I Symboliset operaatiot

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:
>
> 1. *Combining* several simple ideas into one compound one, and thus all complex ideas are made.
>
> 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of *relations*.
>
> 3. The third is separating them from all other ideas that accompany them in their real existence: this is called *abstraction*, and thus all its general ideas are made.
>
> *John Locke [AS96,§1]; korostukset luennoijan*

Tutustumme symbolein suoritettaviin laskutoimituksiin kahdella esimerkillä, joiden kuluessa nähdään yllä korostettuja piirteitä käytännössä.

I.1 "Looginen" päättely

> **Logic**, *n.*, The art of thinking and reasoning in strict accordan with the limitations and incapacities of the human misunderstanding. *Ambrose Bie*

Otetaan esimerkiksi tuttu Aristotelinen syllogisr

1. Sokrates on ihminen.

2. Kaikki ihmiset ovat kuolevaisia.

3. Siispä myös Sokrates on kuolevainen.

Tätä järkeillessämme käytämme *symboleita*: "Sokrates", "(olla) ihminen", "(olla) kuolevaine

Emme käytä niiden *vastineita* todellisuudessa: perunanenäinen partainen kreikkalainen, kaikkie ihmisten joukko, kaikkien kuolevaisten joukko.

Pidämme jopa edellistä päättelyä konkreettisena tapauksena yleisestä *abstraktista* periaatteesta:

1. Alkio X kuuluu joukkoon Y

2. Jokainen joukon Y alkio kuuluu joukkoon Z

3. Siispä myös alkio X kuuluu joukkoon Z.

Tässä X, Y ja Z edustavat mielivaltaisia symboleita (tai niiden *monimutkaisempia* yhdistelmiä kuten "Ksantippan aviomies"), ja periaate kertoo miten *relaatio* "kuuluu joukkoon" käyttäytyy.

- On siis symboleita jotka nimeävät (yhdessä tai erikseen) käsitteitä.

- Lisäksi on symbolein suoritettavia operaatioita jotka (toivottavasti) noudattavat / toteuttavat näille käsitteille mielekkäitä toimituksia.

- Syntyy tarve / mahdollisuus **ohjelmoida** näitä operaatioita / näillä operaatioilla.

I.2 Etuja ja haittoja

\+ Voidaan ohjelmoida siten, että sovellusaluee käsitteet näkyvät ohjelman symboleissa ja n operoinnissa.

Hyödyllistä myös ei-symbolisissa tehtävissä!

\+ Voidaan käyttää symboleiden käsittelyyn perustuvia ohjelmoinnin ja laskennan malleja laitteisiin perustuvien mallien sijasta.

Teoriassa algoritmisen ongelman ("laske annettujen kahden luvun summa") ratkaisu mekaanisesti laskettavissa oleva kuvaus sen tapauksilta ("annettu kaksi lukua") oikealle vastaukselle ("niiden summa").

Käytännössä se on usein (i) kuvaus tapauks laitteen muistissa olevaksi rakenteeksi, (ii) t rakenteen kanssa näpräilyä ja (iii) kuvaus lopulliselta rakenteelta vastaukselle – ilman yhteyttä itse ongelmaan!

+ Symbolisten menetelmien oikeellisuuden tarkasteleminen on yleensä helpompaa kuin laitepohjaisten.

  Voidaan nimittäin käyttää *logiikassa* vuosi(satoj)en saatossa kehittyneitä menetelmiä.

− Symboliset ohjelmointikielet ja -menetelmät eivät ole laajalti käytössä teollisuudessa.

  Tilanne saattaa muuttua kun ohjelmistojen oikeellisuuskysymyksiin aletaan kiinnittää nykyistä enemmän kaupallista huomiota.

− Symbolinen ohjelmointi ei ole yhtä tehokasta kuin perinteinen niissä tehtävissä joissa ei tarvittaisi symboliikkaa.

  Ero kutistuu toteutustekniikkojen edistyessä.

I.3 Derivointi

Derivoinnilla on matematiikassa kaksi merkityst

Määritelmä erotusosamäärän raja-arvona

$$\mathbf{D}f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Sääntöinä muuntaa alkuperäisen funktion f lau
– moniosainen symbolinen nimi – derivaatan
lausekkeeksi:

$$\begin{aligned}
\mathbf{D}c &= 0 \\
\mathbf{D}(x^n) &= n \cdot x^{n-1} \\
\mathbf{D}(f(x) + g(x)) &= (\mathbf{D}f(x)) + (\mathbf{D}g(x)) \\
\mathbf{D}(f(x) \cdot g(x)) &= f(x) \cdot (\mathbf{D}g(x)) + g(x) \cdot (\mathbf{D} \\
\mathbf{D}\frac{f(x)}{g(x)} &= \frac{g(x) \cdot (\mathbf{D}f(x)) - f(x) \cdot (\mathbf{D}g}{g(x)^2} \\
&\vdots
\end{aligned}$$

Säännöt todistetaan oikeiksi osoittamalla et
säilyttävät määritelmän.

Säännöt muodostavat "symbolipelin" jossa derivointisymbolia $\mathbf{D}$ siirrellään muista symboleista koostuvassa lausekkeessa kunnes siitä päästään kokonaan eroon.

Kunkin säännön oikealla puolella $\mathbf{D}$ kohdistuu aidosti pienempiin lausekkeisiin: suuntaamalla säännöt vasemmalta oikealle ($\mathbf{D}c \Rightarrow 0$ jne.) saadaan siis pysähtyvä (ja onnistuva) peli – symbolinen algoritmi!

(Toisin kuin yllä, integraalilausekkeiden peli ei välttämättä johda yksikäsitteiseen tulokseen tai edes pysähdy.)

Ei-symbolisessa ohjelmoinnissa jatketaan esimerkiksi seuraavasti:

1. Laaditaan lausekkeille jäsentäjä joka tuottaa lausekkeesta puuesityksen.

   (Lauseke muistiin.)

2. Muunnetaan derivointisäännöt tämän puun käsittelyoperaatioiksi.

   (Näpelöidään muistia.)

3. Lopuksi tulospuu kirjoitetaan vastauksena.

   (Ulostetaan näpelöinnin lopputulos.)

Tässä ratkaisussa on kuitenkin edelleen puute: uusia derivointisääntöjä lisätään?

- Yksi tapa on kirjoittaa vanhan derivointiruti perään aina lisää uutta koodia.

  Tuloksena on ylläpidon painajainen.

- Toinen tapa on suunnitella ja toteuttaa uus pikku kieli jolla säännöt kirjoitetaan ja jota suorittamalla derivointi tapahtuu.

  Tuloksena on pieni muun ohjelman sisään upotettu symbolinen ohjelmointikieli: Mathematica, Maple, AutoCAD (AutoLISP! GNU/X Emacs,...

Symbolisella ohjelmointikielellä voi joskus kiertä tämän *tietämyksenesitysongelman* ja kirjoittaa säännöt (lähes) suoraan ohjelmana.

Ainakin sillä on helpompi toteuttaa tuon pikkuk tulkki, onhan tulkkauskin symbolinen toimitus.

Kääntäjienkin kirjoittaminen helpottuu, onhan kääntäminenkin symbolinen toimitus.

Vastaava Prolog-kielinen ohjelma on yksinkertaisimmillaan:

```
dx(C, 0):-
        number(C).

dx(x ** N, N * x ** N1):-
        N1 is N-1.

dx(F * G, F * G1 + G * F1):-
        dx(F,F1),
        dx(G,G1).

dx(F + G, F1 + G1):-
        dx(F,F1),
        dx(G,G1).

dx(F / G, (G * F1 - F * G1) / (G ** 2)):-
        dx(F,F1),
        dx(G,G1).
```

Esimerkiksi "+-sääntö" luetaan suoraan "lausekkeen `F + G` derivaatta on `F1 + G1` missä `F1` on alilausekkeen `F` ja `G1` alilausekkeen `G` derivaatta".

Siis ohjelma `dx` on määritelmä kahden lausekkeen relaatiolle.

II Ohjelmointikieli Scheme

Aloitetaan symbolisen ohjelmoinnin käytäntöön tutustuminen (Prolog-kielen sijasta) Scheme-kielestä, joka on yksi Lisp (List process -murre.

II.1 Historiaa

60-luku: Alkuperäisen Lisp-kielen kehitti John McCarthy vuonna 1958 vasta perustetussa MIT AI -laboratoriossa.

Vain alkuperäinen Fortran on (vähän) vanhe

Alunperin Lisp oli notaatio symboleita käsittelevien ohjelmien spesifiointiin; tietokonetoteutus (LISP 1.5) tuli hieman myöhemmin.

Inspiraationa oli matemaattisen logiikan ns. λ-kalkyyli vuodelta 1936.

70-luku: Tekoälyinnostuksen myötä Lisp levisi ja jakautui vaikutusvaltaisiin murteisiin: MacLISP (MIT; Macsyma), InterLisp (Xerox PARC; kokonainen varhainen visuaalinen ympäristö), Zetalisp (MIT; Lisp Machine), Franz LISP (Berkeley; VAX-tietokoneperhe), Portable Standard LISP (Utah; REDUCE), **Scheme** (MIT),. . .

80-luku: Baabelin hajaannusta korjaamaan tehtiin *Common Lisp* -standardi.

Siihen otettiin joitakin Scheme-murteen Lisp-kulttuuriin esittelemiä innovaatioita, erityisesti *Algol*-kieliperheen mukainen lohkorakenne.

90-luku: Common Lisp on standardina käytössä teollisuudessa, ei kuitenkaan laajalti.

Scheme on jatkanut kehittymistään korkeakoulujen opetus- ja tutkimuskäytössä. (Suomessakin TKK:n tietoteekkareilla.)

II.2 Common Lisp vastaan Scheme

Common Lisp	**Scheme**
ANSI-standardoitu.	ANSI-standardoitu mutta kehitys jatku [KCR98].
Standardia iso mapillinen.	Standardia 50 s.
Rikas kokoelma tietotyyppejä ja ilmaisuvoimainen oliojärjestelmä CLOS [S98].	Vain perustieto ja niiden yhdist ilmaisuvoimaiset op tiot, joilla tyypit ja tehtävissä.
Standardikirjasto rikas (koska paljon tyyppejä).	Standardikirjasto su (tee-se-itse).
Jonkin verran historiallista painolastia häiritsee ortogonaalisuutta.	Hyvin ortogonaa rakenne; "peru kat" yhdistyvät toi vapaasti.
Tarkoitettu suuriin ohjelmistohankkeisiin, mm. moduulit.	Ei omaa m lijärjestelmää, tarko periaatteiden opetu ja tutkimukseen.

II.3 Funktionaalinen ohjelmointi

Scheme-kieli liitetään usein ns. funktionaaliseen ohjelmointiin, vaikka sillä voi ohjelmoida myös muissa "paradigmoissa".

Tässä lähestymistavassa algoritmi esitetään funktioina joita sovelletaan abstraktisti esitettyyn dataan sen sijaan että komennettaisiin alla olevaa laskulaitetta käsittelemään datan konkreettista esitysmuotoa muistissa.

C-kielinen vastaesimerkki:

```
int foo(int x)
{
 static int y=0;
 return x+(y++);
}
```

Tämä *ei* ole funktio koska se antaa eri arvoja samalla argumentin `x` arvolla. Silloin esimerkiksi `foo(0)` + `foo(0)` $\neq 2 \cdot$ `foo(0)` eli tutut matemaattiset lait eivät pädekään!

Se ei edes ole "`foo`(z)$= z +$ aiempien `foo`-kutsujen lukumäärä", koska jollakin *laite*riippuvaisella kohdalla `y` pyörähtää ympäri!

Funktionaalinen ratkaisu on että myös `y` on fun parametri, koska arvo riippuu myös siitä: `foo`$(x, y) = (x + y, y + 1) \in \mathbb{Z}^2 \to \mathbb{Z}^2$. Arvon `y` lisäominaisuus on koko funktioin esitetyn algorit ei vain aliohjelman `foo`.

> Kahta ilmausta a ja b voi pitää merkitykseltään samoina täsmälleen silloin kun ilmaus a voidaan joka asiayhteydessä korvata ilmauksella b koko asiayhteyden merkityksen muuttumatta. *"Leibnizin periaat*

Yksi funktionaalisen ohjelmoinnin perusideoista suunnitella kieli siten, että merkityksen samuus voidaan päätellä katsomalla vain ilmauksia a ja niiden ympäristöä.

"Perinteisessä" *tilaperustaisessa* ohjelmoinnissa a ja b voidaan vaihtaa keskenään jos koko ohjelman (muistin) nykyinen tila sen sa eikä sitä voi tietää pelkästään vaihdokkaita a ja tutkimalla.

Tämä periaate taas on keskeinen symbolisessa järkeilyssä...

Scheme ei ole "puhdas" (pure) funktionaalinen koska silläkin voi kirjoittaa C-vastaesimerkkimm Esimerkiksi Haskell-kieli [T99] on puhdas.

II.4 Jakamattomat tietotyypit

Scheme on *piilevästi (latentisti, implisiittisesti)* tyypitetty kieli: tyypit ovat kyllä olemassa, mutta niitä ei kirjoiteta ohjelmakoodiin näkyviin, vaan ne tarkastetaan vasta ohjelmaa suoritettaessa. Tämä mahdollistaa esimerkiksi sen että sama funktio palauttaa tilanteesta riippuen eri tyyppisiä arvoja.

Joskus sanotaan *"tyypitön"* mutta tarkkaan ottaen se tarkoittaisi kieltä jossa on vain yksi tyyppi ja siten kaikki operaatiot soveltuvat kaikkiin alkioihin. Scheme-kielessä taas voi tulla ajonaikaisia virheitä: esimerkiksi totuusarvoja ei voi laskea yhteen.

Perinteisen näkyvästi tyypitettyjä funktionaalisia kieliä ovat esimerkiksi Haskell [T99] ja ML-kieliperhe [CM98]. Näihin kieliin on kuitenkin lisätty *tyypinpäättely* [H97,P02] eli kääntäjä päättelee tyypit itse ellei ohjelmoija niitä kirjoita. Nyt voi spesifikaation (osia) kirjoittaa tyyppeinä ja kääntäjän tarkistaa vastaako koodi sitä...

Aloitetaan tutustumalla *jakamattomiin (atomisi* tietotyyppeihin, joiden sisärakenteeseen ohjelmo siis pääse käsiksi. (Rakenteiset tyypit esitellään myöhemmin.)

Kaikki tyypit ovat *erillisiä*.

II.4.1 Totuusarvot

Yksinkertaisin jakamaton tietotyyppi ovat totuusarvot (Boolean) [KCR98,§6.3.1].

Arvoa "tosi" esittää vakio `#t` (**t**rue), arvoa "epätosi" taas vakio `#f` (**f**alse).

Kun Scheme-kielessä tutkitaan totuusarvoja (esimerkisi ehtolauseessa), *vain* `#f` *on epätosi ja kaikki muut tosia* arvoja.

Vakiokirjastossa on funktio `boolean?` joka palauttaa arvon `#t` jos sen argumentti on totuusarvovakio ja `#f` muuten.

Konventio: Totuusarvon palauttavan funktion nimi loppuu kysymysmerkkiin.

Jokaiselle Scheme-tyypille t on kirjastofunktio t? joilla voi tarkastaa argumentin tyypin.

Common Lisp: Vain tyhjä lista (esitellään kalvolla II.8.1) on epätosi.

Myös jotkut (vanhat) Scheme-toteutukset toimivat näin [D96,s. 113].

II.4.2 Numerot

Scheme-standardi määrittelee (Common Lispiä seuraten) *matemaattisesti* oikean numerohierarl (`number?`) [D96,§6.3;KCR98,§6.2] mutta sallii suppeammat toteutukset.

Kokonaisluvut (`integer?`) voivat olla *mielivalta* pitkiä. Niiden kokoa ei siis kone rajoita!

Murtoluvut (`rational?`) ovat näiden kokonaislukujen osamääriä.

Reaaliluvut (`real?`) ovat muista ohjelmointikie tuttuja äärellisen tarkkuuden liukulukuja.

Kokonais- ja murtoluvut ovat *tarkkoja* (`exac` liukuluvut *epä*tarkkoja (`inexact?`). Epätarkka voidaan muuntaa lähimmäksi tarkaksi naapurikseen (`inexact->exact`).

Konventio: Konversiorutiini tyypistä t tyyp u on nimeltään t`->`u.

Kompleksiluvut (`complex?`) koostuvat kahdesta edellä mainitusta luvusta: reaali- ja imaginääriosasta.

Kompleksilukuja voi käsitellä paitsi tässä suorakulmaisessa myös napakoordinaatistossa.

Kompleksiluku on tarkka vain jos molemmat koordinaatit ovat tarkkoja suorakulmaisessa koordinaatistossa.

Scheme tekee "mielekkäät" muunnokset laskutoimitustensa aikana, esimerkiksi:

- Jos kokonaisluvun jakaminen toisella kokonaisluvulla menee tasan, niin tulos on uusi kokonaisluku; muuten (supistettu) murtoluku.
- Jos kokonais- tai murtolukuun lisää reaaliluvun, niin tulos on reaaliluku – ja siis epätarkka.
- Jos kompleksiluvun imaginääriosasta tulee tarkka 0, niin tulos on reaaliosan tyyppiä.

Näin voidaan numeriikkaa tehdä matematiikan, ei koneen ehdoilla...

II.4.3 Symbolit

Symbolit ovat (tietenkin!) oma erityinen tyyppi (`symbol?`) [KCR98,§6.3.3].

Symbolin kirjoitusasu voi koostua kirjaimista, numeroista ja erikoismerkeistä `! $ % & * + - . = > ? @ ^ _ ~` kunhan sitä ei voi erehtyä pitämä numerovakiona.

- Ohjelmoija voi ajatella Schemen "tuntevan" kaikki mahdolliset symbolit etukäteen.

  Symboleita voi siis esimerkiksi lukea syöttee ilman eri ohjelmointia.

- Symboleiden keskeinen ominaisuus on, että symbolia ovat samat täsmälleen silloin kun kirjoitusasut ovat samat.

  Analoginen esimerkki: Kun meitä pyydetään matemaattiseen lausekkeeseen $x + y + x$ arvo $x = 5$, niin vastaamme $5 + y + 5$. Tulkitsemme pyynnön siis tarkoittavan, että kaikki saman symbolin x eri esiintymät korv vakiolla 5, mutta eri symbolia y ei korvata.

  Samuuteen palataan kalvoilla II.8.5.

- Käytännössä toteutus ylläpitää symbolitaulua näkemistään symboleista, ja nähdessään uuden symboliesiintymän katsoo taulusta, onko kyseessä tuttu vai uusi symboli. Kummassakin tapauksessa toteutuksella on sille *yksikäsitteinen* edustuma.

  (Vertaa Leibnizin periaatteeseen kalvoilla II.3.)

Symboleihin palataan tarkemmin rakenteisten tietotyyppien yhteydessä kalvoilla II.8: yksittäisillä symboleilla ei voi juurikaan operoida, kun taas niiden yhdistelmiä voi käsitellä monin tavoin.

II.5 Kielen ydin

Esitetään aluksi *yksinkertaistetun* Scheme-kielen
syntaksi ja semantiikka [AS96,§1.1] jota
täydennetään myöhemmin uusilla tietorakenteill
operaatioilla.

Tämä tehdään määrittelemällä *induktiivisesti*

lausekkeet eli Scheme-ohjelmien kirjoitusasu,

arvot joita nämä lausekkeet esittävät ja

laskentasäännöt joilla lausekkeen esittämä arv
selviää.

Kalvojen II.4.1 ja II.4.2 totuusarvo- ja numerova
ovat lausekkeita.

Näiden vakioiden arvot ovat tietenkin ne itse.

Vastaava laskentasääntö ei siis tee mitään.

II.5.1 Muuttujanmäärittelyt

Muuttuja määritellään ja sille annetaan arvo seuraavasti:

(**define** *nimi lauseke*)

esittelee muuttujan nimeltä *nimi*, ja sen arvo saadaan laskemalla minkä arvon a saa määrittelevä *lauseke*.

Vastaava "laskentasääntö" esittelee nimen (peittäen mahdollisen aiemman esittelyn) ja liittää siihen vastaavan arvon a. Määrittelyllä itsellään ei ole arvoa, eikä se siis voi olla lauseke.

Tässä vaiheessa muuttujat ovat *matemaattisia:* niillä on kiinteä (tosin tunnettu) vakioarvo, ne eivät (vielä) ole muokattavien muistipaikkojen nimiä, kuten tilaperusteisessa ohjelmoinnissa. (Muokkaus esitellään kalvoilla II.9.3.)

Muuttujaviittaukset ovat lausekkeita, ja ne tarkoittavat tietenkin tätä liitettyä arvoa a. Siten

(**define** *h* 1/1000)

määrittelee muuttujan *h* ja liittää siihen arvon $\frac{1}{1000}$.

Muuttujannimien syntaksi on sama kuin symbol kalvoilta II.4.3.

– Sekaannuksen vaara on ilmeinen!

  Tässä vaiheessa riittää seuraava sääntö: muotoa *'nimi* oleva esiintymä on symboli, il heittomerkkiä oleva esiintymä taas muuttuja Sääntö tarkentuu jatkossa kun tutustumme kielen uusiin piirteisiin.

  Lauseke *'nimi* on siis symbolityypin vakio.

  Heittomerkkiin palataan kalvoilla II.8.4.

+ Scheme-ohjelmien lähdekielinen muoto (jota tässä juuri määritellään) on itsekin symbolie yhdistelmä.

  On siis helppoa kirjoittaa Scheme-ohjelmia, lukevat ja tulostavat muiden Scheme-ohjelm lähdekoodeja.

  (Monissa nykyisissä Scheme-toteutuksissa v jopa suorittaa symbolisessa muodossa oleva vaikkapa syötteenä saatua – koodia.)

Vastaavanlaisilla määritelmillä esitellään myös funktiot kalvolla II.5.2. Scheme-ohjelma koostuu jonosta tällaisia määritelmiä ja lausekkeita.

Tällaisen ohjelman suoritus lukee tätä jonoa järjestyksessä, ja tavatessaan

määritelmän saattaa voimaan vastaavan esittelyn

lausekkeen laskee sen arvon a tällä hetkellä voimassa olevilla esittelyillä, ja tulostaa sen käyttäjälle.

(Näin toimii Scheme-toteutuksen komentorivi eli "REPL" eli "Read-Eval-Print Loop" eli "lue-laske-tulosta-silmukka".)

Common Lisp: Muuttujanmäärittely onkin muotoa

(`defvar` *nimi lauseke*)

II.5.2 Funktionmäärittelyt

Funktionmäärittely on muotoa

(`define` (*nimi parametri*$_1$ ... *parametri*$_k$) *lause*

joka esittelee muuttujan *nimi* ja antaa sen arvo erään k-argumenttisen funktion, joka on intuitiiv "*lausekkeen* arvo kun kunkin (nimetyn) *parame* tilalla on kutsun vastaava *argumentti* i".

Rekursio on sallittua: funktion *nimi* määrittelev *lauseke* saa kutsua itseään sekä muita ohjelmar funktioita.

Näin määriteltu funktio selostetaan sen kutsuje yhteydessä kalvolla II.5.3.

Vakiokirjasto tarjoaa valmiita funktioita (esimer kalvojen II.4.2 numeriikkaan); ne on esitelty valr

Common Lisp: Funktionmäärittely onkin muot

(`defun` *nimi* (*parametri*$_1$ ... *parametri*$_k$) *laus*

II.5.3 Funktionkutsut

Lauseke

$$(\textit{nimi}\ \textit{lauseke}_1 \ldots \textit{lauseke}_k)$$

on intuitiivisesti "edellä määritellyn funktion *nimi* kutsu argumenteilla *nimi* $\textit{lauseke}_1 \ldots \textit{lauseke}_k$".

Tämän kutsun palauttama arvo lasketaan seuraavasti:

- Lasketaan samoilla säännöillä minkä arvon $\textit{argumentti}_i$ kukin $\textit{lauseke}_i$ saa.
- Haetaan kalvon II.5.2 mukainen määrittely

  (`define` (*nimi* $\textit{parametri}_1 \ldots \textit{parametri}_k$) *lauseke*).
- Tehdään sen rungosta *lauseke* sellainen *kopio* jossa kukin muuttuja $\textit{parametri}_i$ saa arvokseen $\textit{argumentti}_i$.
- Jatketaan laskemalla arvo tälle *kopio*lle.

Siis *funktion kutsu korvataan sen määrittelylausekkeella jossa määrittelyaikaiset parametrit on korvattu kutsuaikaisilla argumenteilla*.

II.5.4 Ehtolauseke

Kalvon II.5.3 funktionkutsuperiaate ei sovellu lausekkeisiin muotoa

(`if` $\textit{lauseke}_{ehto}$ $\textit{lauseke}_{totta}$ $\textit{lauseke}_{epätotta}$

joiden intuitio on "jos $\textit{lauseke}_{ehto}$ on tosi niin si
arvo on $\textit{lauseke}_{totta}$ mutta muuten arvo onkin
$\textit{lauseke}_{epätotta}$", koska kutsussa laskettaisiin ens
jokainen $\textit{lauseke}_{ehto/totta/epätotta}$.

Niinpä sen arvo lasketaankin seuraavasti:

1. Ensin lasketaan minkä arvon a saa $\textit{lauseke}_{el}$
2. Jos a on `#f` (kalvolta II.4.1), niin jatketaan laskemalla minkä arvon saa $\textit{lauseke}_{epätotta}$.
3. Muuten jatketaankin laskemalla minkä arvon saa $\textit{lauseke}_{totta}$.

Siis *ehtolause korvataan sen totta- tai epätotta-lausekkeella, riippuen siitä kummaksi ehto-lauseke osoittautuu*.

II.5.5 Arvon laskenta symbolipelinä

Kerrataan näiden kalvojen II.5 asiat hieman täsmällisemmästä näkökulmasta esittämällä sama mekanismi symbolisena pelinä kalvojen I.3 D-sääntöjen tapaan.

Määritellään induktiivisesti seuraavat käsitteet:

Vakioarvot koostuvat kalvojen II.4.1, II.4.2 ja II.5.1 totuusarvo- numero- ja symbolityyppien vakioista. Vakioarvon *tulostusasu* on se itse.

Lausekkeet koostuvat vakioarvoista, muuttujanesiintymistä *nimi*, sekä kalvojen II.5.3 ja II.5.4 funktionkutsu- ja ehtolausekkeista.

Arvot koostuvat vakioarvoista ja *funktioarvo*ista.

Funktioarvo on pari $\langle \mathit{sanakirja}, (\mathit{nimi}_1, \ldots, \mathit{nimi}_n) \mapsto \mathit{lauseke} \rangle$.

Funktioarvolle ei ole määritelty standardimuotoista tulostusasua; toteutus `drscheme` käyttää asua `#<procedure:nimi>` käyttäjän määrittelemille funktioille ja asua `#<primitive:nimi>` kirjastofunktioille.

Sanakirjat ovat äärellisiä jonoja pareista $\langle \mathit{nimi}, \ldots$

Sanakirjoihin talletetaan tiedot tällä hetkellä voimassa olevista määrittelyistä sekä niihin liittyvistä arvoista.

Sanakirjaa luetaan alusta järjestyksessä, kun annetulle nimelle etsitään vastaavaa arvoa.

Arvonlaskentafunktio yrittää laskea sen arvon jonka *lauseke* saa, kun voimassa olevat määrittelyt ja arvot antaa *sanakirja*.

Tämä funktio määritellään induktiivisesti tapauksittain *päättelysäännöillä* muotoa

$$\frac{\text{ne ehdot joilla tämä sääntö soveltuu}}{\mathit{lauseke}; \mathit{sanakirja} \vdash_{\mathsf{eval}} \mathit{arvo}}.$$

$$\frac{c \text{ on vakioarvo}}{c; \mathcal{S} \vdash_{\mathsf{eval}} c}$$

$$\frac{\text{(ei muita ehtoja)}}{x; \langle x, a \rangle \mathcal{S} \vdash_{\mathsf{eval}} a}$$

$$\frac{x \text{ ja } y \text{ ovat eri nimet} \quad x; \mathcal{S} \vdash_{\mathsf{eval}} a}{x; \langle y, a \rangle \mathcal{S} \vdash_{\mathsf{eval}} a}$$

$$\frac{l_1;\mathcal{S}\vdash_{\mathsf{eval}} \texttt{\#f} \quad l_3;\mathcal{S}\vdash_{\mathsf{eval}} a}{(\texttt{if}\ l_1\ l_2\ l_3);\mathcal{S}\vdash_{\mathsf{eval}} a}$$

$$\frac{l_1;\mathcal{S}\vdash_{\mathsf{eval}} b \quad b \text{ on eri arvo kuin } \texttt{\#f} \quad l_2;\mathcal{S}\vdash_{\mathsf{eval}} a}{(\texttt{if}\ l_1\ l_2\ l_3);\mathcal{S}\vdash_{\mathsf{eval}} a}$$

Funktiokutsusäännössä tehdään pieni tekninen temppu: kutsuttava funktio säilytetään sanakirjassa, jotta rekursio olisi mahdollista.

$$\frac{\begin{array}{c} f;\mathcal{S}\vdash_{\mathsf{eval}} \langle\mathcal{T},(x_1,\ldots,x_n)\mapsto g\rangle \\ l_1;\mathcal{S}\vdash_{\mathsf{eval}} a_1\ \cdots\ l_n;\mathcal{S}\vdash_{\mathsf{eval}} a_n \\ g;\langle x_1,a_1\rangle\ldots\langle x_n,a_n\rangle\,\langle f,\langle\mathcal{T},(x_1,\ldots,x_n)\mapsto g\rangle\rangle\,\mathcal{T}\vdash_{\mathsf{eval}} a \end{array}}{(f\ l_1\ \ldots\ l_n);\mathcal{S}\vdash_{\mathsf{eval}} a}$$

Lisäksi tarvitaan säännöt jakamattomien tietoalkioiden perusoperaatioille standardikirjastossa, esimerkiksi

$$\frac{l_1;\mathcal{S}\vdash_{\mathsf{eval}} \frac{p_1}{q_1} \quad \ldots \quad l_n;\mathcal{S}\vdash_{\mathsf{eval}} \frac{p_n}{q_n} \quad \frac{p_1\cdot\cdots\cdot p_n}{q_1\cdot\cdots\cdot q_n}=a}{(*\ l_1\ \ldots\ l_n);\mathcal{S}\vdash_{\mathsf{eval}} a}.$$

Määrittelyt koostuvat kalvojen II.5.1 ja II.5.2 muuttujan- ja funktionmäärittelyistä.

Ohjelma on yksinkertaistaen:

- äärellinen jono määrittelyitä jotka saatet voimaan samassa järjestyksessä.
- Määrittelyjonon lopuksi tulee vielä se lau jonka arvo näillä määrittelyillä halutaan tulostaa.

  (Oikea REPL jäisi odottamaan lisää määrittelyjä ja lausekkeita.)
- Jonossa seuraava määriteltävä nimi on a eri kuin mikään sitä edeltänyt määritelty

  (Oikea Scheme sallisi nimen uudelleenmäärittelyn.)
- Jonossa seuraavan määriteltävän nimen arvolauseke viittaa vain sellaisiin nimiin jo määrittelyt ovat edeltäneet sitä.

  (Oikea Scheme sallisi molemminpuolisen (mutual) rekursion.)
- Ohjelman alussa on määrittelyt standardikirjaston niille funktioille joille e anneta erillisiä sääntöjä.

Suoritus määritellään jälleen sääntöinä.

$$\frac{g;\mathcal{S}\vdash_{\text{eval}} h \qquad \mathcal{O};\langle n,h\rangle\,\mathcal{S}\vdash_{\text{load}} a}{\texttt{(define }\ n\ g\texttt{)}\mathcal{O};\mathcal{S}\vdash_{\text{load}} a}$$

Tässä on rekursiotempun alku: funktioarvon pitäisi oikeasti viitata siihen samaan sanakirjaan, jonka lopussa se itse on — eli olla kehä.

$$\frac{\mathcal{O};\langle n,\langle\mathcal{S},(x_1,\ldots,x_n)\mapsto g\rangle\rangle\,\mathcal{S}\vdash_{\text{load}} a}{\texttt{(define (}n\ x_1\ \ldots\ x_n\texttt{)}\ g\texttt{)}\mathcal{O};\mathcal{S}\vdash_{\text{load}} a}$$

$$\frac{l;\mathcal{S}\vdash_{\text{eval}} a}{l;\mathcal{S}\vdash_{\text{load}} a}$$

Jos päädytään tilanteeseen, jossa mikään sääntö ei sovellu, niin suoritus jumittuu ajonaikaiseen virheeseen.

Matemaattis-loogisen tyyppiteorian soveltaminen ohjelmointikieliin alkoi kysymyksestä "miten kääntäjä voisi nähdä lähdekoodista voiko ohjelma jumittua?"

Tällaista formalisoitua suoritussemantiikkaa voi käyttää esimerkiksi tarkkana spesifikaationa kieltä toteutettaessa.

Lasketaan esimerkkinä käsin luvun 1 kertoma käyttäen näitä sääntöjä.

(Laskenta on paloiteltu ulkoasusyistä.)

$$\frac{\dfrac{}{\texttt{fac};\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L},(n)\mapsto\phi\rangle} \quad \dfrac{\text{1 on vakio}}{1;\mathcal{S}\vdash_{\text{eval}} 1} \quad \phi;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}{\texttt{(fac 1)};\underbrace{\langle\texttt{fac},\langle\mathcal{L},(n)\mapsto\underbrace{\texttt{(if (zero? }n\texttt{) 1 (* }n\texttt{ (fac (- }n\texttt{ 1))))}}_{\text{merkitään }\phi}\rangle\rangle\mathcal{L}}_{\text{merkitään }\mathcal{S}}\vdash \ldots}$$

$$\frac{\texttt{(zero? }n\texttt{)};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}}\texttt{\#f} \qquad \texttt{(* }n\texttt{ (fac (- }n\texttt{ 1)))};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}{\phi;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}$$

$$\frac{\dfrac{\dfrac{\dfrac{\text{standardikirjasto }\mathcal{L}}{\vdots}}{\texttt{zero?};\mathcal{L}\vdash_{\text{eval}}\langle\mathcal{L}',(m)\mapsto\psi\rangle}}{\dfrac{\texttt{zero?};\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L}',(m)\mapsto\psi\rangle}{\texttt{zero?};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L}',(m)\mapsto\psi\rangle}} \quad \dfrac{}{n;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1} \quad \dfrac{\psi\text{:n spesifika}\ldots}{\vdots \atop \psi;\langle m,1\rangle\,\mathcal{L}'\vdash_{\text{e}}\ldots}}{\texttt{(zero? }n\texttt{)};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}}\texttt{\#f}}$$

$$\frac{\dfrac{}{n;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1} \quad \texttt{(fac (- }n\texttt{ 1))};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1 \quad 1\cdot 1=1}{\texttt{(* }n\texttt{ (fac (- }n\texttt{ 1)))};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}$$

$$\frac{\dfrac{\dfrac{}{\texttt{fac};\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L},(n)\mapsto\phi\rangle}}{\texttt{fac};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L},(n)\mapsto\phi\rangle} \quad \texttt{(- }n\texttt{ 1)};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 0 \quad \phi;\langle n,0\rangle\,\mathcal{S}\vdash\ldots}{\texttt{(fac (- }n\texttt{ 1))};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}$$

$$\frac{\dfrac{}{n;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1} \quad \dfrac{\text{1 on vakio}}{1;\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 1} \quad 1-1=0}{\texttt{(- }n\texttt{ 1)};\langle n,1\rangle\,\mathcal{S}\vdash_{\text{eval}} 0}$$

$$\frac{\texttt{(zero? }n\texttt{)};\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}}\texttt{\#t} \quad \dfrac{\text{1 on vakio}}{1;\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}}{\phi;\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}} 1}$$

$$\frac{\dfrac{\dfrac{\vdots}{\texttt{zero?};\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L}',(m)\mapsto\psi\rangle}}{\texttt{zero?};\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}}\langle\mathcal{L}',(m)\mapsto\psi\rangle} \quad \dfrac{}{n;\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}} 0} \quad \dfrac{\psi\text{:n spesifika}\ldots}{\vdots \atop \psi;\langle m,0\rangle\,\mathcal{L}'\vdash_{\text{e}}\ldots}}{\texttt{(zero? }n\texttt{)};\langle n,0\rangle\,\mathcal{S}\vdash_{\text{eval}}\texttt{\#t}}$$

II.6 Korkeampaa ohjelmointia

Tarkennetaan kalvoilla II.5 esiteltyä mallia sallimaan vakioiden lisäksi *funktiot arvoina* joita voidaan antaa argumentteina ja palauttaa tuloksina [AS96,§1.3].

Tätä mahdollisuutta kutsutaan *korkeamman kertaluvun funktioiksi/ohjelmoinniksi* koska

- "passiivisia" tietoalkioita kuten kalvojen II.4 vakioita kutsutaan kertaluvun 0 olioiksi ja
- funktion f kertaluku on sen argumenttien ja tulosten kertalukujen maksimi + 1; f näet operoi yhtä alemman kertaluvun alkioilla.

Esimerkiksi kalvojen I.3 derivaatta raja-arvona määrittelee funktion (joskus sanotaan *funktionaalin*) joka kuvaa reaalifunktion $\mathbb{R} \to \mathbb{R}$ toiseksi reaalifunktioksi. Jos reaalilukujen kertaluvuksi otetaan 0 niin reaalifunktioiden kertaluku on 1 ja derivoinnin siis 2.

Derivointi*säännöissä* "tekstimuotoisia" lausekkeita pidetään taas passiivisina (kertaluvun 0) alkioina, joten itse säännöt ovat kertalukua 1.

Scheme-kielen sanotaan olevan "kertalukua ω" sillä voi kirjoittaa mitä tahansa kertalukua $k \in \mathbb{N}$ olevia funktioita, ja $|\mathbb{N}| = \omega$.

Tämä tarjoaa mahdollisuuden ilmaista laskentatehtäviä entistä abstraktimmin.

Tarkastellaan numeerisena esimerkkinä *Newton menetelmää* reaalifunktioiden nollakohtien löytämiseksi. Jono

$$x_{n+1} = x_n - \frac{f(x_n)}{\mathbf{D}f(x_n)}$$

lähestyy funktion $f : \mathbb{R} \to \mathbb{R}$ nollakohtaa (tai pää derivaatan nollakohtaan ja jumittuu).

Numeriikassa kalvojen I.3 oikean derivaattafunkt tilalle otetaan

$$\mathbf{D}f(x) \approx \frac{f(x+h) - f(x)}{h}.$$

"sopivan pienellä" h joka määrää halutun vastaustarkkuuden, ja jatketaan kunnes $|f(x_n)|$

Voimme ottaa vakion `h` vaikkapa kalvolta II.5.1.

Sovelletaan menetelmää luvun $\sqrt{2}$ likiarvon määräämiseen, eli etsitään funktion $f(x) = x^2 - 2$ nollakohtaa.

Ensimmäinen versiomme on ensimmäistä kertalukua:

```
(define h 1/1000)

(define (f2 x) ; "Nollattava" funktio f.
  (- (* x x) 2))

(define (d x)
  (/ (- (f2 (+ x h)) (f2 x)) h))

(define (newton x)
  (if (< (abs (f2 x)) h)
      x
      (newton (- x (/ (f2 x) (d x))))))
```

II.6.1 Funktiot toistensa parametreina

Newtonin menetelmä sopii muillekin funktioille f, ja on vaivalloista kirjoittaa uusi `newton` jokaiselle uudelle funktiolle. Olisi abstraktimpaa jos yksi `n` voisi saada parametrinään sen funktion jota käsittelee; tästähän menetelmässä on kyse.

Haluamme siis välittää funktioon `newton` parame toisen funktion `f2`. Tämä käy suoraan, koska ka II.5.3 funktiokutsussa

$$(\mathit{nimi}\ \mathit{lauseke}_1 \ldots \mathit{lauseke}_k)$$

voi kutsuttavan funktion *nimi* olla tämän nykyis funktion jokin *parametri* ja silloin kutsutaan tuo parametrinä saatua funktiota argumenteilla *laus* Funktio `newton` nousi toiseen kertalukuun:

```
(define h 1/1000)

(define (f2 x)
  (- (* x x) 2))

(define (d f x)
  (/ (- (f (+ x h)) (f x)) h))

(define (newton f x)
  (if (< (abs (f x)) h)
      x
      (newton f (- x (/ (f x) (d f x))))))
```

II.6.2 Paikalliset määritelmät

Seuraava (väli)askel abstrahoinnissamme on funktioiden määrittely toisten funktioiden sisällä, jolloin ne eivät tule kaikkien tietoon. Tämä parantaa modulaarisuutta.

Samalla astumme ulos kalvojen II.5.5 yksinkertaisesta mallista.

Siirretään "derivaattafunktio" `d` pääfunktionsa `newton` sisään. Samalla huomataan että `d` saa parametrinä saman `f` kuin `newton`, joten sitä ei tarvitse toistaa:

```
(define h 1/1000)

(define (f2 x)
  (- (* x x) 2))

(define (newton f x)
  (define (d x)
    (/ (- (f (+ x h)) (f x)) h))
  (if (< (abs (f x)) h)
      x
      (newton f (- x (/ (f x) (d x))))))
```

Siis `define`-funktionmäärittelyn sisään saa kirjoit toisia `define`-funktionmäärittelyjä ennen määrittelylauseketta.

(Myös `define`-muuttujanmäärittelyjä voisi käytt sisäisinä, mutta vain tietyin teknisin lisäehdoin [KCR98,§5.2.2].)

Tämä sisäinen määrittelyalue on sen määrittelyalueen paikallinen jatke jossa sitä ympäröivä `define`-funktionmäärittely on:

- Sisäisiin määritelmiin näkyvät samat asiat k ympäröivän määrittelyn lausekkeeseenkin, el sisäinen määrittely itse niitä peitä:

  `h` viittaa funktiossa `d` samaan arvoon kuin funktiossa `newton`, eli julkiseen vakioon $\frac{1}{1000}$

  kun taas `x` viittaa funktiossa `d` sen yksityisee parametriin, ei funktion `newtona` samannimis parametriin joka siis peittyi.

- Sisäiset määritelmät näkyvät toisilleen ja ympäröivän määrittelyn lausekkeeseen, mutt eivät sen pidemmälle.

Scheme noudattaa siis muista lohkokielistä tutt *leksikaalisia näkyvyyssääntöjä* (lexical scope). (E Common Lisp -standardin tuloa se oli poikkeuksellista.)

II.6.3 Paikalliset funktiot tuloksina

Funktioita voidaan paitsi saada parametreina myös *palauttaa funktioista niiden arvoina*. Yhdistettynä kalvojen II.6.2 paikallisiin määritelmiin syntyy mahdollisuus rakentaa uusia funktioita:

```
(define h 1/1000)

(define (g a)
  (define (f2 x)
    (- (* x x) a))
  f2)

(define (newton f x)
  (define (d x)
    (/ (- (f (+ x h)) (f x)) h))
  (define (newton-apu x)
    (if (< (abs (f x)) h)
        x
        (newton-apu (- x (/ (f x) (d x))))))
  (newton-apu x))
```

Funktio `g` on nyt "funktiotehdas" joka palautta sisäisestä funktiostaan `f2` eri versioita riippuen parametrin `a` arvosta: `(g 2)` on tuttu $x \mapsto x^2 - 2$ `(g 5)` on taas $x \mapsto x^2 - 5$,...

Funktio `g` palauttaa *tiedon siitä mikä funktio* `f2` *nykyisessä määrittely-ympäristössään* johon siis kuuluu se parametrin `a` arvo, jonka `g` tällä kerta

Vertaa kalvojen II.5.5 funktioarvoihin, jotka koostuivat määrittelyhetkisestä sanakirjasta ja lausekkeesta!

Ohjelmointikielessä

C funktioita ei edes voi määritellä sisäkkäin.

Pascal funktioita voi määritellä sisäkkäin, mutt sisäfunktiota ei saa palauttaa arvona, koska määrittely-ympäristö katoaa.

Esimerkissämme siis `a` katoaa kun `g` palaa kutsujaansa, jolloin sen arvo on "rikki".

Näissä kielissä voi kyllä välittää funktioiden nimiä/osoitteita, mutta korkeamman kertaluvur kieleen tarvitaan siis muutakin.

Funktion sisäiset määritelmät tehdään joka kutsulla uudelleen. (Tämän vuoksi otettiin käyttöön myös sinänsä tarpeeton apufunktio `newton-apu`.)

Näitä "teollisesti" luotuja funktioita voi käyttää vapaasti, esimerkiksi muiden funktioiden parametreinä:

```
(define (neliojuuri x)
  (newton (g x) x))
```

on yleisen neliöjuurifunktion $x \mapsto \sqrt{x}$ esitys muotoa "etsi Newtonin menetelmällä funktion $y \mapsto y^2 - x$ nollakohta alkuarvauksena x itse".

Tuloksena palautettu funktio on siis kaksiosainen paketti:

määrittely-ympäristö joka sisältää kaikkien paikallisten(kin) muuttujien arvot

lauseke jonka arvoja voidaan laskea funktion kutsussa annettavien parametrien ja paikallisten muuttujien arvoilla.

Olio-ohjelmoinnissa sen vastine on seuraava:

instanssin tietokentät vastaavat määrittely-ympäristöä

ainoa metodi vastaa lauseketta.

Rajoittuminen yhteen metodiin ei ole oleellista: sehän voi vaikkapa toimia "puhelinkeskuksena" välittää kunkin syötteenään saamansa (symboli metodikutsun oikeaan osoitteseensa.

$$\underbrace{(\underbrace{(\textit{olio}\ '\textit{metodi})}_{\text{haetaan metodi}}\ \textit{parametrit})}_{\text{kutsutaan haettua metodia}}$$

Alkuperäisessä SmallTalk-kielisessä olio-ohjelmoinnissa puhutaankin metodikutsujen sijasta sanomien (message) välittämisestä ja ni vastaamisesta.

Esimerkiksi tällaisilla ideoilla voikin lähteä esittämään olioita funktionaalisesti [P02,§18].

Esimerkkimme on tässä valossa seuraava (ohjelmointikielen C++ [S97] tyyliin):

- Määritellään luokka `g` joka sisältää kentän `a`, konstruktorin ja yhden metodin:

```
class g
{
  float a;
 public:
  g(float b);
  float f2(float x);
}
```

  Jos `f2` olisi määritelty vieläkin syvemmällä, niin jokainen välitasokin pitäisi "olioida".

- Luokan (oletus)konstruktori alustaa tämän kentän argumenttinsa arvolla:

```
g::g(float b)
 :a(b)
{
}
```

- Luokan ainoa metodi saa argumentin `x` ja palauttaa arvon `x*x-a`:

```
float g::f2(float x)
{
  return x*x-a;
}
```

- Funktionkutsua `(g 2)` vastaa nyt konstruktorikutsu:

```
g o(2);
```

  Se luo tulosfunktiota $f\colon x \mapsto x^2 - 2$ vastaava olion `o`.

- Tämän tulosfunktion f kutsua argumentilla vastaa metodikutsu:

```
o::f2(2);
```

II.6.4 Nimettömät funktiot

Kalvojen II.6.3 "funktiotehdas" `g` osoittaa että *funktio voidaan luoda nimeämättä sitä*: sisäinen funktio `f2` on tosin nimetty, mutta tämä nimi ei näy funktiosta `g` ulos vaikka siihen tällä kertaa liitetty funktio lähteekin funktion `g` arvona.

Scheme-kielessä onkin *funktiontekolauseke*

(`lambda` (*par*$_1$ ... *par*$_k$) *laus*).

Sen arvoksi luodaan uusi nimetön funktio, joka toimii ennalta-arvattavasti: kun sitä kutsutaan, argumentit kopioidaan parametrien tilalle lausekkeeseen ja jatketaan.

Vastaavasti

(`define` (*nimi par*$_1$ ... *par*$_k$) *laus*)

on lyhenne usein tarvitulle nimennälle

(`define` *nimi* (`lambda` (*par*$_1$ ... *par*$_k$) *laus*))

joka luo lausekkeesta *laus* nimettömän funktion parametreilla *par*$_i$ ja laittaa sen esiteltävän muuttujan *nimi* arvoksi.

Siis (`define` *nimi lauseke*) onkin aina *nimeämisoperaatio*, jossa nimeen liitetty lausekk arvo voi olla mitä tyyppiä tahansa: totuusarvo, numero, *funktio*,...

Kuitenkin *nimi* on rekursiivinen *lausekkeessa*. V ajatella, että ensin varataan tilaa jolle annetaan ja vasta sitten lasketaan arvo.

Funktionkutsussa (*lauseke*$_0$ ... *lauseke*$_k$) sallitaa ennen kutsuttavan funktion nimelle varatussa ensimmäisessä paikassa mielivaltainen lauseke *lauseke*$_0$, jolle lasketaan arvo samoin kuin muille lausekkeille. Ei ole väliä saatiinko arvo

- paikallisen tai julkisen `define`-määrittelyn lausekkeesta (normaalisti nimetty funktio)
- jonkin tätä kutsulauseketta ympäröivän `lambda`-lausekkeen parametrilistasta (parame saatu funktio) vai
- laskemalla arvo jollekin `lambda`-lausekkeelle (luotu nimetön funktio)

kunhan se vain on k-parametrinen funktio, jotta voidaan jatkaa.

Common Lisp: Nimipaikka käsitellään toisin säännöin kuin muut funktiokutsun paikat.

Eräs yllättävä mutta yleinen nimettömän funktion käyttötapa on

`((lambda (`$var_1 \ldots var_k$`)` *lauseke*`)` $val_1 \ldots val_k$`)`

jossa luodaan ensin *lausekkeesta* nimetön funktio parametrein var_i, ja kutsutaan sitä heti perään lausekkeiden val_i arvoilla.

Tuloksena on siis *lauseke* jossa jokainen symbolin var_i esiintymä on korvattu lausekkeen val_i arvolla, mutta nämä arvot on laskettu vain kerran.

Toisin sanoen, *lauseke* sai näin paikalliset muuttujat var_i alustettuina arvoilla val_i. Erillistä mekanismia ei tähän siis tarvita.

Tämä on niin tavallista ja hyödyllistä, että sille on määritelty oma notaatio

`(let ((`var_1 val_1`)...(`var_k val_k`))` *lauseke*`)`

joka voidaan lukea "olkoon var_1 yhtä kuin val_1 ja ...ja var_k yhtä kuin val_k tässä *lausekkeessa*". (Nyt var_i-esiintymät val_j-lausekkeissa eivät siis ole rekursiivisia.)

Täysi Scheme-syntaksi sisältää monia muitakin näistä perusrakenteista *johdettuja lausekkeita*.

Operaatio "ota annetusta luvusta p:s juuri" on

```
(define h 1/1000)

(define (newton f x)
  (define (d x)
    (/ (- (f (+ x h)) (f x)) h))
  (define (newton-apu x)
    (if (< (abs (f x)) h)
        x
        (newton-apu (- x (/ (f x) (d x))))))
  (newton-apu x))

(define (juuri p)
  (lambda (x)
    (newton (lambda (y)
              (- (expt y p) x))
            x)))

(define neliojuuri (juuri 2))
(define kuutiojuuri (juuri 3))
```

eli "se funktio joka kuvaa annetun luvun x funk $y \mapsto y^p - x$ nollakohdalle (ja tekee sen Newtonin menetelmällä alkuarvauksenaan x itse)".

Samalla saatiin muillekin funktioille hyödyllinen operaatio `newton`.

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.