinen tarkastelu helpottuu. Erityisen helppoa on *todistaa rekursiivisen algoritmin toiminta induktiolla*:

  Väite: Funktio `len` laskee syötteenä saamansa listan pituuden.

  Perustapaus: Sen rekursioton haara otetaan, kun saatu lista on tyhjä. Haara antaa sille arvon 0, kuten pitääkin.

  Induktio: Sen rekursiivinen haara otetaan, kun saatu lista L ei ole tyhjä, eli on luvun 4.2 nojalla muotoa $L = (A\ .\ L')$, missä L' on yhtä jäsentä lyhyempi lista.
  Haara antaa arvon $1+m$, missä m on funktion `len` arvo listalla L'. Induktion nojalla m on silloin listan L' pituus, jolloin haaran arvo on listan L pituus, kuten pitääkin.

Scheme-kielen tietokone*toteutus* ei sieventele; riittää, että käyttäjälle näkyvä toiminta pitää yhtä sievennysmallin kanssa.

Funktionmäärittely

(`define` (nimi arg$_1$ ... arg$_n$) lauseke$_1$ ... lauseke$_m$)

onkin itse asiassa lyhennysmerkintä symbolinmäärittelylle

(`define` nimi (`lambda` (arg$_1$ ... arg$_n$) lauseke$_1$ ... lauseke$_m$))

missä `lambda` on *funktioabstraktio-operaatio*, joka tekee sisältämistään lausekkeista lauseke$_1$ ... lauseke$_m$ funktion, jonka muodolliset argumentit ovat arg$_1$ ... arg$_n$. Vertaa matematiikkaan: $n+1$ on lauseke, josta saadaan funktio $n \mapsto n+1$ abstrahoimalla (lat. abs-trahere "vetää pois") n symbolista argumentiksi.

4.4. Muistin käsittelystä!

Luvun 4.3 sievennysmalli on teoreettisesti kaunis, mutta joskus käytännön ohjelmoinnissa täytyy käsitellä muistia suoraan. Syynä on *tehokkuus* tai *tarve tallettaa jotakin pysyvästi funktiokutsujen välillä tai globaalisti*.

Scheme-kielessä käsiteltävää muistia ovat kuvien 29–31 *nuolet*:

- Symbolin arvoa — eli siitä lähtevää arvoon osoittavaa nuolta — voidaan muuttaa lausekkeella (`set!` s e), jonka jälkeen symbolin s arvona onkin lausekkeen e tulos.

  Määrittely (`define` s e) onkin itse asiassa `set!` (lukuunottamatta eräitä erikoistilanteita joita emme tapaa kurssillamme).
- `cons`-parin kenttien nuolia voidaan ohjailla uudelleen lausekkeilla (`set-car!` p e) ja (`set-cdr!` p e), joiden jälkeen parin p vastaava kenttä osoittaakin lausekkeen e tulokseen.

On vakiintunut tapa lopettaa muistia muuttelevien proseduurien nimet huutomerkkiin '`!`' osoittamaan, että ne rikkovat sievennyssemantiikan, jolloin niiden toiminnan hahmottamisessa täytyy palata ajattelemaan luvun 4.2 kuvia.

Toinen vakiintunut tapa on lopettaa totuusarvoja palauttavat funktiot kysymysmerkkiin '`?`'.

Koska todellisissa tietokoneissa on vain äärellinen määrä muistia, Scheme-järjestelmä tekee ohjelmien suorituksen aikana *roskankeruuta (garbage collection)*, eli tutkii mitkä muistissa olevat listarakenteet voidaan nyt *hävittää turhina* ja niiden varaama muisti palauttaa kiertoon.

Intuitiivisesti rakenne on turha, jos myöhempi suoritus ei enää siihen voi viitata — koska siihen ei enää tule linkkiketjuja.

4.5. Etsintäalgoritmeja Scheme-kielellä

Ohjelmoidaan ensin yleinen etsintäalgoritmi 2 Scheme-kielellä:

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

```
(define (general-search problem-initial-node
                        problem-successor-function
                        problem-goal-test?
                        memory-store!
                        memory-fetch!
                        memory-empty?)
  (define (search)
    (if (memory-empty?)
        #f
        (let ((node (memory-fetch!)))
          (if (problem-goal-test? node)
              node
              (begin
                (for-each memory-store!
                          (problem-successor-function node))
                (search))))))
  (memory-store! problem-initial-node)
  (search))
```

Parametrit ovat:

- `problem-initial-node` on ratkaistavan etsintäongelman $problem$ alkutila.
- `problem-successor-function` on funktio, joka kuvaa argumenttinaan saamansa ongelman tilan listaksi mahdollisia seuraavia tiloja. Se on siis operaattorit esittävä seuraajafunktio.

- `problem-goal-test?` on predikaatti, joka kertoo onko sille argumenttina annettu ongelman tila lopputila.
- `memory-store!` on proseduuri, joka tallettaa argumenttina annetun ongelman tilan etsinnässä käytettävään muistiin $nodes$. (Eri muistimallithan johtivat eri etsintäalgoritmeihin.)
- `memory-fetch!` on argumentiton proseduuri, joka ottaa ja palauttaa muistista seuraavan käsiteltävän ongelman tilan.
- `memory-empty?` on argumentiton predikaatti, joka kertoo, onko muisti tyhjä.

Luvun 3.1.3 syvyyssuuntaisessa etsinnässä muistimallina oli pino:

```
(define (depth-first-search problem-initial-node
                            problem-successor-function
                            problem-goal-test?)
  (let ((stack '()))
    (general-search problem-initial-node
                    problem-successor-function
                    problem-goal-test?
                    (lambda (pushing)
                      (set! stack
                            (cons pushing
                                  stack)))
                    (lambda ()
                      (let ((popping (car stack)))
                        (set! stack
                              (cdr stack))
                        popping))
                    (lambda ()
                      (null? stack)))))
```

Jokaisella kutsulla suoritetaan `let` uudelleen, eli varataan uusi pino `stack`. Samoin suoritetaan kaikki `lambda`t uudelleen, ja saadaan näin tätä etsintää varten uudet proseduurikopiot, jotka käsittelevät tätä juuri luotua pinoa `stack`. Nimillä on siis tuttu *leksikaalinen* sidonta, eli ne viittaavat aina lähimpään määrittelyynsä. Muistinhallinta huolehtii siitä, milloin kukin `stack` voidaan hävittää tarpeettomana.

Rakenne `let` — kuten useimmat Scheme-rakenteet — on itse asiassa lyhenne eräälle yleisesti esiintyvälle `lambda`-lausekkeelle.

```
(define (knights-tour board-width
                      search-strategy)
  (let ((start-square '(1 . 1)))
    (search-strategy
     (list start-square)
     (lambda (tour)
       (if (member (car tour)
                   (cdr tour))
           '()
           (do ((jump '((-2 . -1) (-2 .  1) (-1 . -2) (-1 .  2)
                        ( 1 . -2) ( 1 .  2) ( 2 . -1) ( 2 .  1))
                      (cdr jump))
                (succ '()
                      (let ((square
                             (cons (+ (caar jump)
                                      (caar tour))
                                   (+ (cdar jump)
                                      (cdar tour)))))
                        (if (and (<= 1
                                     (car square))
                                 (<= (car square)
                                     board-width)
                                 (<= 1
                                     (cdr square))
                                 (<= (cdr square)
                                     board-width))
                            (cons (cons square
                                        tour)
                                  succ)
                            succ))))
               ((null? jump)
                succ))))
     (lambda (tour)
       (and (equal? (car tour)
                    start-square)
            (= (length (cdr tour))
               (* board-width
                  board-width)))))))
```

Lauseke (`knights-tour 8 depth-first-search`) alkaisi silloin selvittää syvyyssuuntaisella etsinnällä, voiko shakkihevonen kiertää koko laudan ympäri käymättä missään ruudussa kahdesti.

Viitteet

[AK83] J.F. Allen and J.A. Koomen. Planning using a temporal world model. In *International Joint Conference on Artificial Intelligence*, pages 741–747, 1983.

[AS96] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Algorithms*. The MIT Press, 2nd edition, 1996.

[Cho97] N. Chomsky. B. F. Skinnerin teos *verbal behaviour*. In P. Raatikainen, editor, *Ajattelu, kieli, merkitys*, pages 279–310. Gaudeamus, 1997. Suom. R. Vilkko.

[CL73] C-L. Chang and R.C-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[CR91] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme, 1991.

[CSC90] P.M. Churchland and P. Smith Churchland. Could a machine think? *Scientific American*, pages 26–31, 1990.

[Myr74a] L. Myrberg. *Differentiaali- ja integraalilaskenta*, volume 1. Kirjayhtymä, 1974.

[Myr74b] L. Myrberg. *Differentiaali- ja integraalilaskenta*, volume 2. Kirjayhtymä, 1974.

[Pen89] R. Penrose. *The Emperor's New Mind*. Oxford University Press, 1989.

[Pin94] S. Pinker. *The Language Instinct*. William Morrow and Company, Inc., 1994.

[Pin97] S. Pinker. *How the Mind Works*. W. W. Norton & Company, Inc., 1997.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, 1995.

[Saa85] E. Saarinen. *Länsimaisen filosofian historia huipulta huipulle*. Werner Söderström oy, 1985.

[Sea90] J. Searle. Is the brain's mind a computer program? *Scientific American*, pages 20–25, 1990.

[Tur50] A.M. Turing. Computing machinery and intelligence. *Mind*, LIX(2236):433–460, 1950.