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 accordance with the limitations and incapacities of the human misunderstanding. *Ambrose Bierce*

Otetaan esimerkiksi tuttu Aristotelinen syllogismi:

1. Sokrates on ihminen.

2. Kaikki ihmiset ovat kuolevaisia.

3. Siispä myös Sokrates on kuolevainen.

Tätä järkeillessämme käytämme *symboleita*: ”Sokrates”, ”(olla) ihminen”, ”(olla) kuolevainen”.

Emme käytä niiden *vastineita* todellisuudessa: perunanenäinen partainen kreikkalainen, kaikkien 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 sovellusalueen käsitteet näkyvät ohjelman symboleissa ja niillä 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 on mekaanisesti laskettavissa oleva kuvaus sen tapauksilta (”annettu kaksi lukua”) oikealle vastaukselle (”niiden summa”).

  Käytännössä se on usein (i) kuvaus tapauksilta laitteen muistissa olevaksi rakenteeksi, (ii) tämän 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 merkitystä:

Määritelmä erotusosamäärän raja-arvona

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

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

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

Säännöt todistetaan oikeiksi osoittamalla että ne 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!

(Integraalilausekkeelle ei ole vastaavaa peliä: ei ole takeita siitä että valittu sääntö olisi ”oikea”.)

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: Miten uusia derivointisääntöjä lisätään?

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

  Tuloksena on ylläpidon painajainen.

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

  Tuloksena on pieni muun ohjelman sisään upotettu symbolinen ohjelmointikieli: Mathematica, Maple, AutoCAD (AutoLISP!), (oikea) Emacs, ...

Symbolisella ohjelmointikielellä voi usein kiertää tämän *tietämyksenesitysongelman* ja kirjoittaa säännöt (lähes) suoraan ohjelmana.

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 processing) -murre.

II.1 Historiaa

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

Vain alkuperäinen Fortran on (vähän) vanhempi!

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

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 [D96] mutta kehitys jatkuu yhä [KCR98].
Standardia iso mapillinen.	Standardia 50 s.
Rikas kokoelma tietotyyppejä ja ilmaisuvoimainen oliojärjestelmä CLOS [S98].	Vain perustietotyypit ja niiden yhdistelyyn ilmaisuvoimaiset operaatiot, joilla tyypit ja oliot tehtävissä.
Standardikirjasto rikas (koska paljon tyyppejä).	Standardikirjasto suppea (tee-se-itse).
Jonkin verran historiallista painolastia häiritsee ortogonaalisuutta.	Hyvin ortogonaalinen rakenne; ”peruspalikat” yhdistyvät toisiinsa vapaasti.
Tarkoitettu suuriin ohjelmistohankkeisiin, mm. moduulit.	Ei omaa moduulijärjestelmää, tarkoitettu periaatteiden opetukseen 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 funktion parametri, koska arvo riippuu myös siitä: $\texttt{foo}(x, y) = (x + y, y + 1) \in \mathbb{Z}^2 \to \mathbb{Z}^2$. Arvon `y` lisäominaisuus on koko funktioin esitetyn algoritmin, ei vain aliohjelman `foo`.

Yleisemmin funktionaalisessa ohjelmoinnissa pyritään ylläpitämään seuraava ominaisuus:

> Jos kaksi ilmausta a ja b ovat kaikin tavoin ekvivalentit, niin ilmaus a voidaan joka asiayhteydessä korvata ilmauksella b merkityksen muuttumatta. *"Leibnizin periaate"*

"Perinteinen" *tilaperustainen* ohjelmointi ei noudata tätä periaatetta – a ja b voidaan vaihtaa keskenään vain jos koko ohjelman nykytila sen sallii, eikä sitä voi tietää pelkästään vaihdokkaita a ja b tutkimalla.

Tämä periaate taas on keskeinen symbolisessa järkeilyssä ...

Scheme ei ole "puhdas" (pure) funktionaalinen kieli, koska silläkin voi kirjoittaa C-vastaesimerkkimme. 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].

Aloitetaan tutustumalla *jakamattomiin (atomisiin)* tietotyyppeihin, joiden sisärakenteeseen ohjelmoija ei 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 myöhemmin) on epätosi.

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

II.4.2 Numerot

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

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

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

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

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

Konventio: Konversiorutiini tyypistä t tyyppiin 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.5 Kielen ydin

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

Tämä tehdään määrittelemällä *induktiivisesti*

lausekkeet eli Scheme-ohjelmien kirjoitusasu,

arvot joita nämä lausekkeet esittävät ja

laskentasäännöt joilla lausekkeen esittämä arvo selviää.

Kalvojen II.4 vakiot c ovat lausekkeita, ja niiden arvoina ovat vastaavat totuusarvot/luvut.

Tämän kertoo sääntö $\mathbf{E}c \Rightarrow c$ eli "vakio *evaluoituu* omaksi itsekseen" (vertaa kalvot I.3).

II.5.1 Symbolit

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

Symbolin kirjoitusasu voi koostua kirjaimista, numeroista ja erikoismerkeistä `! $ % & * + - . / : < = > ? @ ^ _ ~` kunhan sen alkua ei voi erehtyä luulemaan numerovakioksi.

- Vielä nyt käytämme symboleita vain *nimeämään* arvoja kuten funktioita ja niiden argumentteja.

  Sallimme myöhemmin symbolit myös tietorakenteissa.

- Emme vielä salli symbolille sitä esiteltäessä annetun alkuarvon muuttamista jälikäteen.

  Symbolit ovat siis *matemaattisia* eivätkä ohjelmointikielten muuttujia. Toisin sanoen, *vakioita*.

  Kun tämä myöhemmin sallitaan, astutaan ulos funktionaalisesta ohjelmoinnista koska ohjelmalla on siitä lähtien *tila*: suorittavan koneen muistin sisältö.

II.5.2 Muuttujanmäärittelyt

Muuttuja määritellään ja sille annetaan arvo seuraavasti:

(**define** *nimi lauseke*)

esittelee symbolin *nimi* ja asettaa sen viittaamaan arvoon joka saadaan laskemalla $\mathbf{E}$*lauseke*, eli määrittelevän lausekkeen arvoon.

Muuttujaviittaukset tarkoittavat tietenkin tätä arvoa, joten $\mathbf{E}$*nimi* on se arvo johon symboli *nimi* on asetettu viittaamaan sen **define**-määrittelyssä. Siten

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

määrittelee muuttujalle *h* arvon $\frac{1}{1000}$.

Vastaavanlaisilla määritelmillä esitellään myös funktiot kalvolla II.5.3. Scheme-ohjelma koostuu jonosta tälläisiä määritelmiä, ja ohjelman suoritus on annetun lausekkeen arvon laskemista näin tehdyillä määritelmillä.

Common Lisp: Muuttujanmäärittely onkin muotoa

(**defvar** *nimi lauseke*)

II.5.3 Funktionmäärittelyt

Funktionmäärittely on muotoa

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

joka esittelee symbolin *nimi* ja antaa sen arvoksi erään k-argumenttisen funktion, joka on intuitiivisesti "*lausekkeen* arvo kun kunkin (symbolilla nimetyn) *parametrin* i tilalla on kutsun vastaava *argumentti* i".

Rekursio on sallittua: funktion *nimi* määrittelevä *lauseke* saa kutsua itseään sekä muita ohjelman funktioita.

Näin määriteltu funktio selostetaan sen kutsujen yhteydessä kalvolla II.5.4.

Vakiokirjasto tarjoaa valmiita funktioita.

Common Lisp: Funktionmäärittely onkin muotoa

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

II.5.4 Funktionkutsut

Ilmaus

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

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

Formaalimmin $\mathbf{E}$(*nimi lauseke*$_1 \ldots$ *lauseke*$_k$) lasketaan seuraavasti:

- Lasketaan kukin $\mathbf{E}$*lauseke*$_i \Rightarrow$ *argumentti*$_i$.
- Haetaan kalvon II.5.3 mukainen määrittely

  `(define (`*nimi parametri*$_1 \ldots$ *parametri*$_k$`)` *lauseke*`)`.
- Tehdään *lausekkeesta* sellainen *kopio* jossa kukin symbolin *parametri*$_i$ esiintymä on korvattu vastaavalla arvolla *argumentti*$_i$.
- Jatketaan laskemalla $\mathbf{E}$*kopio*.

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

II.5.5 Ehtolauseke

Kalvon II.5.4 funktionkutsuperiaate ei sovellu lausekkeisiin muotoa

(`if` $lauseke_{ehto}$ $lauseke_{totta}$ $lauseke_{epätotta}$)

joiden intuitio on "jos $lauseke_{ehto}$ on tosi niin sitten arvo on $lauseke_{totta}$ mutta muuten arvo onkin $lauseke_{epätotta}$", koska kutsussa laskettaisiin ensin jokainen $lauseke_{ehto/totta/epätotta}$.

Niinpä $\mathbf{E}$(`if` $lauseke_{ehto}$ $lauseke_{totta}$ $lauseke_{epätotta}$) lasketaankin seuraavasti:

1. Ensin lasketaan $\mathbf{E}lauseke_{ehto}$.

2. Jos tulos on `#f` (kalvolta II.4.1), niin jatketaan laskemalla $\mathbf{E}lauseke_{epätotta}$.

3. Muuten jatketaankin laskemalla $\mathbf{E}lauseke_{totta}$.

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

II.6 Korkeampaa ohjelmointia

Tarkennetaan edellä 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.

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.

Toisaalta derivointisäännöt ovat vain kertalukua 1, koska ne kuvaavat kertalukua 0 olevia lausekkeita toisikseen.

Scheme-kielen sanotaan olevan "kertalukua ω" koska 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 *Newtonin 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 päätyy derivaatan nollakohtaan ja jumittuu).

Numeriikassa kalvojen I.3 oikean derivaattafunktion tilalle otetaan

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

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

Voimme ottaa sen vaikkapa kalvolta II.5.2.

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

Haluamme siis välittää funktioon `newton` parametrinä toisen funktion `f2`. Tämä käy suoraan, koska kalvon II.5.4 funktiokutsussa

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

voi kutsuttavan funktion *nimi* olla tämän nykyisen funktion jokin *parametri* ja silloin kutsutaan tuota parametrinä saatua funktiota argumenteilla $\mathit{lauseke}_i$.

```
(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))))))
```

Nyt olemme nostaneet funktion `newton` toiseen kertalukuun.

Kalvoilla II.5 esitettyä ”funktiokutsu korvaamisella” -periaatetta voisi laajentaa korkeampaan kertalukuun yksinkertaisuuden kustannuksella [CM98,§3.2.1].

Emme kuitenkaan tee sitä, vaan palaamme korkeamman kertaluvun funktioiden toimintaan myöhemmin esitellessämme kokonaan uuden ja muutenkin täydellisemmän Scheme-ohjelmien suoritusmallin.

II.6.2 Paikalliset määritelmät

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

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 kirjoittaa toisia `define`-funktionmäärittelyjä ennen määrittelylauseketta.

(Myös `define`-muuttujanmäärittelyjä voisi käyttää 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 kuin ympäröivän määrittelyn lausekkeeseenkin, ellei 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 yksityiseen parametriin, ei funktion `newtona` samannimiseen parametriin joka siis peittyi.

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

Scheme noudattaa siis muista lohkokielistä tuttuja *leksikaalisia näkyvyyssääntöjä* (lexical scope). (Ennen 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 palauttaa 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 (ei funktionsa `f2` nimeä tms. vaan) *tiedon siitä mikä funktio* `f2` *on nykyisessä määrittely-ympäristössään* johon siis kuuluu se parametrin `a` arvo, jonka `g` tällä kertaa sai.

Funktion `g` sisäiset määritelmät tehdään siis 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".

Olio-ohjelmoinnissa saman voisi toteuttaa seuraavasti:

- Määritellään luokka `G` joka sisältää kentän `A`.
- Luokan konstruktori `G(b)` alustaa tämän kentän argumentillaan `b`.
- Luokan ainut metodi `E(x)` palauttaa tuloksenaan `x*x-a`.
- Nyt funktionkutsua `(g 2)` vastaa konstruktorikutsu `G(2)` joka synnyttää tuloksena saatua funktiota f vastaavan olion o.
- Tulosfunktion f kutsua vastaa olion o metodin `E` kutsu samalla argumentilla.

Kun lisäämme myöhemmin muuttujien arvoja muuttavat (ei-funktionaaliset) operaatiot, saamme Scheme-kielen jolla voi tehdä myös olio-ohjelmointia.

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 antaa sille *nimen*.

Siis (`define` *nimi lauseke*) onkin pelkkä *lausekkeen* arvon *nimeämisoperaatio*, olipa tämä arvo sitten minkä tyyppinen vakio tahansa: totuusarvo, numero, *funktio*, ...

Vastaavasti funktionkutsussa ($lauseke_0 \ldots lauseke_k$) sallitaan nyt ennen kutsuttavan funktion nimelle varatussa ensimmäisessä paikassa mielivaltainen lauseke $lauseke_0$, jolle lasketaan arvo samoilla säännöillä kuin muillekin lausekkeille. Ei ole väliä saatiinko arvo lopulta

- paikallisen tai julkisen `define`-määrittelyn lausekkeesta (normaalisti nimetty funktio)
- tätä kutsulauseketta ympäröivän `lambda`-lausekkeen parametrilistasta (parametrina saatu funktio) vai
- laskemalla arvo jollekin $lausekkeen_0$ sisältämälle `lambda`-lausekkeelle (juuri luotu nimetön funktio)

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

Common Lisp: Nimipaikka käsitellään toisin säännöin kuin muut.

Eräs yllättävä mutta yleinen nimettömän funktion käyttötapa on

((`lambda` (*var*$_1$ ... *var*$_k$) *lauseke*) *val*$_1$ ... *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*$_i$.

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*".

Täysi Scheme-syntaksi sisältää monia muitakin *johdettuja lausekkeita* jotka palautuvat esiteltyihin perusrakenteisiin.

Operaatio "ota annetusta luvusta p:s juuri" on nyt

```
(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 funktion $y \mapsto y^p - x$ nollakohdalle (ja tekee sen Newtonin menetelmällä alkuarvauksenaan x itse)".

Samalla saatiin muillekin funktioille hyödyllinen operaatio `newton`.

II.6.5 Lambdakalkyylistä

Loogikko Alonzo *Church* julkaisi vuonna 1936 λ-*kalkyylin* (calculus) seuraavien asioiden formalisointiin:

- Funktiot sääntöinä (eli "koululaisten" eikä joukko-opillisina).

- Mekaaninen laskeminen lausekkeita sieventämällä (eli $(2+3)/5$ on $5/5$ on 1).

  Samana vuonna (hieman aiemmin) Alan Turing oli julkaissut oman *laite*mallinsa mekaanisen laskettavuuden pohjaksi.

  Church-Turingin teesi: näin saadut (ekvivalentit) systeemit ovat oikea formaali vastine intuitiiviselle mekaanisen laskettavuuden käsitteelle.

- Leibnizin operaatio "korvaa tämä ilmaus tuolla ilmaukselle siinä yhteydessä".

Lisp-kielten suunnittelijat ottivat nämä ideat omiin tarkoituksiinsa. Myöhemmin niitä on alallamme hyödynnetty erityisesti ohjelmointikielten teoriassa.

Peruskäsitteet:

- ”Riittävä” määrä *termimuuttujia* $x_0, x_1, x_2, \ldots$

  Termimuuttujan esiintymä yksinään on yksinkertaisin *termi*, ja *vapaa* esiintymä.

- Jos t_1 ja t_2 ovat termejä, niin myös $(t_1\ t_2)$ on termi.

  Intuitiivisesti siinä *sovelletaan* funktiota t_1 argumenttiin t_2.

  Vapaat esiintymät ovat samat kuin termeissä t_1 ja t_2.

- Jos t on termi ja x termimuuttuja, niin $\lambda x.t$ on termi.

  Intuitiivisesti siinä *abstrahoidaan* termistä t funktio jonka parametriposition nimi on x.

  Vapaat esiintymät ovat kuten termissä t paitsi että kaikki termimuuttujan x vapaat esiintymät tulevat nyt *sidotuiksi* koska ne esittävät kohtia johon parametria vastaava argumentti kutsussa tulee.

Termien sievennykseen on β-*reduktio:* lauseke $((\lambda x.t_1)\ t_2)$ voidaan korvata lausekkeella $t_1[x \leftarrow t_2]$ eli "korvaa kaikki termimuuttujan x vapaat esiintymät termissä t_1 termillä t_2".

- Termimuuttujille

$$x_j[x_i \leftarrow t] = \begin{cases} t & \text{kun } i = j \\ x_j & \text{kun } i \neq j. \end{cases}$$

- Sovelluksille
  $(t_2\ t_3)[x \leftarrow t_1] = (t_2[x \leftarrow t_1]\ t_3[x \leftarrow t_1])$.

- Abstraktioille

$$(\lambda x_j.t_2)[x_i \leftarrow t_1] = \begin{cases} \lambda x_j.t_2 & \text{kun } i = j \\ \lambda x_j.(t_2[x_i \leftarrow t_1]) & \text{kun } i \neq j. \end{cases}$$

Ekstensionaalisuusperiaate: termit t_1 ja t_2 kuvaavat samaa funktiota jos $(t_1\ t_3)$ ja $(t_2\ t_3)$ β-redusoituvat yhteiseen muotoon kaikilla t_3.

II.7 Rekursio

Useimmissa ohjelmointikielissä esitellään *iteraatio- eli toistorakenteet* (`for`, `while`, `repeat until`, `...`) ja mainitaan että aliohjelmat voivat kutsua itseään *rekursiivisesti*.

Scheme-kielessä ei ole lainkaan toistorakenteita, vaan ainoa tapa tehdä toistoa on juuri rekursiivisilla aliohjelmilla [AS96,§1.3].

Tai pikemminkin: Scheme-kielessä on vain aliohjelmakutsu, eikä rekursiivisten ja muiden kutsujen välillä ole eroa.

Iterare toistaa.

Recurrere palata.

Toistaminen on *alkuun* palaamista.

Rekursiolla on helppo ilmaista *induktiivisesti* määriteltyjä operaatioita. Esimerkiksi kertomafunktion määritelmä

$$n! = \begin{cases} 1 & \text{kun } n = 0 \\ n \cdot (n-1)! & \text{muuten} \end{cases}$$

taipuu Scheme-kielelle lähes sellaisenaan:

```
(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1)))))
```

Laskettaessa kutsulle `(fac` m`)` arvoa tehdään lauseke `(*` m `(*` $m-1$ `(*` $m-2$ `( ... 1 ... ))))` jonka arvo lasketaan lopuksi.

Rekursiivinen `if`-haara sanoo: ”Pitää kertoa nykyinen `n` luvulla joka saadaan kutsusta `(fac (- n 1))`. Minäpä suoritan nyt tuon kutsun ja *palaan sen arvon kanssa myöhemmin takaisin* tekemään kesken jäävän kertolaskun loppuun”.

Scheme-toteutus joutuu siis pitämään kirjaa keskeneräisistä töistään; tätä kirjanpitoa voi ajatella jäljellä olevana lausekkeena. Suurilla m tämä kirjanpito ei enää mahdu muistiin; silloin joudutaan ohjelmaa muuttamaan kirjanpidon pienentämiseksi.

Rekursiivinen algoritmi voi esittää iteratiivista laskentaprosessia:

```
(define (mystery x)
  (if (= x 1)
      0
      (mystery (if (even? x)
                   (/ x 2)
                   (+ (* 3 x) 1)))))
```

esittää iteratiivisen algoritmin

while $x \neq 1$ **do**
  if x on parillinen **then**
    $x \leftarrow x/2$
  else
    $x \leftarrow 3 \cdot x + 1$
  end if
end while

(joka muuten näyttää pysähtyvän kaikilla alkuarvoilla x mutta tätä ei ole vielä kyetty todistamaan [H92,s.204]).

Siinä kirjanpitoa *ei tarvita* koska rekursiivisesta kutsusta saatu arvo on itsessään valmis vastaus ilman lisäkäsittelyä.

Tämä on iteratiivisen algoritmin idea: vastaavan prosessin tila voidaan aina lukea sen näkyvistä muuttujanarvoista (ja ohjelmakohdasta).

II.7.1 Loppukutsut

Iteratiiviset rekursiokutsut ovat siis niitä jotka eivät jätä muistijälkiä. Scheme-standardi vaatii (useimmista muista ohjelmointikielistä poiketen) että toteutus tekee sellaiset kutsut tilaa säästäen.

Ohjelmoijan on siis syytä tunnistaa sellaiset kutsut Scheme-ohjelmiensa (ajan- ja) tilantarpeen pienentämiseksi [KCR98,§3.5].

Ensiksi tunnistetaan annetun lambda-lausekkeen *loppuyhteydessä (tail context) olevat alilausekkeet*, intuitiivisesti ne jotka "tehdään lopuksi juuri ennen lambdasta poistumista".

Tutkittavan lambda-lausekkeen

$$\texttt{(lambda (}x_1 \ldots x_k\texttt{)}\ e_1 \ldots e_m\texttt{)}$$

viimeinen alilauseke e_m on loppuyhteydessä; ei-funktionaalisissa Scheme-ohjelmissa sitä edeltävät lausekkeet $e_1, \ldots, e_{m-1}$ suoritetaan *sivuvaikutustensa* (syöte/tulostus, uudelleensijoitus, ...) vuoksi, e_m taas arvon saamiseksi koko lambda-lausekkeelle.

Jos ehtolauseke (`if` $e_?$ $e_{\#t}$ $e_{\#f}$) on loppuyhteydessä, niin myös sen haarat $e_{\#t}$ ja $e_{\#f}$ ovat, koska jompi kumpi niistä tehdään ehtolauseen lopuksi. Ehto $e_?$ *ei* ole loppuyhteydessä, koska sen jälkeen tehdään vielä jompi kumpi haaroista.

Jos lambda-lauseketta näin tutkittaessa "ulkoa sisäänpäin" päädytään loppuyhteydessä olevaan alilausekkeeseen

(`lambda` ($var_1 \ldots var_k$) $laus_1 \ldots laus_m$) $val_1 \ldots val_k$)

niin

1. selvitetään ne (lausekkeen $laus_m$ ali)lausekkeet jotka ovat loppuyhteydessä tähän sisempään lambdaan ja

2. otetaan ne myös ulomman lambdan loppuyhteyteen

koska tämän alilausekkeen viimeinen teko on laskea (oman ali)lausekkeensa $laus_m$ viimeinen teko.

Siis kalvojen II.6.4 `let`-rakenteessa

(`let` ((var_1 val_1)$\ldots$(var_k val_k)) $laus_1 \ldots laus_m$)

loppuyhteydessä on vain $laus_m$.

Näillä säännöillä voi (johdettuja lausekkeita purkaen) selvittää loppuyhteydessä olevat lausekkeet.

Esimerkiksi kalvojen II.7 funktiossa `mystery` loppuyhteydessä ovat vain vakio `0` ja rekursiokutsu.

Loppuyhteydessä olevaa funktionkutsua nimitetään loppukutsuksi (tail call) ja sen Scheme-järjestelmä suorittaa ”in situ”, ”paikallaan”.

(Jos kutsu ei ole loppuyhteydessä, se varaa tietenkin tarvitsemansa kirjanpitomuistin.)

Rekursiivinen loppukutsu(jen jono) on Scheme-tapa esittää iteraatio: esimerkiksi funktio `mystery` ja kalvojen II.6 funktio `newton` toimivat itse asiassa **while**-silmukoina sellaisinaan ilman että ohjelmoijan tarvitsee käyttää erillistä rakennetta.

Tälläistä ”epäaitoa” loppurekursiota kutsutaan usein *häntä- tai takarekursioksi* (tail recursion) ja Scheme-kielen standardin sanotaan vaativan sen *poistoa* (elimination) jota pidetään tavallisesti (ohjelmoijien tai) ohjelmointikieli*toteutusten optimointi(palvelu)na*.

Kun imperatiivinen ohjelmoija haluaa kirjoittaa silmukan, hän kirjoittaa `while`-avainsanan tms. eikä hänen tarvitse nimetä silmukkaansa.

Kun Scheme-ohjelmoija haluaa kirjoittaa silmukan, hän siis esittää sen loppurekursiivisena funktiona, joka täytyy nimetä.

Nimetön lambda-funktio ei nimittäin voi olla rekursiivinen (ilman temppuilua) – millä nimellä se kutsuisi itseään?

Scheme-stadardissa on kalvojen II.6.4 tapaan johdettuna lausekkeena *nimetty* `let` muotoa

(`let` *nimi* ((*var*$_1$ *val*$_1$)...(*var*$_k$ *val*$_k$)) *laus*$_1$ ... *laus*$_m$)

jossa lisäksi rungon lausekkeisiin *laus*$_i$ – muttei muualle – näkyy k-parametrinen funktio *nimi*.

Tämän paikallisen funktion kutsu (*nimi* *val*$'_1$ ... *val*$'_k$) tekee rungon uudelleen arvoilla *var*$_i$ = *val*$'_i$.

Loppukutsuna se siis vastaa silmukan seuraavaa kierrosta – tai tavallisena rekursiokutsuna muissa yhteyksissä.

Esimerkiksi kalvojen II.6.4 iteratiivinen aliohjelma `newton-aux` voidaan korvata nimetyllä `let`-lausekkeella seuraavasti:

```
(define h 1/1000)

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

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

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

Erityisesti alustuksessa `(x x)` silmukan sisäinen (vasemmanpuoleinen) `x` saa alkuarvokseen parametrina saadun (oikeanpuoleisen) arvon `x`.

II.7.2 Rekursiosta iteraatioksi

Yleistä rekursiota käyttävä ohjelma(nosa) voidaan usein muuntaa iteratiiviseksi käyttämällä *kerääjiä* (accumulators).

Kerääjä on lisäparametri johon kootaan iteraation aikana niin paljon lisätietoa että sen perusteella voidaan lopuksi laskea alkuperäinen vastaus.

Esimerkiksi kalvojen II.7 kertomafunktio `fac` ei ollut iteratiivinen, koska kutsun (`fac` $m-p$) vastausta piti myöhemmin täydentää kertomalla se tulolla $m \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot (m-p+1)$ lopullisen vastauksen $m!$ saamiseksi.

Kootaan siis tämä täydennysinformaatio kerääjäparametriin `a` ja pidetään näin huolta siitä että rekursiokutsu siirtyy loppuyhteyteen.

Kehitetään vastaava Scheme-funktio `fac2` vaihe vaiheelta *tehtävän spesifikaation ohjaamana ohjelmointityönä*.

- Ensiksi `(define (fac2 n) (fac-acc n` α`))` missä `fac-acc` on iteratiivinen apufunktio jonka kerääjäparametrille `a` antaa alkuarvon vielä tuntematon Scheme-lauseke α.

- **Idea:** kehitetäänpä sellainen `fac-acc` että `(fac-acc m a)` $= m! \cdot a$.

  Idean jälkeen loppu on melko mekaanista.

  Esimerkiksi koska halutaan
  $n!$ = `(fac2 n)` = `(fac-acc n` α`)` $= n! \cdot \alpha$ on valittava $\alpha =$ `1`.

- Aloitetaan kehitys `(define (fac-acc m a)` β`)` missä vaaditaan siis $\beta = m! \cdot a$.

- Edetään *induktiolla parametrin* `m` *suhteen*.

  Koska $m \in \mathbb{N}$ jaetaan konstruktio
  β = `(if (zero? m)` γ δ`)` missä γ on perustapaus $m = 0$ ja δ induktiivinen tapaus $m > 0$.

  Induktiossa on muistettava palata vain tapaukseen $m - 1$ jotta pysähtyminen perustapaukseen olisi taattu.

- Nyt mekaanisesti $\gamma = m \cdot a = 0! \cdot a =$ `a`.

- Samoin
  $\delta = m! \cdot a =$ `(fac-acc (- m 1) η)` $\underset{\text{ind.ol.}}{=} (m-1)! \cdot \eta$
  kunhan valitaan $\eta = m \cdot a =$ `(* m a)`.

Kun kaikki lausekkeet $\alpha, \ldots, \eta$ yhdistetään paikoilleen saadaan lopulta

```
(define (fac2 n)
  (fac-acc n 1))

(define (fac-acc m a)
  (if (zero? m)
      a
      (fac-acc (- m 1)
               (* m a))))
```

joka on

- iteratiivinen kuten pitikin ja

- korrekti jo *konstruktionsa nojalla*.

Helpompaa kuin keksiä ensin algoritmi ja todistaa se(n idea) sitten oikeaksi!

Funktionaalisessa ohjelmointitavassa ohjelmalausekkeiden ja arvojen yhtäsuuruus on helpompi nähdä kuin esim. imperatiivisessa eri käskyjonojen sama merkitys.

II.7.3 Korkeampaa iteroituvuutta

Koska kielessämme on kalvojen II.6 korkeamman kertaluvun funktiokäsite, voidaan kalvojen II.7.2 kerääjätekniikkaa yleistää keräämällä *funktio joka kertoo mitä nykyisen kutsun tuloksella pitikään tehdä*.

Otetaan esimerkiksi (yleistetyt) murtojatkeet (continued fractions)

$$\cfrac{1}{f(n)+\cfrac{1}{f(n-1)+\cfrac{1}{f(n-2)+\cdots\cfrac{1}{f(1)}}}}$$

jotka taipuvat rekursiiviseen Scheme-muotoon

```
(define (cf f n)
  (define (cf-aux m)
    (if (zero? m)
        0
        (/ (+ (f m)
              (cf-aux (- m 1))))))
  (cf-aux n))
```

Muokataan siitä iteratiivinen versio lisäämällä apufunktioon `cf-aux` kerääjä*funktio* `a` invarianttinaan "kun parametrinä annetaan *nykyisen* rekursiokutsun antama tulos, saadaan vastauksena *koko* rekursion tulos".

Siis `a` kerää nyt tietoa siitä miten laskentaa pitää *jatkaa* nykyisen rekursiokutsun jälkeen.

Aluksi `(define (cf2 f n) (cf-acc n` α`))` missä

$$\texttt{(cf f n)} = \texttt{(cf2 f n)} = \texttt{(cf-acc n } \alpha\texttt{)}$$

pätee kun kerääjäfunktion `a` alkuarvoksi valitaan identtinen kuvaus $\alpha =$ `(lambda (v) v)`: ensimmäisen rekursiokutsun tulosta `v` ei tarvitse (eikä pidä) jatkokäsitellä, vaan se annetaan sellaisenaan vastauksena.

Jatkamalla kalvojen II.7.2 tapaan saadaan

`(define (cf-acc m a) (if (zero? m)` γ δ`))`

missä γ edustaa ei-rekursiivista haaraa `0`. Tälläisessä haarassa tämä rekursiokutsu on antamassa tulostaan, ja `a` tietää miten se pitää jälkikäsitellä, joten $\gamma =$ `(a 0)`.

Lauseke δ on puolestaan rekursiivinen haara. Koska apufunktiosta `cf-acc` halutaan iteratiivinen, on

$$\delta = \texttt{(cf-acc (- m 1)}\ \eta\texttt{)} \text{ missä} \qquad (\dagger)$$
$$\eta = \texttt{(lambda (v) (a (/ (+ (f m) v))))}$$

on kerääjäfunktion `a` päivitetty muoto seuraavaa iteraatiokierrosta varten:

- Parametrinä `v` saadaan (aikanaan) tämän rekursiokutsun ($\dagger$) vastaus $v = \texttt{(cf-aux (- m 1))}$.

- Tätä vastausta v jatkokäsitellään ensin paikallisesti lausekkeella
$$w = \frac{1}{f(m) + v} = \texttt{(cf-aux m)}.$$

- Paikallisen jatkokäsittelyn tulos w annetaan lopuksi parametrina saadulle kerääjäfunktiolle `a` ei-paikallista loppukäsittelyä varten.

Jos rekursiohaarassa δ on useita rekursiokutsuja jotka on tehtävä iteratiivisiksi, sovelletaan samaa periaatetta uudelleen lausekkeen η sisällä: `(lambda (v) (a (`acc-kutsu `...` η'`)))` jne.

Lopputulos on yhteen koottuna

```
(define (cf2 f n)
  (define (cf-acc m a)
    (if (zero? m)
        (a 0)
        (cf-acc (- m 1)
                (lambda (v)
                  (a (/ (+ (f m)
                           v)))))))
  (cf-acc n (lambda (v) v)))
```

jossa esimerkiksi kutsu `(cf2 g 2)` päättyy lausekkeen

```
((lambda (v1)
   ((lambda (v2)
      ((lambda (v)
         v)
       (/ (+ (g 2) v2))))
    (/ (+ (g 1) v1))))
 0)
```

arvon laskentaan. Tämä taas on kalvojen II.6.5 β-reduktioiden jälkeen sama lauseke

```
(/ (+ (g 2) (/ (+ (g 1) 0))))
```

jonka vastaava alkuperäinen kutsu `(cf g 2)` tuottaa rekursiokirjanpitonaan, kuten pitikin.

II.7.4 Jatkeet

Kalvoilla II.7.3 tehtiin rekursiivisesta prosessista iteratiivinen keräämällä iteraation kuluessa funktiota joka kertoi *kuinka laskentaa on jatkettava* iteraation jälkeen.

Tälläisiä funktioita kutsutaan *jatkeiksi* (continuations) ja ohjelmointityyliä jossa ohjelmoija hallinnoi jatkeita itse – järjestelmän sijasta – *jatkeidenvälitystyyliksi (Continuation Passing Style, CPS)* [D96,§3.3-3.4].

\+ Jatkeita käsittelemällä voi toteuttaa *minkä tahansa sarjallisen kontrollirakenteen*.

– Epästrukturoitu (eli vaikea) kontrollivuo joka jää ohjelmoijan vastuulle.

\+ Korkeamman kertaluvun kieli ei siis tarvitse erillistä rekursiomekanismia.

CPS onkin suosittu funktionaalisten kielten *toteutus*tason periaatteena.

– Rekursion tilavaativuus ei katoa vaan siirtyy järjestelmältä parametrinvälitykseen.

Tyylin tunteminen pelastaa kuitenkin usein pulasta. Esimerkkinä olkoon *virheidenkäsittely rekursiossa*.

Vaaditaan kalvojen II.7.3 murtojatkefunktiolta ettei nollalla jakoon saa enää "kaatua" vaan silloin *koko rekursion* on palautettava `#f`.

Jos päivitämme alkuperäistä rekursiivista funktiota `cf` niin joudumme tarkastamaan apufunktiossa `cf-aux` ...

- rekursiokutsun jälkeen raportoiko se nollalla jaon, ja jos raportoi, niin lähetettävä raportti edelleen; ja

- onnistuneenkin rekursiokutsun jälkeen onko jakajaksi nyt tulossa 0, eli pitääkö nyt lähettää ensimmäinen virheraportti.

Tähän ongelmaan tarjoavat monet ohjelmointikielet erilaisia *poikkeustenkäsittelymekanismeja* (exception handling): `throw-catch` / Common Lisp [S98,§9.10], `try-catch` / Java sekä C++ [S97,§14.1], `try-with` / ML-kielet [CM98,§4.1], ...

Scheme-kielessä ei ole tähän valmista mekanismia.

Mutta jos ongelmallinen rekursio kirjoitetaankin jatkeidenvälitystyylillä, tulee mahdolliseksi poistua rekursiosta ”kesken kaiken”: *Parametrina saatua ”tee lopuksi tämä” -jatketta ei ole pakko kutsua!*

Sen sijaan voidaan ryhtyä käsittelemään virhettä, ja koska kyseessä onkin iteraatio, virheenkäsittelijä voi palata suoraan rekursion aloittajaan:

```
(define (cf3 f n)
  (define (cf-exit m a)
    (if (zero? m)
        (a 0)
        (cf-exit (- m 1)
                 (lambda (v)
                   (let ((denom (+ (f m) v)))
                     (if (zero? denom)
                         #f
                         (a (/ denom)))))))))
  (cf-exit n (lambda (v) v)))
```

Itse asiassa muiden kielten keskeytysmekanismien voi ajatella luovan *kaksi* jatketta sisältämälleen `try`-ohjelmanosalle: sen tavallisen (näkymättömän) perusrekursiojatkeen ja `catch`-virhejatkeen.

II.7.5 Järjestelmän jatkeet

Kalvoilla II.7.4 nähtiin miten Schemessä voi ilmaista pakeneminen kesken rekursion ohjelmoijan ylläpitämillä jatkeilla.

Scheme-kielessä myös *järjestelmän* ylläpitämä tämänhetkinen suorituksessa oleva jatke on ohjelmoijan ulottuvilla: kirjastofunktiokutsulla

`(call-with-current-continuation (lambda (`c`) `e`))`

lausekkeessa e muuttuja c saa arvokseen yksiparametrisen *pakofunktion*. [KCR98,§6.4;D96,§5.6]. Usein käytetty synonyymi on `call/cc`.

Jos tehdään pakofunktion kutsu (c a) niin suoritus siirtyy *välittömästi ja palaamatta* jatkamaan vastaavasta `call/cc`-kutsusta ja antaa sille arvon a.

Toisin sanoen, `call/cc` paketoi normaalisti näkymättömän järjestelmäjatkeen funktioksi, jota kutsumalla nykyinen järjestelmäjatke vaihtuu paketoituun.

Esimerkiksi kalvojen II.7.4 ongelman olisi voinut ratkaista järjestelmäjatkeilla suoraan poistamatta ensin rekursiota:

```
(define (cf4 f n)
  (call/cc (lambda (exit)
             (define (cf-cc m)
               (if (zero? m)
                   0
                   (let ((denom
                          (+ (f m)
                             (cf-cc (- m 1)))))
                     (if (zero? denom)
                         (exit #f)
                         (/ denom)))))
             (cf-cc n))))
```

Kääntäen, `call/cc`-kutsuja käyttävä ohjelma voidaan aina kirjoittaa jatkeenvälitystyylillä ilman näitä kutsuja, mutta silloin saatetaan joutua kirjoittamaan tyylin mukaiset versiot kirjastofunktioistakin.

Koska paketti c on tavallinen Scheme-funktio, voi suoritus "pomppia" lausekkeesta ulos ja takaisin sisään. Kirjastofunktiokutsulla

(`dynamic-wind` sisään lauseke ulos)

voi kertoa että (esim. alustukset) "sisään" on tehtävä joka kerran kun lausekkeeseen pompataan, ja "ulos" vastaavasti.

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, Schemessä 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 Schemeä 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 jakamaton vakio, mutta siitä aloitetaan listojen induktiivinen määrittely.

Schemessä tyhjää listaa merkitään `'()`.

Lainausmerkin `''` tarkoitukseen palataan myöhemmin. (DrScheme-toteutuksen kielitaso "full scheme" näyttää kuitenkin sallivan sen jättämisen pois.)

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

Common Lisp: Tyhjällä listalla on myös nimi `nil`.

> 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.

Esimerkiksi lauseke

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

tuottaa "sisältä ulos" luettuna

1. yksialkioisen listan yhdistämällä tyhjän listan alkuun alkion `4`

2. kaksialkioisen listan alkioinaan `3` ja `4` tässä järjestyksessä yhdistämällä askeleessa `1` tehdyn listan alkuun alkion `3`

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

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

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

- Jokainen jakamaton tietoalkio on omassa laatikossaan.

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

- Kutsu asettaa edelliseen/vasemmanpuoleiseen lokeroon nuolen, joka osoittaa uuteen lisättyyn alkioon a.

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

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

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

Esimerkiksi lauseke (‡) luo ensin parin jonka vasen jäsen osoittaa vakioon 4 ja oikea on tyhjä, sitten 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 `null?` tutkii onko sen argumentti aito lista.

Aidon n-alkioisen listan *tulostusasu* on

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

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

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

eli päättävän parin piste tulee näkyviin.

II.8.3 Parin osat

Kalvoilla II.8.2 synnytetyn parin sisältöä voi kurkistaa kirjastofunktiolla

`car` edelliselle lokerolle (joka osoitti ensimmäiseen alkioon)

`cdr` jälkimmäiselle lokerolle (joka osoitti loppulistaan).

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

Koska tälläisiä listankaivuuoperaatioita käytetään 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 $1 \leq k \leq 4$.

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 `()`$\mapsto$`()`.

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` testatessa on työlästä kirjoittaa argumentteja kuten `(1 2 3 4)` 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 onkin vakio tulostusasussaan.

Kalvojen II.4 numero- ja totuusarvovakioihin ei vielä tarvittu lainausmerkkiä (mutta se sallitaan).

Listavakiot saattavat kuitenkin sekoittua suoritettaviin lausekkeisiin yhteisen syntaksin vuoksi: ensimmäistä alkiota saattaisi luulla kutsuttavan funktion nimeksi ja muita sen argumenteiksi.

(Vaara vain kasvaa sitten kun tuomme symbolitkin listoihin.)

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.

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 ilmaukselle `(unquote` e`)`) *pakenee* lainausmerkkien sisältä lausekkeen e ajaksi, eli laskee sen arvon ja liittää sen synnytettävän listan (tms.) alkioksi tähän kohtaan.

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 ensimmäisenä alkiona on `1`, toisena muuttujan `x` arvo ja kolmas, neljäs, viides, ... alkio saadaan muuttujan `y` arvona olevasta listasta.

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 kun lausekkeiden a ja b yhteisenä arvona on *sama elementti* kalvojen II.8.2 (muistinkäyttöä kuvaavassa) laatikko-nuoli-piirroksessa.

Se siis tutkii onko kyseessä *yksi ja sama rakenne*.

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

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

- Vakiolla `#t` (ja `#f`) on yksikäsitteinen oma järjestelmän varaama laatikko.

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

- Operaatio `cons` synnyttää aina uuden laatikon, joten `eq?` kertoo pareista ovatko ne syntyneet samalla kertaa.

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

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

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

Yleensä `equal?` lienee luonnollisin vaihtoehto.

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

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

joten miten niitä verrataan toisiinsa?

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

Matematiikassa funktiot ovat samat jos ne saavat 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 samana (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 bar)` on määrittelemätön (yleensä `#f`). Samoin `eqv?` ja `equal?`.

II.8.6 Symbolit datassa

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

Ulkopuolella oleva lausekkeen osa suoritetaan koko lausekkeen lausekkeen arvon laskemiseksi.

Tällä puolella kalvon II.5.1 *symbolit* käyttäytyvät kuten *ohjelmointikielten muuttujat*.

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

Symboleita voi esiintyä myös tällä puolen: silloin ne ovat oman jakamattoman *tietotyyppinsä (`symbol?`) vakioita*.

On harmillista että (Lisp-historiallisista syistä) datassa olevilla

- symboleilla on sama syntaksi kuin muuttujilla
- listoilla on sama syntaksi kuin lausekkeilla

vaikka ne ovat eri käsitteitä.

Esimerkiksi määritelmän `(define foo 1)` jälkeen *muuttuja(symboli)n* `foo` *arvo*ksi on asetettu `1`.

Lausekkeessa `(cons foo '(foo bar))` edellinen `foo` esiintyy heittomerkin **ulko**puolella, jälkimmäinen taas **sisä**puolella samassa listassa toisen symboli(sen vakio)n `bar` kanssa.

Lausekkeen arvoksi saadaan lista `(1 foo bar)` koska edellinen esiintymä korvautuu arvollaan, jälkimmäinen taas on listavakion alkio ja pysyy itsenään.

Datassa (heittomerkin sisällä)

- symboli edustaa omaa itseään (eikä mahdollista arvoaan)

- symbolin tulostusasu on se itse.

- saa esiintyä myös (`define-` tai `lambda-`)määrittelemättömiä symboleita kuten `bar`.

  Tälläiseen symboliin ei liity arvoa (koska heittomerkin sisällä sellaista ei tarvita) eikä se ole muuttuja (vaan symbolinen vakio).

Symbolityypin data-alkioiden erikoispiirre on, 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 oma yksikäsitteinen laatikkonsa.

Kalvoilla I.1 esitelty "symbolein suoritettava päättelyoperaatio" voidaan ajatella:

1. Jos datassa on symbolirakennelma `(on-ihminen` x`)` (missä x on mielivaltainen rakennelma)

2. niin säännöllä 'jos `(on-ihminen` y`)` niin `(on-kuolevainen` y`)' (missä y on säännön parametri)

3. voidaan tuottaa datarakennelma `(on-kuolevainen` x`)`.

Sääntö voidaan lausua vielä abstraktimmin ja korkeammalla kertaluvulla: 'jos $y \in Z$ niin $y \in U$, missä $Z \subseteq U$'.

Voidaan kirjoittaa operaatio `subset-rule` joka saa syötteekseen ominaisuuksien Z ja U nimet – symboleina – ja palauttaa niitä vastaavan päättelysäännön – funktiona symboliyhdistelmiltä $(Z\ x)$ yhdistelmille $(U\ x)$:

```
(define (subset-rule Z U)
  (lambda (Zy)
    (if (eq? (car Zy)
             Z)
        (cons U
              (cdr Zy))
        #f)))

(define multainen-saanto
  (subset-rule 'on-ihminen
               'on-kuolevainen))
```

Näin ohjelma sanoo "ominaisuus (jota kutsumme nimellä) `on-ihminen` on osajoukko ominaisuudesta `on-kuolevainen`, ja vastaava päättelysääntö on `multainen-saanto`".

Edellä symboleita käytettiin vain havainnollisina niminä kohdealueen käsitteille.

Kalvojen I.3 derivointiongelman ratkaisevassa Scheme-ohjelmassa [AS94,§2.3.2] symboleita (+, *) käytetään sen lisäksi eri "tyyppisten" lausekkeiden tunnuksina.

```
(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 tavatulle (syöte)symbolille jokin "arvo" jota ei kalvojen II.8.4 mukaisesti järjestelmä tee. Tämä kirjanpidon voi 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 arvo löytyy kirjastokutsulla (`assq 'key` L) joka palauttaa *ensimmäisen* parin (`key .` $\text{arvo}_{\texttt{key}}$) jos sellainen löytyy, ja muuten `#f`. Arvon antamiseksi tai päivittämiseksi riittää siis uuden parin lisääminen listan alkuun.

`assq` vertaa hakuavainta listan avaimiin kalvojen II.8.5 testillä `eq?`, joten se on luotettava vain symboliavaimille.

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

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

II.8.8 Parametrilistan otto

- `(lambda (`x_1 ... x_n`) ... )` antaa funktion joka vaatii *tasan* n argumenttia kuten kalvoilla II.6.4 kerrottiin.

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

  Kutsussa

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

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

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

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

  Matemaattisesti parametritön funktio on pelkkä vakio.

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

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

- `()` on "ei enää argumentteja"

- symboli y on "kaikki loput argumentit"

- pari $(x_i \ . \ P)$ on "ensimmäinen argumentti olkoon nimeltään x_i; jatka sitten seuraavien argumenttien sovittamista loppuihin parametreihin P" .

II.8.9 Parametrilistan anto

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

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

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

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

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

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

II.8.10 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 etukäteen kuinka pitkän seulalistan tarvitsisimme jotta siihen 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.8.8 lausekkeelle (`lambda () `e) ja että (`define (force `p`) (`p`)`); tätä tarkennetaan myöhemmin.

Laiskoilla listoilla ohjelmointikuri on seuraava:

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

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

Loppulistan ottaminen (eli "`cdr`") parista q hoituu 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 ennenkin – 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 $2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \ldots$; $3, 5, 7, 9, 11, \ldots$; $5, 7, 11, \ldots$; ...

```
(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 $\texttt{from}_{\texttt{low+1}}$`)`

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

- `(filter pred unfiltered)` palauttaa parin jonka

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

  `cdr` on lupaus suodattaa loputkin listasta jos tarpeen.

  Ensimmäisen alkion löytämiseksi saatetaan joutua lunastamaan listan `unfiltered` lupauksia.

- `(sieve numbers)` palauttaa parin jonka

  `car` on seuraava löydetty alkuluku ja

  `cdr` on lupaus poistaa sen monikerrat listasta `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 pakko.

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

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

Laiska ohjelmointi on siis *melko* tehokasta ja funktionaalista:

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

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

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

Tällöin *kaikki* käsiteltävät tietoalkiot – parit, luvut, funktiot – olisivat näkymättömien `delay`-operaatioiden sisällä, joihin ohjelmaa suoritettaessa kohdistettaisiin näkymättömiä `force`-operaatioita *mutta vasta silloin kun niiden arvoa tarvitaan*.

Suoritusmekanismissa varsinkin kalvojen II.5.4 *funktiokutsu* muuttuisi:

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

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

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

Kolme suoritusmekanismia:

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

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

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

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

Tämä mekanismi omaa kauniit formaalit ominaisuudet kalvojen II.6.5 lambdakalkyylissä.

Schemellä jokainen näistä on kohtuullisen vaivatonta.

II.9 Tilaperustaiset piirteet

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

`/usr/games/fortune`

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

- aliohjelmia matemaattisina funktioina ja
- ohjelman suoritusta lausekkeen sievennyksenä.

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

Puhtaasta osasta puuttuu joitakin piirteitä:

- Kalvoilla II.8.10 kerrottiin `delay`-kirjanpidosta joka tallettaa lupausten arvoja muistiin, mutta ei kerrottu miten.

  Muistin käyttöön viitattiin myös kalvoilla II.8.2 ja II.8.5.

  Muisti ja sen muokkaaminen on siis lisättävä.

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

  Syöte ja tulostus on siis lisättävä.

Kohdataan *tilan* käsite: laskulaitteen muistin ja ympäröivän maailman.

Sievennysmallissa ei ollut ohjelman muokattavissa olevaa tilaa, oli vain sievennyksen nykyinen välivaihe.

II.9.1 Ympäristöoppia

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

Kehys (frame) varastoi äärellisen kuvauksen symboleilta arvoille:

- Kuvaus on määritelty vain äärellisen monelle symbolille.
- Arvo jolle symboli kuvataan on joko nuoli kalvojen II.8.2 laatikkoon tai kalvojen II.6.4 `lambda`-lausekkeen arvona saatu funktio josta enemmän tuonnempana.

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

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

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

Ympäristö (environment) on nuoli kehykseen, tai pikemminkin siitä alkava *pino jaettuja kehyksiä*.

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

1. Ensin katsotaan onko ensimmäisen kehyksen kuvaus määritelty muuttujasymbolille s. Jos on, niin saadaan haluttu arvo.

2. Muuten jatketaan sitä ympäröivään kehykseen ja yritetään uudelleen.

3. Jos kehykset loppuvat eikä arvoa löytynyt, on kyseessä virhe: muuttujasymboli s ei olekaan määritelty lausekkeessa e.

II.9.2 Laskentasäännöt

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

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

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

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

Proseduurissa on kaksi osaa [AS96,kuva 3.2]:

- Nuoli nykyiseen ympäristöön jossa se luotiin.

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

- Viite proseduurin parametreihin ja runkoon.

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

Määrittely globaalissa ympäristössä (`define` s e)

- lisää tämän globaalin ympäristön kuvaukseen uuden muuttujasymbolin s
- ja sille arvon a joka saadaan evaluoimalla e tässä globaalissa ympäristössä

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

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

1. Evaluoidaan lausekkeille $e_0, \ldots, e_k$ arvot $a_0, \ldots, a_k$ nykyisessä ympäristössä p.
2. Tarkistetaan että a_0 on proseduuri jolla on parametrit $x_1, \ldots, x_k$ (tai jokin muu kalvojen II.8.8 muoto). Olkoon sen synty-ympäristö q ja runko α.
3. Luodaan uusi kehys r jonka ympäröijä on q ja jonka kuvaus on $x_i \mapsto a_i$. Evaluoidaan α ympäristössä r ja raportoidaan tulos sitä odottavaan ympäristöön p. [AS96,kuvat 3.3 ja 3.5]

   Jos tämä oli kalvojen II.7.1 loppukutsu, niin ohitetaan odottaja p ja raportoidaan suoraan siihen ympäristöön, joka odotti tulosta ympäristöltä p.

Järjestelmäjatke kalvoilta II.7.5 on juuri tämä evaluoitavien lausekkeiden α ”kuka raportoi arvonsa kenelle” -ketju.

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

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

Toinen tapa tehdä paikallisia määrittelyjä on kalvojen II.6.4 `let`-rakenteen rekursiivinen versio `letrec` [D96,§4.3;KCR98,§4.2.2]: siinä luotavat paikalliset muuttujat voi alustaa sellaisilla lausekkeilla joissa *voi viitata* luotaviin muuttujiin itseensä.

Näitä viittauksia *ei voi käyttää* ennen kuin muuttujat on luotu! Tämä lisäehto on täytetty ainakin silloin kun alustajat ovat `lambda`-lausekkeita, mikä on rakenteen yleisin käyttötapa [KCR98,§5.2.2].

Ehtolause kalvolta II.5.5 evaluoidaan vastaavasti kuin ennen: ensin ehto evaluoidaan nykyisessä ympäristössä näillä uusilla säännöillä, sitten tuloksen perusteella valitaan haara josta jatketaan evaluointia.

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

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

`(eval` d p`)`

joka yrittää suorittaa rakenteisen data-alkion kuten

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

ympäristössä p.

Mahdollisia ympäristöjä p ovat:

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

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

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

II.9.3 Arvon vaihtaminen

Nyt kun kalvojen II.9.2 kertovat missä kalvojen II.9.1 ympäristöistä kulloinkin ollaan, voidaan esitellä Scheme-kielen operaatio muuttujasymbolille annetun arvon muuttamiseen [AS96,§3.1;D96,§4.5;KCR98,§4.1.6].

Kun *sijoitusoperaatio*

(set! s e)

suoritetaan nykyisessä ympäristössä p, niin:

- Evaluoidaan e ympäristössä p arvoonsa a.

- Haetaan se kuvaus l josta muuttujasymbolille s katsottaisiin arvo nykyisessä ympäristössä p.

  Muuttujasymboli s luetaan sellaisenaan (ikään kuin se olisi heittomerkin sisällä) eikä evaluoida; `set!` on erikoismuoto, ei tavallinen funktio.

- Päivitetään kuvausta l siten, että s kuvautuukin nyt arvolle a.

  Muutos näkyy juuri niihin evaluointiaskeleisiin joissa viitataan symbolin s arvoon kuvauksessa l.

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

Konventio: Muistia muuttavien operaatioiden nimet päättyvät huutomerkkiin!

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

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

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

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

missä lausekkeista tuotetaan ensin kalvojen II.8.8 "thunk" jota kutsutaan heti sen jälkeen.

Esimerkki [AS96,§3.2.3]:

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

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

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

```
(define balance 100)

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

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

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

II.9.4 Parin osien muuttaminen

Kalvoilla II.9.3 annettiin operaatio `set!` muuttujasymbolin 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 muuttujasymbolin `b` arvo onkin `(y)` — mutta niin on myös muuttujasymbolin `a` koska ne *jakoivat* saman muistirakenteen!

Kutsun `(sour-prize)` jälkeen muuttujasymbolin `c` arvona on *kehällinen* pari `(x . (x . (x . ... )))`!

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 paikassa 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 paikan i (osoittamaksi) alkioksi lausekkeen e arvon.

`(vector->list` v`)` palauttaa listan vektorin sisältämistä alkioista samassa järjestyksessä: ensimmäisenä paikan 0 alkio, toisena paikan 1, jne.

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

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

Vektorivakiot voidaan kirjoittaa kuten kalvojen II.8.4 listavakiot kunhan eteen lisätään merkki `#`: `'#(a b c)` 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 alkaen 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, tai palauttaa `#f` ellei onnistu.

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

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

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

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 jos syötteenlukurutiini `read` tuloksen lukisi, se muodostaisi tulosteesta sellaisen Scheme-alkion d', joka olisi mahdollisimman paljon alkuperäisen alkion d kaltainen.

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

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

(`display` d) tulostaa tietoalkion d ihmiselle miellyttävämmässä muodossa piilottamalla em. `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 siis tätä arvoa, jota `with-output-to-file` vaihtaa.

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

`(open-output-file` v`)` on alimman abstraktiotason liittymä: se (yrittää avata ja) palauttaa tiedostoon nimeltä v tulostusportin g, jonka 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.”

Oletusssyöteportinkin (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-port`.

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

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

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

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: sääntöjärjestelmä

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

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

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

- joukon ongelmanratkaisusääntöjä ja
- ongelman

ja kertoo voiko ongelman ratkaista näillä säännöillä.

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

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

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

$$\texttt{(ihminen (sokrates))}$$

on termin

$$\texttt{(ihminen } x\texttt{)}$$

erikoistapaus asettamalla $x = \texttt{(sokrates)}$.

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

$$\texttt{(kuolevainen } x\texttt{)}$$

haluttu tulostermi

$$\texttt{(kuolevainen (sokrates))}.$$

On siis pidettävä kirjaa muuttujatermien arvoista. Tähän sopii assosiaatiolista.

(Listaus jatkuu yli kalvorajan.)

```
; 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> occurrence
; 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 <binding>)
; would occurr in (binding:expand <compound> <binding>). (The expansion
```

```
; 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 termit 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` x y`)`

joista jälkimmäistä käytettäessä y:n isä z ei löydy uusfaktaa `(esi-isa` x y`)` samastamalla, joten jos 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 keskeinen 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 necessary:
(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 subterms
; 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.)

III.3 Sääntöjen esitys

Esitetään sääntö

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

listana

$$(\underbrace{\text{uustermi}}_{\text{pää}}\ \underbrace{\text{termi}_1 \ldots \text{termi}_k}_{\text{vartalo}})$$

jossa kutakin faktaa vastaava kalvojen III.1 termi on (yleensä) rakennetermi.

Tämä siksi että faktan uloin funktori *väittää* (yleensä) jotakin alitermeistään, jotka puolestaan esittävät niitä *alkioita* joista on kyse:

```
(ihminen (sokrates))
```

väittää että "`sokrates` on `ihminen`".

Tämän vuoksi ulointa funktoria kutsutaankin *predikaatiksi*: faktaa vastaavan väitelauseen verbiksi.

Alitermit ovat silloin muita lauseenjäseniä: subjekti, objekti, jne.

Säännöt

1. Sokrates on ihminen

2. kaikki ihmiset ovat kuolevaisia

esitetään siis muodossa

1. `((ihminen (sokrates)))`

   – tyhjä vartalo koska ei **jos**-esiehtoja!

2. `((kuolevainen` x`) (ihminen` x`))`

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

Näistä säännöistä muodostuu *sääntö(tieto)kanta* josta kysellään nykyiseen tilanteeseen soveltuvia sääntöjä.

Valitaan tälle kannalle yksinkertaisin mahdollinen esitys: lista kirjoitusjärjestyksessä.

```
(((ihminen (sokrates)))
 ((kuolevainen x)
  (ihminen x)))
```

III.4 Ratkaisun etsintä

Kalvojen III.3 sääntökantaa voidaan käyttää seuraavasti ratkaisemaan onko Sokrateskin kuolevainen:

1. Esitetään ongelmakin faktana `(kuolevainen (sokrates))` jota väitetään todeksi.

2. Etsitään kannasta *sopiva* sääntö, eli sellainen jonka pää samastuu kalvojen III.2 tapaan ongelmafaktaan. Nyt ainoa vaihtoehto on `((kuolevainen` x`) (ihminen` x`))` yleisimmällä samastimella $x =$ `(sokrates)`.

3. Valitun säännön vartalosta saadaan uusi ongelma `(ihminen (sokrates))` joka ratkaistaan samoin.

4. Se ratkeaa tuottamatta lisää ongelmia, joten koko ongelma on ratkaistu.

Sääntöjä käytetään tässä *takaperin*: Haluan että uusfakta (pää, **niin**-osa) olisi totta, joten yritän varmistaa että sen ehtona oleva fakta (vartalo, **jos**-osa) on myös totta.

Edellä oletettiin että kohdeongelma eli *maali* (goal) on aina yksi fakta. Kalvojen III.2 säännöissä

```
((esi-isa x y)
 (isa x y))

((esi-isa x y)
 (isa x z)
 (esi-isa z y))
```

jälkimmäisen käyttö synnyttää kuitenkin kaksi eri aliongelmaa `(isa x z)` ja `(esi-isa z y)` jotka on ratkaistava *samalla* `z` eli "yhdessä".

Maali onkin *joukko* ratkaistavia faktoja (joita sanotaan *ali*maaleiksi). Esitetään yksinkertaisuuden vuoksi tämä(kin) joukko listana (duplikaatteja poistamatta).

Samalla nähdään että jokaisella käytöllä halutaan *eri* z koska tehdään ketjua

`(isa x` z_1`), (isa` z_1 z_2`), (isa` z_2 z_3`), ... , (isa` z_p `y)`

missä p on henkilöiden `x` ja `y` välissä olevien sukupolvien lukumäärä.

Tämän vuoksi säännön muuttujat *nimetään uudelleen* ennen käyttöä sellaisiksi, joita ei ole käytetty ennen (merkitystä muuttumatta):

```
((esi-isa X0 X1) (isa X0 X2) (esi-isa X2 X1))
```

Ratkaisun etsijää ohjelmoitaessa on tehtävä 2 päätöstä:

- *Miten seuraava ratkaistava alimaali valitaan?*

  ”Koska jokainen niistä on kuitenkin joskus ratkaistava, ei kai järjestyksellä ole niin väliä?”

  Tämä intuitio voidaan todistaa oikeaksi [L87,§9].

  Ratkaisun löytymiseen valinta ei siis vaikuta, löytymisnopeuteen kylläkin ...

  Yksinkertaisuuden vuoksi päätetään ottaa aina *ensimmäinen* alimaali, korvataan se käytetyn säännön vartalolla, ja saadaan seuraava maali.

  Maali on siis *pino* ratkaisua odottavia alimaaleja.

  Sääntöä ratkotaan siis sen *vartalon kirjoitusjärjestyksessä*.

  Silloin maalit ovat esimerkiksi

```
1   ((esi-isa (aabraham) (jaakob)))
2   ((isa (aabraham) X2) (esi-isa X2 (jaakob)))
3   ((esi-isa (iisak) (jaakob)))
4   ((isa (iisak) (jaakob)))
5   ()
```

  ja etsintä päättyy onnistumiseen eli tyhjään maaliin.

- *Miten valitaan sääntö jota käytetään tuohon valittuun alimaaliin?*

  Yksinkertaisuuden vuoksi päätetään *kokeilla sääntöjä niiden kirjoitusjärjestyksessä* muistaen samalla jatkaa *edellistä kokeilua* jos nykyinen päätyy umpikujaan.

  Esimerkissä on maalina $1\frac{1}{2}$ umpikuja `(isa (aabraham) (jaakob))` – sille ei ole yhtään sääntöä – josta *peruutetaan* yrittämään toista (nyt oikeaa) `esi-isa`-sääntöä.

  Periaate on kuitenkin "epäreilu" koska se suosii aikaisempia sääntöjä myöhäisempien kustannuksella:

  \+ Tehokkaampi etsintäohjelma.

  "Reilut" vaihtoehdot söisivät (vielä) huomattavasti enemmän *muistia*!

  – Sääntöjen järjestys tulee tärkeäksi.

  Väärällä järjestyksellä etsintä voi jäädä *ikuiseen silmukkaan*!

  Tekoälyssä tarkastellaan tätä (ja) fiksumpia etsintäperiaatteita.

Kun ratkaisu löydetään, kutsujasta kiinnostavaa on (onnistumisen lisäksi myös) etsinnän kuluessa kerääntynyt muuttujien sidonta. Esimerkiksi maalilla

(esi-isa x y)

haetaan (kaikkia) pareja (x, y) missä x on y:n esi-isä, ei pelkkää tietoa ”kyllä, sellaisia pareja todellakin on”.

Etsintä voi siis *epä*onnistua säännön valinnassa, päätyä umpikujaan ja peruuttaa. Kalvojen II.8.10 laiskat listat mahdollistavat tällaisen etsinnän ohjelmoimisen *listana onnistumisia*:

Pidetään etsinnän kuluessa yllä (eräänlaisena kalvojen II.7.2 ja II.7.3 kerääjäparametrina) lupausta että pystymme haluttaessa tuottamaan myös ne ratkaisut jotka saadaan tekemällä nykyinen valinta toisin.

Vastauksena palautetaan siis pari

(seuraava ratkaisusidonta . lupaus muista pareista)

tai () jos ratkaisusidontoja ei enää löydy.

Lisätään vielä välineet etsinnän edistymisen seuraamiseen.

(Listaus jatkuu yli kalvorajojen.)

```
; A <rule> consists of the <head> and the <body>
; where <head> is a (usually <compound>) <term>
; and the <body> is a sequence of (usually <compound>) <term>s.
;
; A <guide> is a sequence of these <rule>s to be tried in order.
;
; A <goal> is a sequence of (usually <compound>) <term>s which
; represents a problem to solve using a given <guide>.

; A <rule> is represented as a (nonempty) list of terms,
; the first of which is the <head> and the rest form the <body>.
;
; A <guide> is represented as a list of <rule>s.
;
; A <goal> is represented as a list of <term>s.

; Return the <head> of this <rule>.
(define rule:head car)

; Return the <body> of this <rule> as a list.
(define rule:body cdr)

; Return the next <rule> of this <guide> to try.
(define guide:next car)

; Return the remaining <guide> to try as a list.
(define guide:rest cdr)

; Have all the <rule>s of this <guide> already been tried?
(define guide:none? null?)

; Return the next <term> to solve from this <goal>.
(define goal:next car)

; Return the unsolved <term>s from this <goal> as a list.
(define goal:rest cdr)

; Has this <goal> been fully solved?
(define goal:none? null?)

; Rename the <variable>s within a given <rule> with previously
; unused <variable>s while retaining the meaning of the <rule>.
(define rule:fresh
  (letrec ((fresh-term
            (lambda (stale-term
                     fresh-binding)
              (cond ((term:variable? stale-term)
```

```
                    (if (binding:get stale-term
                                     fresh-binding)
                        fresh-binding
                        (binding:add stale-term
                                     (term:new-variable)
                                     fresh-binding)))
                   ((term:compound? stale-term)
                    (fresh-subterms (term:subterms stale-term)
                                    fresh-binding)))))
          (fresh-subterms
           (lambda (subterms
                    fresh-binding)
             (if (null? subterms)
                 fresh-binding
                 (fresh-subterms (cdr subterms)
                                 (fresh-term (car subterms)
                                             fresh-binding))))))
    (lambda (<rule>)
      (term:subterms
       (let ((term-from-rule (term:combine '<functor>
                                           <rule>)))
         (binding:expand term-from-rule
                         (fresh-term term-from-rule
                                     binding:empty)))))))

; Should we output a trace while we work?
(define rule:trace #f)
(define rule:tracer
  (let ((mention
         (lambda (title
                  datum)
           (display title)
           (display datum)
           (newline))))
    (lambda (rule
             goal
             vars)
      (newline)
      (mention "RULE: " rule)
      (mention "GOAL: " goal)
      (mention "VARS: " vars)
      (display "PURSUE THIS GOAL [y/n]? ")
      (shell:more?))))

; A given <goal> is solved by a given <guide> as follows.
(define rule:solve
  (lambda (<guide>
           <goal>)
    (define engine
```

```
      (lambda (guide    ; The <rule>s not yet tried for this goal.
               goal     ; This goal.
               binding  ; Collected binding.
               choices) ; Delayed other solutions.
        (cond ((goal:none? goal)
               ; Solved! Report collected binding and other solutions.
               (cons binding
                     choices))
              ((guide:none? guide)
               ; Failed! Try to find another solution.
               (force choices))
              (else
               ; Take next <rule> with its variables renamed
               ; and try to unify its head with first sub<goal>.
               (let ((next-rule (rule:fresh (guide:next guide))))
                 (let ((next-binding (unify (goal:next goal)
                                            (rule:head next-rule)
                                            binding)))
                   (if (and next-binding
                            ; Trace the process if requested:
                            (or (not rule:trace)
                                (rule:tracer next-rule
                                             goal
                                             binding)))
                       ; They unified! Combine <body> with other
                       ; sub<goal>s and start solving this problem.
                       ; Other solutions would have been to try
                       ; next <rule> instead.
                       (engine <guide>
                               (append (rule:body next-rule)
                                       (goal:rest goal))
                               next-binding
                               (delay (engine (guide:rest guide)
                                              goal
                                              binding
                                              choices)))
                       ; They did not unify: try next <rule> instead.
                       (engine (guide:rest guide)
                               goal
                               binding
                               choices)))))))
    ; Initially neither bindings nor other solutions.
    (engine <guide>
            <goal>
            binding:empty
            (delay '())))))

; List the variables in this <goal> without repetitions.
(define goal:variables
```

```
(letrec ((search-term
          (lambda (<term>
                   found)
            (cond ((term:variable? <term>)
                   (if (member <term>
                               found)
                       found
                       (cons <term>
                             found)))
                  ((term:compound? <term>)
                   (search-terms (term:subterms <term>)
                                 found)))))
         (search-terms
          (lambda (<subterms>
                   found)
            (if (null? <subterms>)
                found
                (search-terms (cdr <subterms>)
                              (search-term (car <subterms>)
                                           found))))))
  (lambda (<goal>)
    (search-terms <goal>
                  '())))
```

(Listaus jatkui yli kalvorajojen.)

Sääntöjärjestelmämme toimii mutta tulostusasu on varsin raaka. Lisätään lopuksi pieni komentoriviympäristö käyttäjän vaivojen helpottamiseksi.

Samalla saadaan esimerkki kalvojen II.9.7 tulostus- ja kalvojen II.9.8 syötekirjastofunktioiden alkeiskäytöstä.

(Listaus jatkuu yli kalvorajan.)

```
; Read in a file of Scheme data as a list.
(define shell:read-file
  (letrec ((reader
            (lambda ()
              (let ((next (read)))
                (if (eof-object? next)
                    '()
                    (cons next
                          (reader)))))))
    (lambda (name)
      (with-input-from-file name
        reader))))

; Wait for the user to press 'y/n' and [return].
(define shell:more?
  (lambda ()
    (let ((reply (char-downcase (read-char))))
      (cond ((char=? reply
                     #\y)
             #t)
            ((char=? reply
                     #\n)
             #f)
            (else
             (shell:more?))))))

; Converse with the user.
(define shell:run
  (lambda (<guide>)
    (letrec ((next-goal
              (lambda ()
                (let ((<goal> (begin
                                (display "NEXT GOAL [() EXITS]? ")
```

```
                        (read))))
            (if (null? <goal>)
                (display "EXIT.")
                (next-answer (goal:variables <goal>)
                             (rule:solve <guide>
                                         <goal>))))))
        (next-answer
         (lambda (variables
                  answers)
           (if (null? answers)
               (begin
                 (display "NO (MORE) ANSWERS.")
                 (newline)
                 (next-goal))
               (begin
                 (if (null? variables)
                     (begin
                       (display "TRUE.")
                       (newline))
                     (for-each (lambda (variable)
                                 (display variable)
                                 (display " = ")
                                 (display (binding:expand
                                           variable
                                           (car answers)))
                                 (newline))
                               variables))
                 (display "ANOTHER ANSWER [y/n]? ")
                 (if (shell:more?)
                     (next-answer variables
                                  (force (cdr answers)))
                     (begin
                       (display "(POSSIBLE) OTHER ANSWERS OMITTED.")
                       (newline)
                       (next-goal)))))))))
(next-goal))))
```

(Listaus jatkui yli kalvorajan.)

III.5 Järjestelmän käyttöä

Olkoon sukupuu esitetty seuraavina faktoina:

`(daughter` x y z`)` : x on äidin y ja isän z tytär.

`(son` x y z`)` : x on äidin y ja isän z poika.

Niillä voidaan määritellä säännöillä käsitteet

`(mother` x y`)` : x on y:n äiti.

`(grandmother` x y`)` : x on y:n äidinäiti.

`(ancestress` x y`)` : x on joku y:n esiäiti.

```
((mother x y) (daughter y x z))
((mother x y) (son y x z))

((grandmother x y) (mother x z) (mother z y))

((ancestress x y) (mother x y))
((ancestress x y) (mother x z) (ancestress z y))
```

IV Ohjelmointikieli Prolog

Tutustutaan lopuksi logiikkaohjelmointikielen Prolog (eli "Programming in Logic") perusteisiin [B01,§1-7;CM87,§1-6;SS86,§1-12].

Suuri osa työstä on jo tehty: kalvojen III sääntöjärjestelmän toteutus on itse asiassa (hyvin) yksinkertainen Prolog-tulkki!

Jatkossa kerrataan siis uudesta näkökulmasta mitä silloin tulikaan tehtyä, ja kerrotaan mitä uusia piirteitä siihen on lisätty tehtäessä kokonaista ohjelmointikieltä.

Logiikkaohjelmoijat siteeraavat usein "yhtälöä"

$$\text{algoritmi} = \text{logiikka} + \text{kontrolli}$$

jonka mukaan ohjelmoinnissa voidaan ja pitää erottaa toisistaan

- se osa joka kuvailee itse ratkaistavan ongelman
- siitä osasta joka kertoo miten tietokone ratkaisuun päätyy.

IV.1 Historiaa

Prolog syntyi 1970-luvun alussa (1972?) kun kaksi samoin ajattelevaa tutkijaa löysi toisensa:

Alain Colmerauer Marseillen yliopistosta (Ranskasta) tutki käytännössä *luonnollisen kielen käsittelyä* tietokoneella.

Robert Kowalski Edinburghin yliopistosta (Skotlannista) tutki teoriassa korkean abstraktiotason *laskenta- ja ohjelmointimalleja*.

Kumpikin käytti työkalunaan logiikkaa, jonka *todistusteoriassa* – opissa todistusten laatimisesta – oli päästy 1960-luvulla niin pitkälle, että voitiin ohjelmoida automaattisia todistusten etsijöitä. Tämä olikin ajan tekoälytutkimuksen päälinjoja.

Colmerauer ja Kowalski käänsivät nämä ideat toisin päin: itse *ohjelma kirjoitettakoon logiikalla* ja sen *suoritus olkoon todistuksen etsintää*.

Standardointi alkoi 1985 Edinburghin murteen pohjalta ja ISO hyväksyi tuloksen 1995 [DEC96].

IV.2 Perussyntaksi

Kalvoihin III.1 ja III.3 verrattuna [B01,§1.1-1.3]:

Muuttuja tunnistetaan siitä että se alkaa joko ISOLLA kirjaimella tai alaviivamerkillä '_'.

Se jatkuu (isoilla tai pienillä) kirjaimilla, numeroilla tai alaviivamerkeillä.

Rakenteinen termi kirjoitetaan matematiikasta tutussa muodossa $f(\tau_1, \ldots, \tau_k)$ listamuodon $(f\ \tau'_1\ \ldots\ \tau'_k)$ sijasta.

Jos $k = 0$ niin (tyhjät) sulut jätetään pois, eli vakiot kuten `(sokrates)` kirjoitetaan `sokrates` eikä `sokrates()`.

Prolog-kielenkäytössä merkintä f/k tarkoittaakin funktoria f jonka paikkaluku on k.

Funktori f kirjoitetaan **atomi**na eli

- joko kuten muuttuja mutta pienellä alkukirjaimella
- tai (lähes) mielivaltaisena jonona merkkejä yksinkertaisten lainausmerkkien '' sisällä.

Sääntö kirjoitetaan muodossa

$$p\texttt{:-}q_1\texttt{,}q_2\texttt{,}q_3\texttt{,}\ldots\texttt{,}q_m\texttt{.}$$

listamuodon

$$(p' \;\; q'_1 \;\; q'_2 \;\; q'_3 \;\; \ldots \;\; q'_m)$$

sijasta.

Pää p on aina rakenteinen termi eli muotoa $r(\tau_1, \ldots, \tau_k)$; tällöin sääntö on yksi niistä, jotka *määrittelevät* yhdessä predikaatin r/k.

Vartalon termit q_i ovat myös (yleensä) rakenteisia, ja esittävät miten vastaavia predikaatteja käytetään predikaatin r/k (tämän säännön) määrittelemisessä.

Vertaa klassinen sanakirjamääritelmä: "ihminen on höyhenetön kaksijalkainen eläin" eli

```
ihminen(X):-
        hoyheneton(X),
        kaksijalkainen(X),
        elain(X).
```

Jos $m = 0$, eli kyseessä on pelkkä fakta p, niin pään ja (tyhjän) vartalon erottava 'kaula' eli ':-' jää myös pois.

Esimerkiksi Prolog-ohjelmassa

```
p(o,Y,Y).
p(s(X),Y,s(Z)):-
        p(X,Y,Z).
```

määritellään predikaatti `p`/3 yhdellä faktalla ja yhdellä säännöllä.

Niissä käytetään vakiota `o`/0 ja funktoria `s`/1.

Muuttujien `X`, `Y` ja `Z` näkyvyysalue on yksi sääntö, eli päättyy pisteeseen '.':

- Faktan `Y` on eri kuin säännön `Y`.

- Säännön pään `X` on sama kuin vartalon `X`.

Sovitaan merkintöjen selkeyden vuoksi, että tämän esimerkkiohjelman nimi on **sum**, ja että merkintä $\mathtt{s}^n\tau$ tarkoittaa sen yhteydessä syvää rakenteista termiä

$$\underbrace{\mathtt{s(s(s(...s(}\tau\mathtt{)...)))}}_{n \text{ kpl.}}.$$

IV.3 Semantiikka

Logiikkaohjelmoinnissa ohjelmien merkitystä voidaan kuvailla kolmesta eri näkökulmasta:

Suorituslähtöisesti muiden ohjelmointikielten tapaan [B01,§2.4].

"Mitä askeleita ohjelma tekee?"

Ohjelma **sum** siirtää toisen argumenttinsa kolmanteen, ja sen päälle kaikki ensimmäisen s-funktorit.

Malliteoreettisesti käyttäen *loogisen totuuden* käsitettä [B01,§2.3;L87,§6].

"Millaisen maailman ohjelma kuvailee?"

Ohjelma **sum** kertoo mitä merkitsee väite "kolmas luku on kahden ensimmäisen summa".

(Loogiset yksityiskohdat sivuutetaan; logiikkaa opiskelleet kohtaavat tuttujen käsitteiden erikoistapauksia.)

Todistusteoreettisesti käyttäen loogisen *todistuksen* käsitettä [L87,§7-10].

Ohjelma **sum** on aksiomatisointi jolla voidaan todistaa näitä väitteitä "kolmas luku on kahden ensimmäisen summa".

- Malliteoria kuvaa ohjelman merkityksen ilman viittauksia mihinkään laskentasääntöihin.

  (Ohjelma on siis symbolikasauma, jolle ihmisellä on mielessään tulkinta ...)

- Todistusteoria kuvaa sellaiset mahdolliset laskentasäännöt, jotka ovat malliteorian mukaisia.

  (... joten säännöt ovat siis niitä symbolisia operaatioita, jotka eivät sodi tulkintaa vastaan ...)

- Suorituskone toteuttaa (jonkin) strategian jolla todistusteorian sallimia laskentasääntöjä yritetään soveltaa laskentatehtäviä ratkaistaessa.

  (... joten ohjelmoidaan tietokone ratkomaan laskutehtäviä näillä operaatioilla!)

Samaa kolminaisuutta käytetään yleisemminkin ohjelmointikielten teoriassa, mutta logiikkaohjelmoinnissa se näkyy harvinaisen puhtaana.

IV.3.1 Suoritussemantiikka

Tarkastellaan predikaatin p/k määritelmää n säännöllä:

$$p(t_{(1,1)},\ldots,t_{(1,k)}) \texttt{ :- } q_{(1,1)},\ldots,q_{(1,m_1)}.$$
$$\vdots$$
$$p(t_{(n,1)},\ldots,t_{(n,k)}) \texttt{ :- } q_{(n,1)},\ldots,q_{(n,m_n)}.$$

Sen voi lukea määritelmänä aliohjelmalle nimeltä p jolla on k parametria.

Jokaisessa haarassa $1 \leq i \leq n$ termit $t_{(i,1)},\ldots,t_{(i,k)}$ määräävät *hahmon* johon aliohjelmakutsun $p(\tau_1,\ldots,\tau_k)$ argumenttien $\tau_1,\ldots,\tau_k$ on *sovittava* (kalvojen III.2) samastuksen mielessä.

Haara i on *mahdollinen* jos argumentit sopivat hahmoon; muuten haara sivuutetaan. Hahmo siis kertoo minkä "muotoisille" syötteille τ_j tämä haara on tarkoitettu.

Koska nyt samastetaan argumentteja parametrihahmoihin, on mukana myös sidonta θ jota laajennetaan suorituksen edetessä, ja joka hoitaa parametrinvälityksen. (Vrt. kalvojen II.9.1 ympäristöt.)

Loogikot kirjoittavat $\tau\theta$ operaatiomme `(binding:expand` τ θ`)` vastineen.

(Ali)ohjelman lukuperiaate on tuttu ”ylhäältä alas, vasemmalta oikealle”:

Etsitään ensimmäinen mahdollinen haara i, eli sellainen jolla operaation

$$\theta_0 = (\texttt{unify}\ p(\tau_1, \ldots, \tau_k)\ p(t_{(i,1)}, \ldots, t_{(i,k)})\ \theta)$$

vastine onnistuu.

Ensin kuitenkin nimettiin haaran i muuttujat uudelleen, jotta vanhat muuttujanarvot θ ja uudet parametrit eivät sekoittuisi keskenään.

Jatketaan kokeiltavan haaran i vartaloon, eli

1. tehdään samalla tavalla kutsu $q_{(i,1)}$ sidonnalla θ_0, josta saadaan vastaukseksi sidonta θ_1,

2. tehdään kutsu $q_{(i,2)}$ sidonnalla θ_1, josta saadaan sidonta θ_2, ...

Lopulta vastataan kysyjälle että kutsu $p(\tau_1, \ldots, \tau_k)$ laajensi alkusidonnat θ loppusidonnoiksi θ_{m_i}.

Entä jos jokin seuraava kutsu $q_{(i,j+1)}$ ilmoittaakin, ettei se pysty laajentamaan edelliseltä kutsulta $q_{(i,j)}$ saamaansa sidontaa θ_j sidonnaksi θ_{j+1}?

Silloin *peruutetaan* (backtrack) takaisin edelliseen kutsuun $q_{(i,j)}$, ja pyydetään siltä *jotakin toista* sidonnan θ_{j-1} laajennusta sidonnaksi θ'_j.

Jos tämän kokeiltavan haaran i vartalosta peruutetaan takaisin päähän asti, niin tämä kutsu $p(\tau_1, \ldots, \tau_k)$ sidonnoilla θ on sittenkin suoritettava käyttäen jotakin toista haaraa.

Jatketaan siis seuraavan mahdollisen haaran etsimistä haarasta $i + 1$ alkaen. Jos sellainen löytyy, niin kokeillaan sitä.

Jos taas haarat loppuvat kesken, niin vastataan kysyjälle "kutsu $p(\tau_1, \ldots, \tau_k)$ ei pysty laajentamaan sidontaa θ". Tämä aiheuttaa peruutusta vuorostaan kysyjässä.

Vastaavasti myös kysyjältä voi tulla pyyntö "vaikka kutsu $p(\tau_1, \ldots, \tau_k)$ laajensikin sidontansa θ onnistuneesti sidonnaksi θ_{m_i} haaralla i, ei tulos sopinutkaan myöhempään laskentaa, joten voisinko sittenkin saada *jonkin toisen* vaihtoehdon θ'_{m_i}?"

Silloin peruutetaan takaisin viimeiseen onnistuneeseen kutsuun $q_{(i,m_i)}$, ja yritetään laajentaa sen saamaa sidontaa θ_{m_i-1}.

Perimmäisenä kysyjänä on käyttäjä, joka saattaa olla tyytymätön hänelle tulostettuun vastaukseen.

Prolog-ohjelmien suorituksessa vuorottelee siis:

Etenemisvaihde jonka aikana suoritetaan aina heti seuraava mahdollinen haara.

Kun näin ajaudutaan umpikujaan, kytketään päälle ...

Peruutusvaihde jonka aikana ohjelman suoritusta ”kelataan takaisinpäin” edelliseen sellaiseen kohtaan, joka olisi voitu tehdä toisinkin.

Siinä kohdassa valitaankin nyt toisin, ja jatketaan taas etenemisvaihteella ...

Samoin aliohjelmilla ei ole kiinteitä syöte- ja tulosparametreja, vaan jokainen kutsu lukee syötteenään sille annettua sidontaa, ja tuottaa tuloksenaan siitä laajennettua versiota.

Esimerkiksi predikaattia `p/3` voi käyttää laskemaan yhteen $1 + 2$ kutsulla `p(s(o),s(s(o)),Z)` eikä mitään peruutusta tapahdu, koska etenemisvaihtoehtoja on aina tasan yksi, ja lopuksi onnistutaan saamaan vastaus `Z=s(s(s(o)))`.

Kääntäen luku 3 voidaan hajoittaa summiksi $0 + 3, 1 + 2, 2 + 1, 3 + 0$ kutsulla `p(X,Y,s(s(s(o))))` jossa kukin käyttäjän pyytämä peruutus tuottaa seuraavan vastauksen.

```
pajakari$ /opt/pl/bin/pl
Welcome to SWI-Prolog (Version 4.0.1)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('sum.pl').
% sum.pl compiled 0.01 sec, 652 bytes

Yes
?- p(s(o),s(s(o)),Z).

Z = s(s(s(o))) ;

No
?- p(X,Y,s(s(s(o)))).

X = o
Y = s(s(s(o))) ;

X = s(o)
Y = s(s(o)) ;

X = s(s(o))
Y = s(o) ;

X = s(s(s(o)))
Y = o ;

No
?-
```

IV.3.2 Mallisemantiikka

Olkoon L tarkasteltava ohjelma. Sen sisältämät funktorit voidaan (yleensä) jakaa predikaatteihin P_L ja muihin funktoreihin F_L.

Esimerkiksi $P_{\textbf{sum}} = \{\mathtt{p}/3\}$ ja $F_{\textbf{sum}} = \{\mathtt{o}/0, \mathtt{s}/1\}$.

(Ranskalainen Jacques Herbrand kehitti todistusteoriaa 1930-luvulla.)

Annetun ohjelman L Herbrandin ...

Universumi U_L on ne muuttujattomat termit jotka voidaan muodostaa ilman predikaatteja, eli pienin sellainen joukko, että

- jos $a/0 \in F_L$, niin myös $a/0 \in U_L$ ja
- jos $f/k \in F_L$ ja $\tau_1, \ldots, \tau_k \in U_L$ (missä $k > 0$), niin myös $f(\tau_1, \ldots, \tau_k) \in U_L$.

$$
\begin{aligned}
U_{\textbf{sum}} = {} & \{\mathtt{o}\} \\
& \cup \{\mathtt{s(o)}\} \\
& \cup \{\mathtt{s(o)}, \mathtt{s(s(o))}\} \\
& \vdots \\
& = \{\mathtt{s}^n\mathtt{o} \colon n \in \mathbb{N}\}\,.
\end{aligned}
$$

Kanta B_L on ne faktat jotka voidaan muodostaa universumin U_L termeistä, eli

$$B_L = \{p(\tau_1, \ldots, \tau_k) : p/k \in P_L\}.$$
$$B_{\textbf{sum}} = \left\{\texttt{p(s}^a\texttt{o,s}^b\texttt{o,s}^c\texttt{o)} : a, b, c \in \mathbb{N}\right\}.$$

Malli on mikä tahansa kannan B_L osajoukko, johon on poimittu tosiksi ainakin niin paljon faktoja, että ohjelman L kaikki *lauseet* (clauses, eli säännöt ja faktat) ovat tosia:

Olkoon θ mielivaltainen muuttujien sidonta universumin U_L alkioihin.

- Malliin poimitaan (muuttujaton) fakta $p\theta$ ohjelman L jokaiselle faktalle p.

  Haluamme näet pitää totena faktaa p *kaikilla mahdollisilla sen muuttujien saamilla arvoilla*.

- Jos malliin on jo poimittu faktat $q_1\theta, \ldots, q_n\theta$ ja ohjelmassa on sääntö p**:-**q_1**,** $\ldots$ **,**q_n**.** niin poimitaan myös $p\theta$.

  Haluamme näet pitää totena *ehtolausetta*

  jos q_1 **ja** $\ldots$ **ja** q_n **niin myös** p

  jonka loogikko kirjoittaisi *implikaationa*

  $$\forall \vec{X}.q_1 \wedge \ldots \wedge q_n \rightarrow p$$

  missä $\vec{X}$ kattaa kaikki säännön muuttujat. (Ns. *Hornin lauseena*.)

Osoittautuu että on olemassa *pienin* malli $M_L \subseteq B_L$, ja se saadaan poimimalla mukaan vain ne faktat, jotka nämä ehdot pakottavat, eikä mitään muuta.

Tämä M_L on ohjelman L deklaratiivinen merkitys, ja ohjelman L suorittaminen annetulla (muuttujattomalla) maalifaktalla p on kysymykseen "Onko $p \in M_L$?" vastaamista.

(Tarkkaan ottaen joukkoon F_L on lisättävä myös maalifaktan p funktorit ...)

Kalvojen IV.3.1 menetelmä on yksi algoritmi vastauksen laskemiseksi, kuitenkin *epätäydellinen*:

- algoritmi saattaa joutua ikuiseen silmukkaan löytämättä malliteorian mukaista vastausta

- koska algoritmi kokeilee sääntöjä kiinteässä järjestyksessä, joka saattaa olla tällä kertaa väärä. Malliteoriassa säännöt ovat tasa-arvoisia, mutta sellainen algoritmi söisi liikaa työmuistia.

Faktasta: $\mathtt{p(o,s^{}}^{b}\mathtt{o,s^{}}^{b}\mathtt{o)} \in M_{\mathbf{sum}}$ kaikilla $b \in \mathbb{N}$.

Säännöstä: jos $\mathtt{p(s^{}}^{a}\mathtt{o,s^{}}^{b}\mathtt{o,s^{}}^{c}\mathtt{o)} \in M_{\mathbf{sum}}$ niin myös $\mathtt{p(s^{}}^{a+1}\mathtt{o,s^{}}^{b}\mathtt{o,s^{}}^{c+1}\mathtt{o)} \in M_{\mathbf{sum}}$.

Minimaalisuudesta: muuta ei mallissa $M_{\mathbf{sum}}$ ole.

Osoitetaan $M_{\mathbf{sum}} = \left\{ \mathtt{p(s}^a\mathtt{o,s}^b\mathtt{o,s}^{a+b}\mathtt{o)} : a, b \in \mathbb{N} \right\}$:

"$\supseteq$" : Olkoon $\mathtt{p(s}^a\mathtt{o,s}^b\mathtt{o,s}^{a+b}\mathtt{o)}$ mielivaltainen. Näytetään induktiolla että $\in M_{\mathbf{sum}}$:

$a = 0$: Seuraa suoraan faktasta.

$a > 0$: Seuraa induktio-oletuksesta

$$\mathtt{p(s}^{a-1}\mathtt{o,s}^b\mathtt{o,s}^{a-1+b}\mathtt{o)} \in M_{\mathbf{sum}},$$

ja siihen sovelletusta säännöstä.

"$\subseteq$" : Olkoon $\rho = \mathtt{p(s}^a\mathtt{o,s}^b\mathtt{o,s}^c\mathtt{o)} \in M_{\mathbf{sum}}$ mielivaltainen. Näytetään induktiolla suureen a suhteen, että $c = a + b$.

Miten ρ ilmestyi joukkoon $M_{\mathbf{sum}}$?

- Jos käytettiin faktaa, niin $a = 0$ ja $b = c$, ja siis $c = a + b$.
- Jos käytettiin sääntöä, niin sen vartalolle

$$\mathtt{p(s}^{a-1}\mathtt{o,s}^b\mathtt{o,s}^{c-1}\mathtt{o)} \in M_{\mathbf{sum}},$$

ja siitä induktio-oletuksella $c - 1 = a - 1 + b$ eli $c = a + b$.

Minimaalisuuden nojalla *muita tapoja ei ole*.

IV.3.3 Todistussemantiikka

Todistusmenetelmän pitää olla **oikeellinen**: sillä voi todistaa *vain varmoja asioita*.

Varmat asiat ovat ne, jotka ovat *tosia kaikissa malleissa*.

Kalvojen IV.3.2 mukaan Prolog-ohjelmalla L on *pienin* malli M_L joka on samalla sen merkitys.

Kysymykseen "Onko maalifakta $p \in M_L$?" – eli kalvojen IV.3.1 aliohjelmakutsuun p – voi siis yrittää vastata *todistamalla* p väitteistä L.

Valitaan *täydellistä todistussääntö* eli sellainen jolla pystyy todistamaan kaiken sen, mikä on varmaa.

Kalvojen IV.3.2 *todistuksen etsinnän* epätäydellisyyteen se ei auta: vaikka sääntö itse onkin täydellinen, algoritmi joka sitä käyttää ei edelleenkään ole.

Prolog-ohjelmien todistussäännön perusaskeleena käytetään *resoluutiota*. Sen muuttujaton muoto on

looginen ehto 1: $q_1 \wedge \ldots \wedge q_m \rightarrow p_1 \vee \ldots \vee p_n$

looginen ehto 2: $q'_1 \wedge \ldots \wedge q'_{m'} \rightarrow p'_1 \vee \ldots \vee p'_{n'}$

sivuehto: p_1 ja q'_1 ovat samat

johtopäätös:

$$q_1 \wedge \ldots \wedge q_m \wedge q'_2 \wedge \ldots \wedge q'_{m'} \rightarrow p_2 \vee \ldots \vee p_n \vee p'_1 \vee \ldots \vee p'_{n'}$$

missä kaikki p:t ja q:t ovat faktoja.

"Jos yhden ehdon oikealla puolella on sama kuin toisen vasemmalla, niin ne voidaan pyyhkiä pois ja loput yhdistää uudeksi ehdoksi."

Muuttujat mukaan resoluutiosääntöön:

- Kummankin loogisen ehdon edessä on "kaikilla"-kvanttori $\forall \vec{X}$, $\forall \vec{X}'$ jokaiselle ehdon muuttujalle, kuten kalvoilla IV.3.2.

- Nimetään loogisten ehtojen muuttujat uudelleen siten, ettei sama nimi esiinny molemmissa: $\forall \vec{Y}$, $\forall \vec{Y}'$.

- Luetaan sivuehto " ... samastuvat yleisimmällä samastimella θ".

- Johtopäätökseen sovelletaan tätä samastinta θ.

- Johtopäätöksen edessä on kvanttori $\forall \vec{Z}$ jokaiselle siihen jääneelle muuttujalle.

Miten nämä implikaatiot liittyvät Prologiin?

- Kalvoilla IV.3.2 sääntö

$$p\text{:-}q_1\text{, } \ldots \text{ ,}q_m\text{.}$$

  tulkittiin loogisesti

$$\forall \vec{X}.q_1 \wedge \ldots \wedge q_m \rightarrow p.$$

- Koska tyhjä konjunktio ("ja") on **tosi**, niin puuttuva vartalo (eli $m = 0$) eli fakta

$$p\text{.}$$

  on

$$\forall \vec{X}.\textbf{tosi} \rightarrow p$$

  eli

$$\forall \vec{X}.p.$$

- Koska tyhjä disjunktio (”tai”) on **epätosi**, niin puuttuva pää on

$$\forall \vec{X}.q_1 \wedge \ldots \wedge q_m \rightarrow \textbf{epätosi}$$

eli

$$\forall \vec{X}.\neg(q_1 \wedge \ldots \wedge q_m)$$

(missä ’$\neg$’ on negaatio eli ”ei”), eli

$$\neg\exists \vec{X}.q_1 \wedge \ldots \wedge q_m$$

(missä ’$\exists$’ on ”on olemassa” -kvanttori).

”Ei ole olemassa sellaisia arvoja muuttujille $\vec{X}$, joilla ongelmat $q_1, \ldots, q_m$ ratkeaisivat yhtä aikaa.”

Tämä on *maali*

?-q_1**,** $\ldots$ **,**q_m**.**

jota yritetään *kumota* vastauksella ”onpas; esimerkiksi arvot ... ”.

Resoluutio onkin *refutaatio*- eli ristiriitatäydellinen päättelysääntö: jos säännöt ja maali eivät voi olla yhtä aikaa totta, niin sen voi osoittaa resoluutioaskeleilla.

Prologissa resoluutioaskel saa siis muodon

sääntö: p:-q_1, ... ,q_m.

maali: ?-q'_1, ... ,$q'_{m'}$.

sivuehto: säännön päällä p ja ensimmäisellä alimaalilla q'_1 on yleisin samastin θ.

Ensin säännön muuttujat on kuitenkin nimetty erilleen maalin muuttujista.

uusi maali: ?-$q_1\theta$, ... ,$q_m\theta$,$q'_2\theta$, ... ,$q'_{m'}\theta$.

Maali toimii siis (ensimmäistä) ratkaisuaan odottavien alimaalien *pinona*.

Kun saavutetaan *tyhjä* maali ?-., on löydetty ristiriita **tosi** $\rightarrow$ **epätosi**. Vastaesimerkkinä kerrotaan todistuksen kuluessa kertynyt sidonta. Kun vielä päätetään

- kokeilla sääntöjä niiden kirjoitusjärjestyksessä
- peruuttaa edelliseen maaliin ja sen seuraavaan sääntöön kun nykyinen maali osoittautuu mahdottomaksi

päästään kalvojen IV.3.1 suorituskoneeseen.

```
pajakari$ /opt/pl/bin/pl
Welcome to SWI-Prolog (Version 4.0.1)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('sum.pl').
% sum.pl compiled 0.00 sec, 1,112 bytes

Yes
?- trace,p(X,Y,s(s(s(o)))).
   Call: (7) p(_G282, _G283, s(s(s(o)))) ? creep
   Exit: (7) p(o, s(s(s(o))), s(s(s(o)))) ? creep

X = o
Y = s(s(s(o))) ;
   Redo: (7) p(_G282, _G283, s(s(s(o)))) ? creep
   Call: (8) p(_G378, _G283, s(s(o))) ? creep
   Exit: (8) p(o, s(s(o)), s(s(o))) ? creep
   Exit: (7) p(s(o), s(s(o)), s(s(s(o)))) ? creep

X = s(o)
Y = s(s(o)) ;
   Redo: (8) p(_G378, _G283, s(s(o))) ? creep
   Call: (9) p(_G380, _G283, s(o)) ? creep
   Exit: (9) p(o, s(o), s(o)) ? creep
   Exit: (8) p(s(o), s(o), s(s(o))) ? creep
   Exit: (7) p(s(s(o)), s(o), s(s(s(o)))) ? creep

X = s(s(o))
Y = s(o) ;
   Redo: (9) p(_G380, _G283, s(o)) ? creep
   Call: (10) p(_G382, _G283, o) ? creep
   Exit: (10) p(o, o, o) ? creep
   Exit: (9) p(s(o), o, s(o)) ? creep
   Exit: (8) p(s(s(o)), o, s(s(o))) ? creep
   Exit: (7) p(s(s(s(o))), o, s(s(s(o)))) ? creep

X = s(s(s(o)))
Y = o ;
   Redo: (10) p(_G382, _G283, o) ? creep
   Fail: (10) p(_G382, _G283, o) ? creep
   Fail: (9) p(_G380, _G283, s(o)) ? creep
   Fail: (8) p(_G378, _G283, s(s(o))) ? creep
   Fail: (7) p(_G282, _G283, s(s(s(o)))) ? creep

No
[debug]  ?-
```

Edellä todistussemantiikka kuvattiin samassa järjestyksessä kuin kalvojen IV.3.1 suoritussemantiikassa, jossa sääntöjä käytetään "takaperin" eli **niin**-osasta **jos**-osaan.

Kalvojen IV.3.2 mallisemantiikassa sääntöjä käytettiinkin "etuperin". Vastaava todistussemantiikka saadaan lukemalla Prolog-lause

$$p(t_{(i,1)}, \ldots, t_{(i,k)}) \text{ :- } q_{(i,1)}, \ldots, q_{(i,m_i)}. \text{ eli}$$
$$\forall \vec{X}. q_{(i,1)} \wedge \ldots \wedge q_{(i,n)} \rightarrow p(t_{(i,1)}, \ldots, t_{(i,k)})$$

(ei enää totuudesta puhuvana väitteenä vaan) todistuvuudesta puhuvana päättelysääntönä

$$\frac{q_{(i,1)} \quad \cdots \quad q_{(i,m_1)}}{p(t_{(i,1)}, \ldots, t_{(i,k)})}$$

joka luetaan "jos olet saanut todistuksen K_i jokaiselle oletukselle $q_{(i,j)}$, niin voit yhdistää ne todistukseksi K johtopäätökselle $p(t_{(i,1)}, \ldots, t_{(i,k)})$".

Muuttujat tulkitaan kalvojen IV.3.1 hahmojen tavoin:

- Jokainen K_i todistaa jokin muuttujattoman muodon q_i' omasta johtopäätöksestään $q_{(i,j)}$.
- Näiden q_i' on yhdessä sovittava säännön hahmoon.
- Sovittamisella määräytyy johtopäätöksen todistettu muuttujaton (?) muoto.

Tässä ohjelman luennassa päättelysääntöinä kalvojen IV.3.1 suoritussemantiikka voidaan selittää seuraavasti:

- Todistusta etsitäänkin "alhaalta ylöspäin" eli johtopäätöksistä oletuksiin.

- Etsinnän kuluessa muuttujia sidotaan samastuksella väin sen verran kuin on välttämätöntä kokeiltavan säännön käyttämiseksi.

- Sääntöjä kokeillaan kirjoitusjärjestyksessä $i = 1, \ldots, n$.

- Oletukset yritetään todistaa kirjoitusjärjestyksessä $j = 1, \ldots, m_i$.

Entä mikä on tässä luennassa sellainen looginen päättelysäännöstö, jossa annetun väitteen todistusta etsittäessä ei koskaan ole kuin olennaisesti yksi mahdollisuus?

Sehän on ohjelmointikielen suoritusmekanismin abstrakti määrittely "vaihtoehdottomana logiikkana"!

Vastaava malliteoria on kielen denotationaalinen eli tarkoitesemantiikka.

Esimerkkinä tästä SAT-ratkoja:

```
/*
onko(Kaava,Totuusarvo) on totta kun

1. Totuusarvo on joko 'tosi' tai 'vale'.

2. Lauselogiikan Kaava on esitetty seuraavasti:

   a. Totuusarvoiset muuttujat X_i on esitetty terminä x(V_i)
      missä V_i on oma Prolog-muuttuja kullekin i.

   b. Konjunktiot 'F/\G' on esitetty terminä 'ja(F,G)'.

   c. Konjunktiot 'F\/G' on esitetty terminä 'tai(F,G)'.

   d. Negaatiot '~F' on esitetty terminä 'ei(F)'.

   Esimerkiksi kaava '(A\/B)/\~A' esitetään terminä
   'ja(tai(x(A),x(B)),ei(x(A)))'.

3. Kaava saa Totuusarvon muuttujien V_i nykyisillä sidonnoilla.

*/

onko(x(T),T).

onko(ja(F,G),tosi):-
        onko(F,tosi),
        onko(G,tosi).
onko(ja(F,G),vale):-
        onko(tai(ei(F),ei(G)),tosi).

onko(tai(F,_),tosi):-
        onko(F,tosi).
onko(tai(_,G),tosi):-
        onko(G,tosi).
onko(tai(F,G),vale):-
        onko(ja(ei(F),ei(G)),tosi).

onko(ei(F),tosi):-
        onko(F,vale).
onko(ei(F),vale):-
        onko(F,tosi).
```

Esimerkkinä Prologin käytöstä perinteisessä ohjelmoinnissa olkoon lisäys 2-3-puuhun [AHU83,§5.4;B01,§10.1].

2-3-puu on (haku)puu, jossa

- sisäsolmuilla on 2 tai 3 alipuuta, ja
- talletettu tieto on lehdissä ...
- ... jotka ovat samalla tasolla.

(Listaus jatkuu yli kalvorajojen.)

```
/*

* e/0: Tyhjä 2-3-puu.

* l/1: Lehti jossa alkio (=luku).

* b/3: 2-haarainen sisäsolmu, välissä
       oikeanpuoleisen haaran pienin alkio.

* t/5: 3-haarainen sisäsolmu, väleissä
       oikeanpuoleisten haarojen pienimmät alkiot.

* c/3: Kuten b/3 mutta korkeus kasvanut yhdellä.
       Algoritmi pyrkii tasoittamaan syntyneen eron.

*/

% Pääpredikaatti:
% Kun luku X lisätään puuhun T0 niin saadaan puu T.
add23(T0,X,T):-
        ins23(T0,X,T1),
        put23(T1,T).

% Jos vielä lisäyksen lopussakin on korkeus kasvanut,
```

```
% niin koko puuta on kasvatettava uudella juurella:
put23(c(T0,X,TX),
      b(T0,X,TX)).
put23(l(X),
      l(X)).
put23(b(T0,X,TX),
      b(T0,X,TX)).
put23(t(T0,X,TX,Y,TY),
      t(T0,X,TX,Y,TY)).

% Tyhjään puuhun lisäys on lehti.
ins23(e,X,l(X)).

% Lehteen lisäys on korkeutta kasvanut taimi.
ins23(l(Y),X,c(l(Y),X,l(X))):-
        Y<X.
ins23(l(Y),X,c(l(X),Y,l(Y))):-
        X<Y.

% 2-haaraiseen sisäsolmuun lisäys:
% Valitse alipuu, lisää uusi alkio siihen, ja
% pistä kasvanut alipuu takaisin paikoilleen.
ins23(b(T0,Y,TY),X,T):-
        X<Y,
        ins23(T0,X,TX),
        put2l(TX,Y,TY,T).
ins23(b(T0,Y,TY),X,T):-
        Y<X,
        ins23(TY,X,TX),
        put2r(T0,Y,TX,T).

% 3-haaraiseen sisäsolmuun lisäys:
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        X<Y,
        ins23(T0,X,TX),
        put3l(TX,Y,TY,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Y<X,
        X<Z,
        ins23(TY,X,TX),
        put3m(T0,Y,TX,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Z<X,
        ins23(TZ,X,TX),
        put3r(T0,Y,TY,Z,TX,T).

% Vaikka 2-haaraisen alipuu olisikin kasvanut korkeutta,
% ei hätää: tehdään sen tilalle 3-haarainen solmu!
put2l(c(T0,X,TX),Y,TY,
```

```
    t(  T0,X,TX, Y,TY)).
put2l(b(T0,X,TX),Y,TY,
    b(b(T0,X,TX),Y,TY)).
put2l(t(T0,X,TX,Y,TY),Z,TZ,
    b(t(T0,X,TX,Y,TY),Z,TZ)).

put2r(T0,X,c(TX,Y,TY),
    t(T0,X,  TX,Y,TY)).
put2r(T0,X,b(TX,Y,TY),
    b(T0,X,b(TX,Y,TY))).
put2r(T0,X,t(TX,Y,TY,Z,TZ),
    b(T0,X,t(TX,Y,TY,Z,TZ))).

% Jos 3-haaraisen alipuu on kasvanut korkeutta,
% on hätä: korjaus on tehtävä lähempänä juurta!
put3l(c(T0,X,TX),Y,  TY,Z,TZ,
    c(b(T0,X,TX),Y,b(TY,Z,TZ))).
put3l(b(T0,X,TX),Y,TY,Z,ZY,
    t(b(T0,X,TX),Y,TY,Z,ZY)).
put3l(t(T0,W,TW,X,TX),Y,TY,Z,ZY,
    t(t(T0,W,TW,X,TX),Y,TY,Z,ZY)).

put3m(T0,X,c(TX, Y,  TY),Z,TZ,
  c(b(T0,X,  TX),Y,b(TY, Z,TZ))).
put3m(T0,X,b(TX,Y,TY),Z,TZ,
    t(T0,X,b(TX,Y,TY),Z,TZ)).
put3m(T0,W,t(TW,X,TX,Y,TY),Z,TZ,
    t(T0,W,t(TW,X,TX,Y,TY),Z,TZ)).

put3r(T0,X,TX, Y,c(TY,Z,TZ),
  c(b(T0,X,TX),Y,b(TY,Z,TZ))).
put3r(T0,X,TX,Y,b(TY,Z,TZ),
    t(T0,X,TX,Y,b(TY,Z,TZ))).
put3r(T0,W,TX,X,t(TX,Y,TY,Z,TZ),
    t(T0,W,TX,X,t(TX,Y,TY,Z,TZ))).

% Lisärutiineja puiden luomiseen ja katseluun:
build23([],e).
build23([X|Xs],T):-
        build23(Xs,T1),
        add23(T1,X,T).

show(T):-
        show(T,0).
show(e,_).
show(l(X),H):-
        view(X,H).
show(b(T0,X,TX),H):-
        H1 is H+5,
```

```
        view(-----,H),
        show(TX,H1),
        view(X,H),
        show(T0,H1),
        view(-----,H).
show(t(T0,X,TX,Y,TY),H):-
        H1 is H+5,
        view(-----,H),
        show(TY,H1),
        view(Y,H),
        show(TX,H1),
        view(X,H),
        show(T0,H1),
        view(-----,H).
view(A,H):-
        tab(H),
        write(A),
        nl.
```

(Listaus jatkui yli kalvorajojen.)

- Ohjelma haarautuu tapauksiin parametrihahmojen mukaan.

- Tietorakenneoperaatiot esitetään suoraan näillä hahmoilla.

- Hahmonsovitus on otettu myös uusiin funktionaalisiin kieliin kuten Haskell [T99] ja ML [CM98].

IV.4 Kirjastopalveluita

Tutustutaan Prolog-standardin peruspalveluihin listankäsittelyyn aritmetiikkaan ja samastukseen [B01,§2.7,3.1-3.2,3.4,7.2-7.3]. Toteutukset tarjoavat usein enemmänkin, varsinkin listoille.

IV.4.1 Listat

Myös Prolog tarjoaa kalvojen II.8 kaltaiset listat.

Prolog kirjoittaa listansa *alkiot pilkuilla eroteltuina hakasulkeisiin*: `[tämä,on,listavakio]`.

Tyhjä lista on siis `[]`.

Erillistä lainausmerkkimekanismia ei nyt tarvita, koska listan erottaa muista termeistä hakasulkeiden perusteella.

Listoja voi käsitellä kalvojen IV.3.1 *listahahmoilla*:

$$\texttt{[}A_1\texttt{, ... ,}A_k\texttt{|}D\texttt{]}$$

on hahmo sellaiselle listalle, jossa on ainakin k alkiota.

Jos loppuhahmo '|D' puuttuu, on kyseessä hahmo tasan k-alkioiselle listalle (eli loppuhahmoksi otetaan '`|[]`').

Hahmossa kukin A_i ja D ovat Prolog-termejä.

Listan i. alkiota yritetään samastaa termin A_i kanssa, ja loppulistaa termin D kanssa:

```
?- [tämä,on,lista]=[A1,A2|D].

A1 = tämä
A2 = on
D = [lista]
```

Näillä hahmoilla voidaan purkaa listoja kuten Scheme-kirjastofunktioilla `cd`$^{k-1}$`ar` ja `cd`^{k}`r`.

Niillä voidaan myös *rakentaa* listoja lukemalla ne toisin päin: muodosta uusi lista lisäämällä listan D alkuun alkiot $A_k, \ldots, A_1$.

Esimerkiksi kahden listan liittäminen kolmanneksi on:

```
conc([],Y,Y).
conc([X|Xs],Y,[X|Z]):-
        conc(Xs,Y,Z).
```

- Jos ensimmäinen lista on tyhjä, niin toinen ja kolmas lista ovat samat.

- Jos ensimmäisellä listalla on ensimmäinen alkio `X`, niin kolmas lista alkaa tällä alkiolla ja jatkuu listana `Z`, joka puolestaan on ensimmäisen loppulistan `Xs` ja koko toisen listan `Y` liitos.

Tuloksella voi paitsi liittää myös *jakaa* listoja:

```
?- conc([1],[2],Z).

Z = [1, 2] ;

No
?- conc(X,Y,[1,2]).

X = []
Y = [1, 2] ;

X = [1]
Y = [2] ;

X = [1, 2]
Y = [] ;

No
?-
```

(Eikä ihme, sehän on kalvojen IV.2 `p/3` listoille ...)

IV.4.2 Aritmetiikka

Kalvoilla I.3 nähtiin Prolog-ohjelma (yksinkertaiseen) derivointiin.

Sen sääntöjen päissä käytettiin aritmeettisia operaattoreita `+`, `*`, `/` ja `**` (potenssiin korotus) *symbolisina* kaksipaikkaisina funktoreina (jotka kirjoitettiin argumenttiensa väliin matematiikan tavalliseen tapaan).

Niiden avulla ohjelma haarautui ensimmäisen argumentin hahmon perusteella oikeaan sääntöön, ja tuotti toiseen argumenttiin vastaavan derivaattatermin vartalosta saatujen dervivaattatermien avulla.

Mutta ohjelman toisen säännön vartalossa eksponenttia `N` käytettiinkin *numerona*, josta vähennettiin `1`, jotta tuloksena saataisiin uusi eksponentti `N1`.

Tämän sai aikaan (myös väliin kirjoitettu) kirjastopredikaatti V `is` E joka on totta kun (i) E on nykyisellä sidonnalla sellainen aritmeettinen lauseke jonka arvo voidaan laskea ja (ii) V sopii tuohon lakettuun arvoon.

Jos (i) ei pädekään, seuraa ajonaikainen virhe.

Kalvojen I.3 ohjelmassa kutsu `N1 is N-1` on siis:

1. Tarkista, että lauseke `N-1` on nykyisessä sidonnassa aritmeettinen:

   - Ei vapaita muuttujia.
   - Vain aritmeettisia funktoreita (kuten +/2, */2, //2, **/2, ...).
   - Vain numerovakioita.

   Silloin lausekkeella on numeerinen arvo a.

   Muuten kyseessä on ajonaikainen virhe.

2. Yritä sitoa `N1` arvoon a.

Prolog-kielessä on äärellisen tarkkuuden kokonais- ja liukuluvut. Liian suurista kokonaisluvuista tulee automaattisesti liukulukuja, liian suurista liukuluvuista ääretön atomi `$Infinity`.

Myös (jälleen väliin kirjoitettavat) aritmeettiset vertailut =:=/2 ($=$), =\=/2 ($\neq$), >/2 ($>$), >=/2 ($\geq$), </2 ($<$) ja =</2 ($\leq$) laskevat argumenttiensa numeroarvot samalla tavalla.

Maali `?-1<X,X<3.` antaa siis virheen.
(Logiikkaohjelmointi voidaan tosin laajentaa tällaisiin *reaalilukurajoitteisiin* [B01,§14.2].)

IV.4.3 Samastus

Kalvoilla III.2 esiteltiin keskeinen samastusoperaatio `unify`, jolla sääntöjärjestelmä teki annetuista termeistä samoja sitomalla niiden muuttujia.

Samastusalgoritmin haara `uvc` eli "vapaa muuttuja x vastaan rakenteinen termi t" käytti testiä `binding:occurs?` varmistamaan, ettei x esiinny termissä t. Muutenhan sidonnan $x = t$ lisääminen aiheuttaisi kehän.

Tämä *esiintymistarkistus* (occurs check)

- hidastaa samastusta selvästi
- on harvoin tarpeen käytännön ohjelmoinnissa

joten Prologissa se *jätetään pois – vaikka silloin kalvojen IV.3 puhdas rakenne likaantuu!*

Puhtaan samastuksen voi tehdä kirjastopredikaatilla `unify_with_occurs_check`/2.

Myös Prologin omaa samastusta voi kutsua nimellä:

- =/2 luetaan "samastuvat" (eli "ovat samat"). (On myös ==/2 eli "taysin samat ilman samastustakin".)

- \=/2 luetaan "eivät samastu". (On myös \==/2.)

Esiintymistarkistuksen puuttuminen tekee näistä hieman arvaamattomia: ne saattavat esimerkiksi päätyä ikuiseen silmukkaan.

Prolog tarjoaa myös *nimettömän muuttujan* _ (eli pelkän alaviivan) joka samastuu aina mutta arvoa ei muistella. Esimerkiksi `f(a,_,_)` samastuu jokaiseen `f(a,`t_1`,`t_2`)` vaikka t_1 ja t_2 eivät samastuisikaan keskenään.

Samastuksen kuluessa rakenteisia termejä puretaan osiinsa. Sama palvelu tarjotaan (väliin kirjoitettavalla) kirjastopredikaatilla =../2 (nimeltään "univ"). Se on seuraava syntaktinen relaatio:

$$f(t_1, \ldots, t_k) \texttt{ =.. } [f, t_1, \ldots, t_k]$$

(Myös $k = 0$ käy.) Sitä voidaan käyttää myös "oikealta vasemmalle" eli kokoamaan termi annetusta listasta.

IV.5 Negaatio

Kalvojen IV.3 puhtaassa Prologissa ei voi ilmaista *negatiivista* tietämystä " ... *ei* ole ... ".

- Olisi yksinkertaista kirjoittaa faktoja "henkilö X sairastaa tautia Y" ja

- määritellä "henkilö Z on terve jos hän ei sairasta mitään tautia".

Tämän vuoksi Prologiin on lisätty *eräänlainen* negaatio [B01,§5.3-5.4]: vartalossa voi esiintyä alimaali

$$\texttt{not}(p)$$

jonka haluttu luenta on "väite p ei päde".

Seuraavaksi tarkastellaan mikä on sen todellinen luenta.

Useissa toteutuksissa sulut voidaan jättää pois.

Useissa toteutuksissa nimi onkin \+ korostamaan ettei kyseessä olekaan "oikea" negaatio.

IV.5.1 Suoritussemantiikassa

Kalvojen IV.3.1 suoritussemantiikassa kutsu `not`(p) tehdään seuraavasti:

1. Yritetään suorittaa negaation sisällä oleva kutsu p nykyisellä sidonnalla θ_i.

2. Jos sisäkutsu p epäonnistuu, niin koko kutsu `not`(p) *onnistuu*.

   Sidontana pysyy θ_i.

   (Eihän epäonnistunut p voinut kasvattaa sidontaa θ_i. Myöskään myöhempi peruutus takaisin kutsuun `not`(p) ei tuota uusia vaihtoehtoja.)

3. Muuten koko kutsu *epäonnistui* (ja peruutetaan hakemaan vaihtoehtoista θ_i).

Tämän suoritustavan nimi on *negaatio epäonnistumisena* (Negation as Failure).

Sairausesimerkkimme on nyt käytännössä:

```
sairastaa(leopold_duke_of_albany,hemofilia).
sairastaa(rupert,hemofilia).
sairastaa(frederick,hemofilia).
sairastaa(alexis,hemofilia).
sairastaa(waldemar,hemofilia).
sairastaa(henry,hemofilia).
sairastaa(leopold,hemofilia).
sairastaa(maurice,hemofilia).
sairastaa(alfonso,hemofilia).
sairastaa(gonzalo,hemofilia).
sairastaa(luennoija,kevätflunssa).

terve(X):-not(sairastaa(X,_)).

?- trace,terve(luennoija).
   Call: (7) terve(luennoija) ? creep
^  Call: (8) not(sairastaa(luennoija, _G319)) ? creep
   Call: (9) sairastaa(luennoija, _G319) ? creep
   Exit: (9) sairastaa(luennoija, kevätflunssa) ? creep
^  Fail: (8) not(sairastaa(luennoija, _G319)) ? creep
   Fail: (7) terve(luennoija) ? creep

No

?- trace,terve(teräsmies).
   Call: (7) terve(teräsmies) ? creep
^  Call: (8) not(sairastaa(teräsmies, _G313)) ? creep
   Call: (9) sairastaa(teräsmies, _G313) ? creep
   Fail: (9) sairastaa(teräsmies, _G313) ? creep
^  Exit: (8) not(sairastaa(teräsmies, _G313)) ? creep
   Exit: (7) terve(teräsmies) ? creep

Yes
```

Tähän saakka vastaukset olivat järkeviä, mutta ...

```
?- trace,terve(X).
   Call: (8) terve(_G225) ? creep
^  Call: (9) not(sairastaa(_G225, _G284)) ? creep
   Call: (10) sairastaa(_G225, _G284) ? creep
   Exit: (10) sairastaa(leopold_duke_of_albany, hemofilia) ? creep
^  Fail: (9) not(sairastaa(_G225, _G284)) ? creep
   Fail: (8) terve(_G225) ? creep

No
```

Kalvoilla IV.3.3 Prolog-maali `?-` q`.` luettiin "onko olemassa sellaisia vapaiden muuttujien arvoja, joilla q pätee (ja jos on niin mitkä ne ovat)?"

Tarkoitimme maalillamme `?- terve(X).` siis "hae kaikki terveet ihmiset `X`".

Saimme vastauksen `No` eli "heitä ei ole":

1. Normaali suoritus johti ensin uuteen maaliin `?- not(sairastaa(X,_)).`

2. Valittu negaation suoritustapa teki siitä maalin `?- sairastaa(X,_).`

3. Tämän maalin `Yes/No`-vastaus käännetään nurin.

   Mutta tämä maalihan luetaankin "sairastaako joku jotakin?"

   Siis alkuperäinen maali kysyykin "ovathan kaikki terveitä?"

Negaatioon epäonnistumisena ei voi luottaa jos negatoidussa alimaalissa on kutsuhetkellä vapaita muuttujia!

Maalin implisiittinen $\exists$`X.terve(X)` vaihtuu näet negaation ajaksi muotoon $\forall$`X,_.`$\neg$`sairastaa(X,_)`.

IV.5.2 Mallisemantiikassa

Kalvojen IV.5.1 negatoitujen muuttujien ongelmaa voi yrittää selittää kalvojen IV.3.2 mallisemantiikalla.

Positiivinen muuttujaton maali `?-` p`.` vastaa `Yes/No`-kysymykseen "$p \in M_L$".

Negatiivinen muuttujaton `?- not(`q`).` vastaa kysymykseen "$q \notin M_L$".

Tämän toteuttu negaatio kyllä osaa.

Positiivinen muuttujallinen p on hahmo joka edustaa kaikkia niitä muuttujattomia p' joilla vastaus on `Yes`.

Negatiivinen muuttujallinen q edustaa kaikkia q' joilla vastaus on `No`.

Tätä toteutettu negaatio ei osaa: se tarkistaa vain onko sellaisia olemassa.

HUOM! Tämä selittää vain käyttäjän *kyselyihin* kirjoittamat negaatiot. *Ohjelmaan* kirjoitetut negaatiot saattavat (rekursioon osallistuessaan) jopa tuhota pienimmän mallin M_L yksikäsitteisyyden!

IV.5.3 Todistussemantiikassa

Kalvojen IV.3.3 todistussemantiikassa kalvoilla IV.5.1 valittu negaation toteutus on uusi päättelysääntö

> Väite `not`(p) voidaan todistaa osoittamalla ettei väitettä p voi todistaa.

Tämä on ns. *suljetun maailman oletus* (Closed World Assumption, CWA). Sitä käytetään paljon tekoälyssä.

Koska todistaa sai vain sellaista mikä on varmaa on tämä oletus (täydellisessä päättelyjärjestelmässä)

> ”koska p ei ole varmaa niin se ei ole totta”.

Ohjelman ”maailma(nkuva)” sulkee siis pois *epävarmuuden*: jos ohjelmalla ei ole tarpeeksi tietoa väitteestä p, niin se ei ohjelman mielestä voi pitää paikkaansa.

Tietojenkäsittelyllisesti perusteltavissa: ohjelman on reagoitava oman tietämyksensä pohjalta vaikkei se tiedäkään kaikkea ulkomaailmasta.

Tämä on ns. *epämonotoninen* päättelysääntö: uuden tiedon lisääminen saattaa *kumota* aiempia päätelmiä.

IV.6 Vaihtoehtojen karsinta

Kalvojen IV.5 Prolog-negaatio `not(`p`)` on toteutettava sellaisella ohjelmalla, joka epäonnistuu jos p onnistuu [B01,§5.1-5.2,7.5]:

```
not(P):-call(P),!,fail.
not(_).
```

- Muuttujan `P` arvo voidaan tulkita predikaatin kutsuksi kirjastopredikaatilla `call/1`; vrt. Scheme-kirjastofunktio `eval` kalvoilla II.9.2.

  (Kutsuksi `call(P)` käy usein myös pelkkä `P`.)

- (Kirjasto)predikaatti `fail/0` epäonnistuu aina – sillä ei ole määritelmää.

- Kirjastopredikaatti `!/0` on ns. *katkaisu* (cut).

  Se luetaan intuitiivisesti "jos olet tätä kutsua suorittaessasi päässyt tähän saakka, niin olet löytänyt ainoan mahdollisen suoritustavan; unohda kaikki muut".

  Tässä luenta on "jos olet onnistunut kutsussa `P`, niin ainoa mahdollisuus on `fail`; unohda fakta".

Olkoon tarkemmin sanoen ratkaistavana maali

?-g, ...

ja kokeiluvuorossa ohjelman sääntö

p:-q_1, ... ,q_m,!,r_1, ... ,r_n.

joka vaikuttaa sopivalta katkaisuun saakka, eli on saatu sidonta θ_m (samastamalla g ja p sidonnaksi θ_0, laajentamalla se kutsulla q_1 sidonnaksi θ_1, ...).

- Suoritus etenee normaalisti eteenpäin yli katkaisukohdan vartalon loppuosaan r_1, ... ,r_n. sidonnan θ_m muuttumatta.

- Sivuvaikutuksena etsintäkone hävittää kaikki kutsun g jäljellä olleet vaihtoehtoiset ratkaisutavat:

  – vartalon alkuosan q_1, ... ,q_m

  – loput ohjelmasta.

- Jos suoritus myöhemmin peruuttaa taaksepäin yli katkaisukohdan, niin koko nykyinen maali epäonnistuu heti; sen ensimmäistä kutsua g ei voida enää ratkaista toisin.

  Suoritus jatkuu eteenpäin edellisestä maalista, eli siitä jossa g ei vielä ollut ensimmäisenä.

Kalvojen IV.5.1 negaation toteutus on esimerkki säännöstä (uutta) muotoa

jos ... **niin** ... **muuten** ...

"**Jos** `P` onnistuu **niin** epäonnistu **muuten** onnistu."

\+ Katkaisu laajentaa Prolog-kielen ilmaisuvoimaa.

\+ Katkaisujen lisääminen parantaa Prolog-ohjelman tehokkuutta koska turha peruutus jää pois.

\− Katkaisun käyttö tekee Prolog-ohjelmasta (lopullisesti) sellaisen jonka sääntöjä on seurattava kirjoitusjärjestyksessä.

Katkaisu on

vihreä jos se tehostaa ohjelmaa muuttamatta sen merkitystä (= löytämiä vastauksia)

punainen jos ohjelman merkitys muuttuu.

Punaisia katkaisuja tulee käyttää varoen, koska deklaratiiviselta näyttävään ohjelmaan on nyt ilmaantunut proseduraalisia riippuvuuksia!

Kalvojen IV.3.3 lopun 2-3-puuesimerkkiin voidaan lisätä vihreitä katkaisuja koska sen säännöt ovat (vertailujen nojalla) toisensa poissulkevia:

```
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        X<Y,
        !,
        ins23(T0,X,TX),
        put3l(TX,Y,TY,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Y<X,
        X<Z,
        !,
        ins23(TY,X,TX),
        put3m(T0,Y,TX,Z,TZ,T).
ins23(t(T0,Y,TY,Z,TZ),X,T):-
        Z<X,
        ins23(TZ,X,TX),
        put3r(T0,Y,TY,Z,TX,T).
```

Mutta jos koodia optimoi edelleen poistamalla *toisesta* säännöstä (nyt tarpeettoman) vertailun `Y<X`, niin *ensimmäisen* säännön katkaisusta tuleekin punainen:

- sääntöjä ei voi enää järjestellä vapaasti uudelleen
- ensimmäisen säännön katkaisua ei saa enää poistaa.

Tässä punainen katkaisu ei voittane paljoakaan.

On myös perusteltuja punaisia katkaisuja: samassa esimerkissä säännöt

```
put2l(c(T0,X,TX),Y,TY,
    t(  T0,X,TX, Y,TY)).
put2l(b(T0,X,TX),Y,TY,
    b(b(T0,X,TX),Y,TY)).
put2l(t(T0,X,TX,Y,TY),Z,TZ,
    b(t(T0,X,TX,Y,TY),Z,TZ)).
```

sanovat "jos vasen alipuu on kasvanut korkeutta, niin tee asialle jotakin; muuten se käy sellaisenaan".

Tämä voidaan ilmaista luontevasti, lyhesti ja tehokkaasti

```
put2l(c(T0,X,TX),Y,TY,
    t(  T0,X,TX, Y,TY)):-
        !.
put2l(T,Y,TY,b(T,Y,TY)).
```

punaisella katkaisulla.

Punaisia katkaisuja käytettäessä on syytä noudattaa erityistä varovaisuutta ja huolellisuutta ...

IV.7 Prolog ja tietokannat

Kalvojen III yksinkertaisessa Prolog-tulkissa säännöt talletettiin alkeelliseen ”tietokantaan” – listaan kirjoitusjärjestyksessä.

Myös oikeaan Prolog-toteutukseen kuuluu *sisäinen tietokanta* jossa sääntöjä säilytetään – edelleen järjestyksessä siltä varalta että ohjelman merkitys riippuisi sääntöjen suoritusjärjestyksestä ...

Prologilla on myös yhteyksiä ”ulkoisten” *relaatiotietokantojen* moderneihin kyselykieliin.

IV.7.1 Sisäinen tietokanta

[B01,§7.4]

`asserta(C)` (onnistuu aina kasvattamatta sidontaa ja) lisää sisäiseen tietokantaan säännön `C` muiden saman predikaatin sääntöjen *eteen*.

(Jos `C` on vartalollinen, niin se on syntaktisista syistä kirjoitettava toisiin sulkuihin).

Peruutettaessa `asserta/1` ei anna uusia "vastauksia", mutta sen lisäämä `C` jää tietokantaan.

`assertz(C)` on kuin `asserta/1` paitsi että `C` lisätäänkin edellisten *perään*.

`assert(C)` on näistä kirjastopredikaateista se jota suositellaan käytettäväksi kun sisäisen tietokannan järjestyksellä ei ole väliä – esimerkiksi kun lisätään (muuttujaton) fakta eli talletetaan tietoalkioita eikä aliohjelmia.

(Yleensä `assert/1` on `assertz/1`.)

`retract(C)` poistaa sisäisestä tietokannasta ensimmäisen säännön (tai faktan) joka sopii hahmoon `C` ja laajentaa sidontaa siten kuin `C` sopii poistettuun sääntöön.

Peruutettaessa poistetaan myös toinen, kolmas, ... kunnes poistettavaa ei enää ole.

Esimerkiksi maali

```
?- retract((apu(X):-B)).
```

poistaa predikaatin `apu/1` ensimmäisen sellaisen säännön r jolla on vartalo sekä sitoo muuttujan `X` säännön r pään parametriin ja muuttujan `B` säännön r vartaloon.

Maali

```
?- retract(apu(Y)),fail.
```

taas poistaa peruutellen kaikki predikaatin `apu/1` vartalottomat faktat.

Prolog-standardissa (mutta ei välttämättä kaikissa toteutuksissa)

- `consult`oitu tai kirjastopredikaatti on *staattinen* eikä muokattavissa suoritusaikana ilman edeltävää kutsua `dynamic/1`.

  Intuitiivisesti, suorituskelpoinen ohjelma*koodi* pidetään sittenkin erillään vapaasti muokattavasta tietokanta*datasta*, kunnes erikseen julistetaan aikomus muokata jotakin osaa koodista.

- vaikka predikaatin määrittelyä muutettaisiinkin kesken sen suorituksen, jatkuu sen nykyinen suoritus loppuun sillä samalla määrittelyllä jolla alkoikin [DEC96,§10.2].

  Intuitiivisesti, tietokantapäivitykset *ensin laskevat* vanhan tietokannan sisällöstä sen, millainen sisältö uudelle tietokannalle halutaan, ja vasta *sitten vaihtavat* vanhan sisällön uuteen.

  (Sen epäloogisempi mutta usein käytetty vaihtoehto on, että jokainen yksittäinen `assertz/1` ja `retract/1` tehdään heti, jat sen tulos vaikuttaa jo saman päivityksen seuraaviin askeliin.)

IV.7.2 Datalog

Kalvojen IV.7.1 ajatusta datatietokannan pitämisestä erillään säännöistä voi jatkaa edelleen:

- Unohdetaan rakenteiset termit (kuten listat) eli tyydytään pelkkiin atomeihin.

- **Ekstensionaalinen** tietokanta koostuu pelkästään muuttujattomista – eli vain atomeja sisältävistä – faktoista.

  Kukin tällainen predikaatti vastaa tutun *relaatiomallin* [AHV95,§3] yhtä (muokattavaa) tietokantataulua.

  ```
  töissä(työntekijä,yritys)
  johtaa(johtaja,yritys)
  valvoo(konserniyritys,tytäryritys)
  ```

- **Intensionaalinen** tietokanta sisältää ne säännöt, joilla ekstensionaalista kysellään: kyselyt, näkymät, ...

  Säännöissä on eri predikaatit kuin tauluissa.

Päädytään kyselykieleen *datalog* (database Prolog) [AHV95,§D].

Intensionaaliseen kantaan voidaan kirjoittaa relaatiokyselyjä kuten "pomot ja alaiset":

```
pomo(P,A):-johtaa(P,Y),töissä(A,Y).
```

$$\texttt{pomo} = \pi_{1,3}\sigma_{2=4}(\texttt{johtaa} \times \texttt{töissä})$$

```
CREATE VIEW pomo AS
       SELECT J.johtaja,T.työntekijä
       FROM   johtaa J,
              töissä T,
       WHERE  J.yritys=T.yritys
```

- Sama muuttuja vartalon eri predikaatinkutsuissa vastaa (luonnollista) liitosta ($\sigma_{\ldots=\ldots}(\ldots \times \ldots)$).

- Vartalossa esiintyvän muuttujan esiintyminen myös päässä vastaa projektiota ($\pi_{\ldots}$).

  Jos taas (nimetön) muuttuja esiintyy päässä muttei vartalossa, on sille vaikea keksiä tietokantatulkintaa; siksi sellaisia ei nyt sallita.

Samaa "relationaalista" ajattelua voi käyttää usein myös Prolog-ohjelmissa: vartalon kukin predikaatinkutsu hakee vastaavasta (ehkä intensionaalisesta) relaatiosta seuraavan monikon, ja koko sääntö kertoo miten nämä vartalon monikot yhdistetään pään mukaisen relaation monikoksi.

Puhtaan relationaalinen tietokanta-ajattelu ei tosin aina ole mahdollista:

- Prologin suoritusjärjestys saattaa vaatia säännöille tai vartalon kutsuille jonkin tietyn järjestyksen, kun taas relaatioajattelussa järjestys on vapaa.

- Vastauksen kokoamisen ajatellaan tapahtuvan datalogissa kokonaisia tietojoukkoja käsittelemällä yksittäisten kutsujen ratkomisen sijaan.

Saman predikaatin eri säännöt vastaavat yhdistettä:

```
käläkättäjä(X):-töissä(X,nokia).
käläkättäjä(X):-töissä(X,ericsson).
```

$$\texttt{käläkättäjä} = \pi_1 \sigma_{2=\texttt{nokia}} \texttt{töissä} \cup \pi_1 \sigma_{2=\texttt{ericsson}} \texttt{töissä}$$

```
CREATE VIEW käläkättäjä AS
       (SELECT T.työntekijä
        FROM   töissä T
        WHERE  T.yritys='nokia')
       UNION
       (SELECT T.työntekijä
        FROM   töissä T
        WHERE  T.yritys='ericsson')
```

Kalvoilla IV.5.1 suositeltiin, että Prolog-negaatiota

$$\texttt{not}(p)$$

sovellettaisiin vain muuttujattomiin p.

Datalogissa riittää, että kaikki negatoidussa p esiintyvät muuttujat esiintyvät samassa vartalossa myös negatoimattomina (ennen itse negaatiota).

Silloin negaatio vastaa joukkoerotusta:

```
työläinen(X):-töissä(X,Y),not(johtaa(X,Y)).
```

$$\texttt{työläinen} = \pi_1(\texttt{töissä} \setminus \texttt{johtaa})$$

```
CREATE VIEW työläinen AS
       SELECT T.työntekijä
       FROM   töissä T
       WHERE  T NOT IN johtaa
```

Näin datalog jossa on *negaatio muttei rekursiota* on ”relaatiotäydellinen” kyselykieli.

(Ilman negaatiota ei saada erotusta.)

Rekursiiviset datalog-säännöt mahdollistavat sellaiset kyselyt, joita ei voi enää kirjoittaa perusrelaatiomallissa, kuten "ne tytäryritykset `B` jotka konserni `A` omistaa (myös epäsuorasti)":

```
omistaa(A,B):-valvoo(A,B).
omistaa(A,B):-valvoo(A,C),omistaa(C,B).
```

Nyt haetaan relaatioiden välisen yhtälön

$$\texttt{omistaa} = \texttt{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\texttt{valvoo} \times \texttt{omistaa})$$

ratkaisua $\texttt{omistaa} = \ldots$

Kaikista ratkaisuista halutaan *pienin* koska

- seurataan kalvojen IV.3.2 ajatusta pienimmästä mahdollisesta mallista joten
- halutaan pienin mahdollinen vastausrelaatio.

Osoittautuu että tämä pienin ratkaisu voidaan laskea kasvattamalla aluksi tyhjää tulosta yhtälön oikealla puolella kunnes se ei enää muutu:

sitten := $\emptyset$;
repeat
  ennen := sitten;
  sitten := $\texttt{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\texttt{valvoo} \times \text{ennen})$;
until ennen = sitten

(Parempia mutta mutkikkaampia algoritmeja on!)

Tämä operaatio on itse asiassa ekstensionaalisen relaation `valvoo` *transitiivinen sulkeuma* (transitive closure, TC): $\langle a, b\rangle \in$ `omistaa` jos ja vain jos on ketju $\langle a, c_1\rangle \in$ `valvoo`, $\langle c_1, c_2\rangle \in$ `valvoo`, $\ldots$, $\langle c_k, b\rangle \in$ `valvoo`.

Relaatiomallia onkin ehdotettu laajennettavaksi TC-operaatiolla [DD98,§10.6].

SQL3-standardiluonnoksessa tämä laajennus on mukana rekursiivisina näkyminä [DD98,s.395] (syntaksi luennoijan oma arvaus):

```
CREATE VIEW omistaa AS
       valvoo
       UNION
       (SELECT V.konserniyritys,O.tytäryritys
        FROM   valvoo  V,
               omistaa O
        WHERE  V.tytäryritys=O.konserniyritys)
```

Negaation sovittaminen tähän rekursioon on yhä ongelma (samoin kuin kalvoilla IV.5).

Patologinen esimerkki on

```
p:-not(q).
q:-not(p).
```

jossa negaatio osallistuu rekursioon.

Kalvojen IV.3.2 mukaan $\emptyset$ ei vielä ole malli: `not(q)` on totta mutta `p` ei.

Hieman suurempi $\{\mathtt{p}\}$ on jo malli: `not(p)` ei ole totta.

Mutta myös $\{\mathtt{q}\}$ on malli, eikä siis ole vain yhtä pienintä mallia.

Datalog-ajattelussa saadaan yhtälöpari

$$\mathtt{p} = \mathrm{full} \setminus \mathtt{q}$$
$$\mathtt{q} = \mathrm{full} \setminus \mathtt{p}$$

missä $\mathrm{full} = \{\langle\rangle\}$ on relaatiovakio: kaikki 0-paikkaiset monikot.

Jos (yleistetty) algoritmi yrittää laskea sille pienintä ratkaisua (jota ei siis ole), se jää sykkimään:

kierros	p	q
0	$\emptyset$	$\emptyset$
1	full	full
2	$\emptyset$	$\emptyset$
⋮	⋮	⋮

Myös vastaava Prolog-suoritus jää ikuiseen silmukkaan.

Prolog ja datalog ovat uusien *deduktiivisten* eli (epätriviaalia) päättelyä suorittavien tietokantojen keskeisiä perusasioita.

IV.7.3 Vastausjoukot

Kalvojen IV.7 tietokanta-ajattelua tuetaan Prolog-ohjelmoinnissa kirjastopredikaateilla, joilla kaikki kyselyn vastaukset voidaan kerätä yhteen listaan ilman että ne pitäisi poimia yksi kerrallaan [B01,§7.6]:

`findall(`X`,`P`,`L`)` on intuitiivisesti "kerää listaksi L kaikki sellaiset X joilla P".

Toisin sanoen, kutsu P ratkaistaan kaikin mahdollisin tavoin, ja joka löydetyllä ratkaisulla θ (johonkin kohtaan) listaan L lisätään $X\theta$. Sama alkio voi siis toistua.

"Tulostusasu" X voi olla mikä tahansa termi joka sisältää (mm.) kutsun P muuttujia; nämä täytetään vastaavilla löydetyillä arvoilla.

`findall(esimies(P),pomo(P,A),L)` antaa listan `L` kaikista esimiehistä.

L `:= SELECT` X
`FROM` P

Jos kyselyssä P on muuttujia jotka eivät esiinny asussa X (kuten tässä A) niin niiden eri arvot (tässä alaiset) katoavat – ne projisoitiin pois.

Jos P ei ratkea on $L =$ `[]`.

`bagof(`X`,`P`,`L`)` on muuten kuten `findall/3`, mutta ne kyselyn P muuttujat Y jotka puuttuvat asusta X käsitellään kuten Prologissa (eikä kadoteta).

Silloin jokaiselle muuttujan Y eri arvolle a, jolla P ratkeaa, kootaan oma epätyhjä lista L vastaavia tuloksia X.

$$L_a \texttt{ := SELECT } X \texttt{ FROM } P \texttt{ WHERE } Y \texttt{ = } a$$

Nykyinen a löytyy muuttujan Y arvona. Peruutus siirtyy seuraavaan a, tai epäonnistuu, jos niitä ei enää ole.

Notaatiolla `bagof(`X`,`Y`^`P`,`L`)` voi määrätä muuttujan Y käyttäytymään sittenkin kirjastopredikaatin `findall/3` tapaan.

`setof(`X`,`P`,`L`)` on muuten kuin `bagof/3`, mutta L on järjestetty ja sisältää kunkin X vain kerran.

$$L_a \texttt{ := SELECT DISTINCT } X \texttt{ FROM } P \texttt{ WHERE } Y \texttt{ = } a \texttt{ ORDER BY } X$$

Kirjastossa on termien vertailuun predikaatti `@</2` (sekä `@=<`, `@>`, `@>=`, `==` ja `\==`).

IV.8 Syöte ja tulostus

Prolog-kielen syöte ja tulostus noudattaa samaa ajattelutapaa kuin Scheme-kielessäkin (kalvot II.9.7–II.9.8):

- Syöte ja tulostus voidaan ohjata portteihin [joita kutsutaan *virroiksi* (streams)]
- Voidaan lukea (`read`/1 yms.) ja kirjoittaa (`write`/1 yms.) kokonaisia Prolog-termejä.

Yksityiskohdat sivuutetaan. (Toteutukset eivät aina tue kaikkia standardin piirteitä.)

Periaatteessa Prologin pitäisi pystyä *peruuttamaan* myös syötteenluku- ja tuloksenkirjoitusoperaatiot, mutta tämä olisi käytännössä liian vaikeaa: luettu syöte pitäisi palauttaa takaisin ja kirjoitettu tuloste pyyhkiä pois.

Näin *ei* siis tehdä, joten tällaisten ohjelmien käyttäytyminen täytyy selvittää kalvojen IV.3.1 suoritussemantiikalla.

Esimerkiksi kutsu `read(`X`)` lukee muuttujaan X (nykyisestä syöteportista) seuraavan Prolog-termin t ja sen perässä olevan pisteen `'.'`.

Peruutettaessa ei löydy muita ratkaisuja ja t (pisteineen) pysyy luettuna.

Jos X onkin termi u, niin t luetaan (pisteineen) ensin, ja sitten yritetään samastaa t ja u. Jos samastus onnistuu, niin kutsukin onnistuu; muuten kutsu epäonnistuu – mutta molemmissa tapauksissa t on luettu (pisteineen)!

Lukea (ja kirjoittaa) voi myös sääntöjä: ne tulkitaan termeiksi `':-'(pää,vartalo)` missä

- funktori `:-/2` kirjoitetaan väliin
- vartalo on joko yksinäinen termi tai sitten `termi,vartalo`

ja muuttujat numeroidaan uudelleen.

Prolog-ohjelman lataus(kirjasto)predikaatti `consult('tiedostonimi')` (vertaa Scheme-kalvoa II.9.9) lukeekin `read`illa (vertaa Scheme-kalvoja II.9.8) termejä (kalvojen IV.7.1) operaatiolle `assertz/1`.

IV.9 Operaattorit

Väliin kirjoitettavat funktorit (`is`, `:-`, aritmeettiset operaatiot, vertailut, ...) ovat syntaktisesti *operaattoreita*:

- Ne voidaan kirjoittaa operandiensa väliin.
- Kalvojen IV.8 syöte- ja tulostusrutiinitkin tuntevat ne.

Uusia voi esitellä kirjastopredikaatilla

`op`(sidontavoima,assosiatiivisuus,nimi).

Sidontavoima on kokonaisluku 0, ... ,1200.

Pienempi luku sitoo tiukemmin: esimerkiksi `*` on 400, + 500.

Nimi on se atomi josta tehdään operaattori.

Assosiatiivisuus kertoo operaattorin paikan lausekkeessa:

`fx` on "ainoan operandinsa `x` edessä, jonka on oltava aidosti pienempää prioriteettia (tai suluissa)".

`yf` on "ainoan operandinsa `y` jäljessä, joka saa olla myös samaa prioriteettia".

`xfy` on "kahden operandinsa välissä".

Tällainen on vartalossa olevien kutsujen välissä oleva pilkku `','` (vertaa kalvot IV.8).

`yfx` on useimmilla aritmeettisilla operaatioilla, joilla pois jääneet sulut `X-Y-Z` ovat `(X-Y)-Z`.

`xfx` on niillä operaattoreilla, jotka esiintyvät vain yksinään (kuten vertailut).

IV.10 Esimerkki: tyypinpäättelijä

Toteutetaan lopuksi yksinkertainen *tyypinpäättelyalgoritmi* [H97,§3]. Sellaisia löytyy modernien ohjelmointikielten toteutuksista.

Esimerkissä yhdistyvät kurssin teemat:

- funktionaalinen ohjelmointi
- logiikkaohjelmointi
- päättely.

Tyypitämme Scheme-fragmenttia, jossa on vain

- yhden muuttujan funktion luonti `(lambda (`x`)` e`)`
- sellaisen kutsu (`$f$ $a$`)`.

(Laajentaminen jätetään harjoitustehtäväksi ...)

Haluamme esimerkiksi päätellä lausekkeesta

`(lambda (`g`) (lambda (`y`) (`g y`)))`

seuraavaa:

- Symbolin g arvon on oltava tyyppiä "funktio (tuntemattomalta) argumenttityypiltä u (tuntemattomalle) tulostyypille t" koska sitä kutsutaan.

  Merkitään tätä väitettä "$g\colon u \to t$".

- Symbolin y arvon on oltava tätä samaa argumenttityyppiä u (eli $y\colon u$) koska se annetaan argumenttina funktiolle g.

- Koko lausekkeen tyyppi on siis $(u \to t) \to (u \to t)$ koska jos sille annetaan jokin g niin saadaan tuloksena funktio, joka kuvaa parametrin y arvolle $(g\ y)$.

Jos nämä ohjelman käännösaikana tutkittavat tyyppiväitteet eivät päde, on vaarana ajonaikainen virhe; jos pätevät, niin ajonaikainen tyypintarkistus voidaan jättää pois.

Kirjoitamme kunkin tyypitysperiaatteen väitteenä

$$i_1 : t_1, \ldots, i_n : t_n \vdash e : t$$

joka luetaan "koko lausekkeella e on tyyppi t, jos kullakin sen symbolilla i_k on tyyppi t_k".

Ensimmäinen periaatteemme on

$$i : t, \ldots \vdash i : t$$

eli "symbolin i esiintymällä on se tyyppi joka sillä on" – tietenkin!

Toinen periaatteemme on

$$\frac{\Delta \vdash i : t \qquad i \neq j}{j : u, \Delta \vdash i : t}$$

eli "haettaessa symbolin i tyyppiä listasta Δ voidaan sitä edeltävät symbolit j ohittaa".

Vaakaviivan alapuolelle kirjoitetaan tyypitysjohtopäätös, yläpuolelle ne oletukset joilla sen saa päätellä: $j : u, \Delta \vdash i : t$ voidaan päätellä jos $i \neq j$ on totta ja $\Delta \vdash i : t$ voidaan vuorostaan päätellä.

Listaa Δ luetaan siis vasemmalta oikealle ja pysähdytään heti kun sopiva löytyy.

Tässä ovat yksittäisen symbolin i periaatteet.

Kolmas periaatteemme on

$$\frac{x : u, \Delta \vdash e : t}{\Delta \vdash \texttt{(lambda (}x\texttt{) } e\texttt{)} : u \to t}$$

eli "jos lausekkeella e on tyyppi t kun symbolilla x on tyyppi u, niin abstrahoimalla x lausekkeen e parametriksi saadaan funktio tyypiltä u tyypille t".

Koska tamä x laitetaan listan Δ alkuun, se peittää näkyvistä mahdolliset aiemmat x; näin säilyy Schemen leksikaalinen näkyvyys.

Neljäs ja viimeinen periaatteemme on

$$\frac{\Delta \vdash f : u \to t \qquad \Delta \vdash a : u}{\Delta \vdash (f\ \ a) : t}$$

eli "funktion kutsussa kutsuttavan f pitää olla jotakin funktiotyyppiä $u \to t$ ja argumentin a funktion parametrityyppiä u; silloin kutsun palauttama arvo on funktion tulostyyppiä t".

Oletuksiin ilmestyi uusi tyyppi u jota ei esiintynyt johtopäätöksessä: emme tunne (vielä) sitä tarkemmin, mutta sen on oltava *sama molemmissa oletuksissa*.

Tämä on helppo tehdä Prologin loogisilla muuttujilla: luodaan oletuksille yhteinen uusi vapaa muuttuja `U`.

Muutenkin Prolog-ohjelmamme etenee periaatteissa johtopäätöksistä oletuksiin.

Valitaan merkinnöillemme Prolog-vastineet:

- Scheme-symbolit esitetään Prolog-atomeina.

- Δ esitetään listana, jossa kukin pari $i : t$ esitetään terminä `var(`i`,`t`)`.

- Tyyppi $u \to t$ esitetään terminä `fun(`u`,`t`)`.

- Kutsu $(f\ a)$ esitetään terminä `val(`f`,`a`)`.

- Abstaktio `(lambda (`x`)` e`)` esitetään terminä `lam(`x`,`u`,`e`)`.

  Tässä u edustaa symbolille x pääteltävää tyyppiä; se voitaisiin jättää poiskin, mutta kun sen tilalle kirjoitetaan vapaa Prolog-muuttuja `U`, saadaan tietää myös se tyyppi jonka algoritmi symbolille x päätteli.

- Väite $\Delta \vdash e : t$ kirjoitetaan predikaattina `ct(`Δ`,`e`,`t`)`.

  Sen säännöt saadaan suoraan tyypitysperiaatteista: pää johtopäätöksestä, vartalo oletuksista.

```
ct([var(I,U)|_],I,T):-
        unify_with_occurs_check(U,T).

ct([var(J,_)|V],I,T):-
        atom(I),
        I\==J,
        ct(V,I,T).

ct(D,val(F,A),T):-
        ct(D,F,fun(U,T)),
        ct(D,A,U).

ct(D,lam(I,U,E),fun(U,T)):-
        ct([var(I,U)|D],E,T).
```

Esimerkkilausekkeemme tyypitys on nyt

```
?- ct([],lam(g,G,lam(y,Y,val(g,y))),E).

G = fun(_G355, _G462)
Y = _G355
E = fun(fun(_G355, _G462), fun(_G355, _G462))
```

missä $u =$ _G355 ja $v =$ _G462.

Tämä esimerkkitehtävä on tosin puolueellinen Prologin eduksi, sillä käytetty tyypinpäättelyalgoritmi nojaa samastukseen. Toisaalta ohjelmassa ei voitu käyttää pelkästään Prologin omaa virheellistä samastusta, koska kehälliset tyypit piti kieltää:

```
?- ct([],lam(f,F,val(f,f)),_).

No.
```

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: *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.

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.