

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 symboleihin 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 konkreettisenä 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

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

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

$$\mathbf{D}c = 0$$

$$\mathbf{D}(x^n) = n \cdot x^{n-1}$$

$$\mathbf{D}(f(x) + g(x)) = (\mathbf{D}f(x)) + (\mathbf{D}g(x))$$

$$\mathbf{D}(f(x) \cdot g(x)) = f(x) \cdot (\mathbf{D}g(x)) + g(x) \cdot (\mathbf{D}f(x))$$

$$\mathbf{D} \frac{f(x)}{g(x)} = \frac{g(x) \cdot (\mathbf{D}f(x)) - f(x) \cdot (\mathbf{D}g(x))}{g(x)^2}$$

⋮

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.

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

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

Ei-symbolisessa ohjelmoinnissa jatketaan esimerkiksi seuraavasti:

1. Laaditaan lausekkeille jäsentäjä joka tuottaa lausekkeesta puuesityksen.

(Lauseke muistiin.)

2. Muunnetaan derivointisäännöt tämän puun käsittelyoperaatioiksi.

(Näpelöidään muistia.)

3. Lopuksi tulospuu kirjoitetaan vastauksena.

(Ulostetaan näpelöinnin lopputulos.)

Tässä ratkaisussa on kuitenkin edelleen puute: 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 suorittamalla derivointi tapahtuu.

Tuloksena on pieni muun ohjelman sisään upotettu symbolinen ohjelmointikieli:

Mathematica, Maple, AutoCAD (AutoLISP!), GNU/X Emacs, . . .

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

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

Kääntäjienkin kirjoittaminen helpottuu, onhan kääntäminenkin symbolinen toimitus.

Vastaava Prolog-kielinen ohjelma on yksinkertaisimmillaan:

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

```
    number(C).
```

```
dx(x ** N, N * x ** N1):-
```

```
    N1 is N-1.
```

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

```
    dx(F,F1),
```

```
    dx(G,G1).
```

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

```
    dx(F,F1),
```

```
    dx(G,G1).
```

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

```
    dx(F,F1),
```

```
    dx(G,G1).
```

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

Siis ohjelma `dx` on määritelmä kahden lausekkeen relaatiolle.

II Ohjelmointikieli Scheme

Aloitetaan symbolisen ohjelmoinnin käytäntöön tutustuminen (Prolog-kielen sijasta) Scheme-kielestä, joka on yksi Lisp (List 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 notaatio symboleita käsittelevien ohjelmien spesifiointiin; tietokonetoteutus (LISP 1.5) tuli hieman myöhemmin.

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

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

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

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

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

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

II.2 Common Lisp vastaan Scheme

Common Lisp

ANSI-standardoitu.

Standardia iso mapillinen.

Rikas kokoelma tietotyypp-
pejä ja ilmaisuvoimainen
oliojärjestelmä CLOS
[S98].

Standardikirjasto rikas
(koska paljon tyyppejä).

Jonkin verran historiallis-
ta painolastia häiritsee or-
togonaalisuutta.

Tarkoitettu suuriin oh-
jelmistohankkeisiin, mm.
moduulit.

Scheme

ANSI-standardoitu [D96]
mutta kehitys jatkuu yhä
[KCR98].

Standardia 50 s.

Vain perustietotyypit
ja niiden yhdistelyyn
ilmaisuvoimaiset operaa-
tiot, joilla tyypit ja oliot
tehtävissä.

Standardikirjasto suppea
(tee-se-itse).

Hyvin ortogonaalinen
rakenne; ”peruspali-
kat” yhdistyvät toisiinsa
vapaasti.

Ei omaa moduu-
lijärjestelmää, 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 *laiteriippuvaisella* kohdalla y pyörähtää ympäri!

Funktionaalinen ratkaisu on että myös y on funktion parametri, koska arvo riippuu myös siitä:

$\text{foo}(x, y) = (x + y, y + 1) \in \mathbb{Z}^2 \rightarrow \mathbb{Z}^2$. Arvon y lisäominaisuus on koko funktion esitetyn algoritmin, ei vain aliohjelman foo .

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

"Leibnizin periaate"

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

"Perinteisessä" tilaperustaisessa

ohjelmoinnissa a ja b voidaan vaihtaa keskenään vain jos koko ohjelman (muistin) nykyinen tila 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]. Näihin kieliin on kuitenkin lisätty *tyypinpäättely* [H97,P02] eli kääntäjä päättelee tyypit itse ellei ohjelmoija niitä kirjoita. Nyt voi spesifikaation (osia) kirjoittaa tyyppinä ja kääntäjän tarkistaa vastaako koodi sitä. . .

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**ue), arvoa "epätosi" taas vakio #f (**f**alse).

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

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

Konventio: Totuusarvon palauttavan funktion nimi loppuu kysymysmerkkiin.

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

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

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

II.4.2 Numerot

Scheme-standardi määrittelee (Common Lispiä seuraten) *matemaattisesti* oikean 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 ohjelmointikielistä tuttuja äärellisen tarkkuuden liukulukuja.

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

Konventio: Konversiorutiini tyypistä t tyyppiin u on nimeltään $t \rightarrow u$.

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 reaalityyppistä, niin tulos on reaalityyppi – ja siis epätarkka.
- Jos kompleksiluvun imaginääriosasta tulee tarkka 0, niin tulos on reaalityyppiä.

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

II.4.3 Symbolit

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

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

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

Symboleita voi siis esimerkiksi lukea syötteenä ilman eri ohjelmointia.

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

Analoginen esimerkki: Kun meitä pyydetään matemaattiseen lausekkeeseen $x + y + x$

arvo $x = 5$, niin vastaamme $5 + y + 5$.

Tulkitsemme pyynnön siis tarkoittavan, että kaikki saman symbolin x eri esiintymät korvataan vakiolla 5, mutta eri symbolia y ei korvata.

Samuuteen palataan kalvoilla II.8.5.

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

(Vertaa Leibnizin periaatteeseen kalvoilla II.3.)

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

II.5 Kielen ydin

Esitetään aluksi *yksinkertaistetun* Scheme-kielen syntaksi ja semantiikka [AS96,§1.1] jota täydennetään myöhemmin uusilla 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.4.1 ja II.4.2 totuusarvo- ja numerovakiot ovat lausekkeita.

Näiden vakioiden arvot ovat tietenkin ne itse.

Vastaava laskentasääntö ei siis tee mitään.

II.5.1 Muuttujanmäärittelyt

Muuttuja määritellään ja sille annetaan arvo seuraavasti:

(define *nimi lauseke*)

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

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

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

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

(define *h* 1/1000)

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

Muuttujanimien syntaksi on sama kuin symboleiden kalvoilta II.4.3.

– Sekaannuksen vaara on ilmeinen!

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

Lauseke '*nimi* on siis symbolityypin vakio.

Heittomerkkiin palataan kalvoilla II.8.4.

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

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

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

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

Tällaisen ohjelman suoritus lukee tätä jonoa järjestyksessä, ja tavatessaan

määritelmän saattaa voimaan vastaavan esittelyn

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

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

Common Lisp: Muuttujanmäärittely onkin muotoa

(defvar nimi lauseke)

II.5.2 Funktionmäärittelyt

Funktionmäärittely on muotoa

(define (nimi parametri₁ ... parametri_k) lauseke)

joka esittelee muuttujan *nimi* ja antaa sen arvoksi erään *k*-argumenttisen funktion, joka on intuitiivisesti "lausekkeen arvo kun kunkin (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.5.3.

Vakiokirjasto tarjoaa valmiita funktioita (esimerkiksi kalvojen II.4.2 numeriiikkaan); ne on esitelty valmiiksi.

Common Lisp: Funktionmäärittely onkin muotoa

(defun nimi (parametri₁ ... parametri_k) lauseke).

II.5.3 Funktionkutsut

Lauseke

$(nimi\ lauseke_1 \dots lauseke_k)$

on intuitiivisesti ” edellä määritellyn funktion *nimi* kutsu argumenteilla *nimi lauseke₁ ... lauseke_k*” .

Tämän kutsun palauttama arvo lasketaan seuraavasti:

- Lasketaan samoilla säännöillä minkä arvon *argumentti_i* kukin *lauseke_i* saa.
- Haetaan kalvon II.5.2 mukainen määrittely
(*define (nimi parametri₁ ... parametri_k) lauseke*).
- Tehdään sen rungosta *lauseke* sellainen *kopio* jossa kukin muuttuja *parametri_i* saa arvokseen *argumentti_i*.
- Jatketaan laskemalla arvo tälle *kopiolle*.

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

II.5.4 Ehtolauseke

Kalvon II.5.3 funktionkutsuperiaate ei sovellu lausekkeisiin muotoa

$(\text{if } lauseke_{ehto} \ lauseke_{totta} \ lauseke_{epätotta})$

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 sen arvo lasketaankin seuraavasti:

1. Ensin lasketaan minkä arvon a saa $lauseke_{ehto}$.
2. Jos a on #f (kalvolta II.4.1), niin jatketaan laskemalla minkä arvon saa $lauseke_{epätotta}$.
3. Muuten jatketaankin laskemalla minkä arvon saa $lauseke_{totta}$.

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

II.5.5 Arvon laskenta symbolipelinä

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

Määritellään induktiivisesti seuraavat käsitteet:

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

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

Arvot koostuvat vakioarvoista ja *funktioarvoista*.

Funktioarvo on pari $\langle \text{sanakirja}, (nimi_1, \dots, nimi_n) \mapsto \text{lauseke} \rangle$.

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

Sanakirjat ovat äärellisiä jonoja pareista $\langle \text{nimi}, \text{arvo} \rangle$.

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

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

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

Tämä funktio määritellään induktiivisesti tapauksittain *päätelysäännöillä* muotoa

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

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

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

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

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

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

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

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

Lisäksi tarvitaan säännöt jakamattomien tietoalkioiden perusoperaatioille standardikirjastossa, esimerkiksi

$$\frac{l_1; \mathcal{S} \vdash_{\text{eval}} \frac{p_1}{q_1} \quad \dots \quad l_n; \mathcal{S} \vdash_{\text{eval}} \frac{p_n}{q_n} \quad \frac{p_1 \cdot \dots \cdot p_n}{q_1 \cdot \dots \cdot q_n} = a}{(* \ l_1 \ \dots \ l_n); \mathcal{S} \vdash_{\text{eval}} a}$$

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

Ohjelma on yksinkertaistaen:

- äärellinen jono määrittelyitä jotka saatetaan voimaan samassa järjestyksessä.
- Määrittelyjonon loppuksi tulee vielä se lauseke jonka arvo näillä määrittelyillä halutaan tulostaa.
(Oikea REPL jäisi odottamaan lisää määrittelyjä ja lausekkeita.)
- Jonossa seuraava määriteltävä nimi on aina eri kuin mikään sitä edeltänyt määritelty nimi.
(Oikea Scheme sallisi nimen uudelleenmäärittelyn.)
- Jonossa seuraavan määriteltävän nimen arvolauseke viittaa vain sellaisiin nimiin joiden määrittelyt ovat edeltäneet sitä.
(Oikea Scheme sallisi molemminpuolisen (mutual) rekursion.)
- Ohjelman alussa on määrittelyt standardikirjaston niille funktioille joille ei anneta erillisiä sääntöjä.

Suoritus määritellään jälleen sääntöinä.

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

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

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

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

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

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

Tällaista formalisoitua suoritussemantiikkaa voi käyttää esimerkiksi tarkkana spesifikaationa kieltä toteutettaessa.

Lasketaan esimerkkinä käsin luvun 1 kertoma käyttäen näitä sääntöjä.

(Laskenta on paloitetu ulkoasusyistä.)

$$\begin{array}{c}
 \frac{\frac{\text{fac}; \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}, (n) \mapsto \phi \rangle}{(\text{fac } 1); \langle \text{fac}, \langle \mathcal{L}, (n) \mapsto \underbrace{(\text{if } (\text{zero? } n) \ 1 \ (* \ n \ (\text{fac } (- \ n \ 1))) \ \rangle} \rangle} \mathcal{L} \vdash_{\text{eval}} 1}}{\text{merkitään } \mathcal{S}} \quad \frac{\text{1 on vakio}}{1; \mathcal{S} \vdash_{\text{eval}} 1} \quad \phi; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1}{\text{merkitään } \phi} \\
 \\
 \frac{(\text{zero? } n); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} \#f \quad (* \ n \ (\text{fac } (- \ n \ 1))); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1}{\phi; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \\
 \text{standardikirjasto } \mathcal{L} \\
 \vdots \\
 \frac{\frac{\text{zero?}; \mathcal{L} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle}{\text{zero?}; \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle} \quad \psi: n \text{ spesifikaatio}}{\text{zero?}; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle} \quad \frac{\text{1 on vakio}}{n; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \quad \psi; \langle m, 1 \rangle \mathcal{L}' \vdash_{\text{eval}} \#f}{(\text{zero? } n); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} \#f} \\
 \\
 \frac{n; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1 \quad (\text{fac } (- \ n \ 1)); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1 \quad 1 \cdot 1 = 1}{(* \ n \ (\text{fac } (- \ n \ 1))); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \\
 \\
 \frac{\frac{\text{fac}; \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}, (n) \mapsto \phi \rangle}{\text{fac}; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}, (n) \mapsto \phi \rangle} \quad (- \ n \ 1); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 0 \quad \phi; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} 1}{(\text{fac } (- \ n \ 1)); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \\
 \\
 \frac{\frac{\text{1 on vakio}}{n; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \quad \frac{\text{1 on vakio}}{1; \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \quad 1 - 1 = 0}{(- \ n \ 1); \langle n, 1 \rangle \mathcal{S} \vdash_{\text{eval}} 0} \\
 \\
 \frac{(\text{zero? } n); \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} \#t \quad \frac{\text{1 on vakio}}{1; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} 1}}{\phi; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} 1} \\
 \\
 \frac{\frac{\text{zero?}; \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle}{\text{zero?}; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle} \quad \psi: n \text{ spesifikaatio}}{\text{zero?}; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} \langle \mathcal{L}', (m) \mapsto \psi \rangle} \quad \frac{\text{1 on vakio}}{n; \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} 0} \quad \psi; \langle m, 0 \rangle \mathcal{L}' \vdash_{\text{eval}} \#t}{(\text{zero? } n); \langle n, 0 \rangle \mathcal{S} \vdash_{\text{eval}} \#t}
 \end{array}$$

II.6 Korkeampaa ohjelmointia

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

Tätä mahdollisuutta kutsutaan *korkeamman kertaluvun funktioiksi/ohjelmoinniksi* koska

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

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

Derivointisäännöissä ”tekstimuotoisia” lausekkeita pidetään taas passiivisina (kertaluvun 0) alkioina, joten itse säännöt ovat kertalukua 1.

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 $f : \mathbb{R} \rightarrow \mathbb{R}$ nollakohtaa (tai päättyy 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 vakion h vaikkapa kalvolta II.5.1.

Sovelletaan menetelmää luvun $\sqrt{2}$ likiarvon määrittämiseen, 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 parametrinaan sen funktion jota käsittelee; tästähän menetelmässä on kyse.

Haluamme siis välittää funktion `newton` parametrina toisen funktion `f2`. Tämä käy suoraan, koska kalvon II.5.3 funktiokutsussa

(nimi lauseke₁ ... lauseke_k)

voi kutsuttavan funktion *nimi* olla tämän nykyisen funktion jokin *parametri* ja silloin kutsutaan tuota parametrina saatua funktiota argumenteilla *lauseke_i*. Funktio `newton` nousi toiseen kertalukuun:

```
(define h 1/1000)
```

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

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

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

II.6.2 Paikalliset määritelmät

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

Samalla astumme ulos kalvojen II.5.5 yksinkertaisesta mallista.

Siirretään ”derivaattafunktio” `d` pääfunktionsa `newton` sisään. Samalla huomataan että `d` saa parametrina 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 näkyvyysääntöjä* (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 "funktiokehdas" joka palauttaa sisäisestä funktiostaan f_2 eri versioita riippuen parametrin a arvosta: ($g\ 2$) on tuttu $x \mapsto x^2 - 2$, ($g\ 5$) on taas $x \mapsto x^2 - 5, \dots$

Funktio g palauttaa *tiedon siitä mikä funktio f_2 on nykyisessä määrittely-ympäristössään* johon siis kuuluu se parametrin a arvo, jonka g tällä kertaa sai.

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

Ohjelmointikielessä

C funktioita ei edes voi määritellä sisäkkäin.

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

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

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

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

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

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

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

Tuloksena palautettu funktio on siis kaksiosainen paketti:

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

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

Olio-ohjelmoinnissa sen vastine on seuraava:

instanssin tietokentät vastaavat
määrittely-ympäristöä

ainoa metodi vastaa lauseketta.

Rajoittuminen yhteen metodiin ei ole oleellista:
sehän voi vaikkapa toimia ”puhelinkeskuksena” joka
välittää kunkin syötteenään saamansa (symbolisen!)
metodikutsun oikeaan osoitteseensa.

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

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

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

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

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

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

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

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

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

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

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

- Funktionkutsua $(g \ 2)$ vastaa nyt konstruktorikutsu:

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

Se luo tulosfunktioita $f: x \mapsto x^2 - 2$ vastaavan olion o .

- Tämän tulosfunktion f kutsua argumentilla 2 vastaa metodikutsu:

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

II.6.4 Nimettömät funktiot

Kalvojen II.6.3 "funktio tehdas" g osoittaa että *funktio voidaan luoda nimeämättä sitä*: sisäinen funktio f_2 on tosin nimetty, mutta tämä nimi ei näy funktiosta g ulos vaikka siihen tällä kertaa liitetty funktio lähteekin funktion g arvona.

Scheme-kielessä onkin *funktio tekolauseke*

`(lambda (par1 ... park) laus).`

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

Vastaavasti

`(define (nimi par1 ... park) laus)`

on lyhenne usein tarvitulle nimennälle

`(define nimi (lambda (par1 ... park) laus))`

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

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

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

Funktionkutsussa (*lauseke₀ . . . lauseke_k*) sallitaan nyt ennen kutsuttavan funktion nimelle varatussa ensimmäisessä paikassa mielivaltainen lauseke *lauseke₀*, jolle lasketaan arvo samoin kuin muillekin lausekkeille. Ei ole väliä saatiinko arvo

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

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

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

Eräs yllättävä mutta yleinen nimettömän funktion käyttötapa on

$$((\text{lambda } (var_1 \dots var_k) \text{ lauseke}) val_1 \dots val_k)$$

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

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

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

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

$$(\text{let } ((var_1 val_1) \dots (var_k val_k)) \text{ lauseke})$$

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

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

Operaatio ”ota annetusta luvusta p :s juuri” on 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 ("lauseke $f(x)$ ") joukko-opin ("parien $\langle x, f(x) \rangle$ joukko") sijasta.
- Mekaaninen laskeminen lausekkeita sieventämällä (eli $(2 + 3)/5$ on $5/5$ on 1).

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

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

- Leibnizin operaatio "korvaa tämä ilmaus tuolla ilmauksella 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, \dots

Termimuuttujan esiintymä yksinään on yksinkertainen *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 parametripoition 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 $((\lambda x.t) t')$ voidaan korvata
 lausekkeella $t[x \leftarrow t']$ eli " korvaa kaikki
 termimuuttujan x vapaat esiintymät termissä t
 termillä t' .

- Termimuuttujille

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

- Sovelluksille $(t t')[x \leftarrow t''] = (t[x \leftarrow t''] t'[x \leftarrow t''])$.

- Abstraktioille

$$(\lambda x_j.t)[x_i \leftarrow t'] = \begin{cases} \lambda x_j.t & \text{kun } i = j \\ \lambda x_j.(t[x_i \leftarrow t']) & \text{kun } i \neq j. \end{cases}$$

Operaatiolta $t[x \leftarrow t']$ edellytetään kuitenkin lisäksi,
 että jos jokin x' esiintyy vapaana termissä t' , niin
 mikään muuttujan x vapaa esiintymä termissä t ei
 sijaitse minkään alitermin $\lambda x'.t''$ sisällä.

Muutenhan rekursiivinen korvaus $t''[x \leftarrow t']$ tekisi
 ennen vapaasta esiintymästä x' sidotun esiintymän!

Lisäedellytys voidaan taata α -konversiolla: alitermi $\lambda x'.t''$ voidaan tarvittaessa korvata alitermillä $\lambda x''.t''[x' \leftarrow x'']$ kunhan korvaava muuttuja x'' ei esiinny vapaana alitermissä t'' .

Konversion intuitio: $\lambda x'$ esittelee *parametriposition* eli "aukon" johon β -reduktio voi korvata argumenttiterminsä, mutta tälle positiolle annettu nimi x' ei ole merkityksellinen sisälausekkeen t'' ulkopuolella.

Vastaavasti β -reduktion intuitio on *proseduurikutsu* kalvojen II.5 korvausmallissa: $((\lambda x.t) t')$ "suoritetaan" kopioimalla argumentti t' parametripositioon x kutsuttavassa rungossa t .

Useampiparametriset funktiot sisäkkäisin sovelluksin:

$$\underline{((\lambda x_1.\lambda x_2.((f x_1) x_2) p_1) p_2)} \text{ on } \underline{(\lambda x_2.((f p_1) x_2) p_2)} \\ \text{on } ((f p_1) p_2)$$

eli 2-parametrin funktion f arvo argumenteilla p_1 ja p_2 . Välivaiheena on 1-parametrinen "osittain kutsuttu" lauseke.

Menetelmän nimi on *kuritus* (Currying) loogikko Haskell B. Curryn kunniaksi (vaikka sen keksikin toinen loogikko Harold Schönfinkel).

Kurituksen vuoksi λ -kalkyyllissä vähennetäänkin pakollisia sulkuja sopimalla, että termi $(t_1 t_2 t_3)$ tarkoittaa termiä $((t_1 t_2) t_3)$ jne.

[Tämä muistuttaa kalvojen II.5.3 funktionkutsua, mutta Scheme-kielessä *ei* ole suoraan mahdollista kutsua funktiota vähemmällä parametreilla kuin se vaatii ja saada funktio joka odottaa loppuja. Monissa funktionaalisissa kielissä (kuten ML, Haskell,...) on. Scheme-parametrinvälitykseen palataan kalvoilla II.8.9.]

Haluaisimme funktioiden samuudelle *ekstensionaalisuusperiaatteen*: termit t ja t' kuvaavat samaa funktiota täsmälleen silloin kun $(t t'')$ ja $(t' t'')$ päätyvät (α -konversioita vailla) yhteiseen muotoon kun β -reduktiota sovelletaan niihin toistuvasti, olipa syöte t'' mikä tahansa.

On kuitenkin termejä, joihin β -redusoinnin toistaminen ei pääty: Olkoon termi $Y = \lambda x_1.(x_1 x_1)$. Silloin sen sovellus itseensä $(Y Y)$ antaa tulokseksi jälleen $(Y Y)$ (muuttujat muuttaen).

Intuitiivisesti tällaiset termit vastaavat sellaista ohjelman suoritusta, jotka jäävät ikuiseen silmukkaan.

Mikä on silloin termin

$$((\lambda x_1.\lambda x_2.x_2) (Y Y))$$

kuvaama funktio?

1. Ikuinen silmukka jos tarpeettoman parametrin x_1 termi $(Y Y)$ yritetään sieventää ennen sovellusta.
2. Identiteettifunktio $\lambda x_2.x_2$ jos ei yritetäkään.

Ekstensionaalisuusperiaatetta on siis tarkennettava kertomalla myös *missä järjestyksessä* β -reduoidaan.

Scheme ja monet muut (funktionaalisetkin kuten ML) ohjelmointikielet ovat valinneet "sisältä ulos"-vaihtoehdon 1: Kutsuja laskee argumenttien arvot ennen kutsun suoritusta.

Matemaattisena teoriana λ -laskenta onkin valinnut "ulkoa sisään"-vaihtoehdon 2: Valitaan kirjoitusjärjestyksessä vasemmanpuoleisin λ johon voi soveltaa β -reduktiota.

Kutsun suoritus laskee argumenttien arvot vasta silloin jos niitä tarvitsee. Tämän ns. laiskan suorituksen Scheme-toteutukseen palataan kalvoilla II.8.8. (Haskell-ohjelmat suoritetaan laiskasti.)

Tyypiteoria matemaattisen logiikan haarana [H97] saikin alkunsa λ -laskennan termien syntaktisena luokitteluna

hyviin joilla β -redusoinnit aikanaan pysähtyvät (tehtiinpä ne missä järjestyksessä tahansa)

huonoihin joilla voidaan joutua ikuiseen silmukkaan.

Esimerkkitermimme $Y = \lambda x_1.(x_1 x_1)$ on huono:

- Olkoon parametri x_1 tyyppiä τ_1 .
- Koska x_1 on kutsuttavan funktion paikalla lausekkeen Y sisällä, on τ_1 tyyppiä "funktio tyyppin τ_2 syötearvoilta tyyppin τ_3 tulosarvoille".

Merkitään tätä funktiotyyppiä $\tau_2 \rightarrow \tau_3$.

- Koska x_1 on myös oma syötteensä, pitää tyyppien τ_1 ja τ_2 olla samat.
- Silloin myös tyyppien τ_2 ja $\tau_2 \rightarrow \tau_3$ pitää olla samat.
- Silloin tyyppin τ_2 pitäisi olla ääretön $((\dots \rightarrow \tau_3) \rightarrow \tau_3) \rightarrow \tau_3$ — ristiriita!

Tämä luokittelu — tyyppinpäätely — on jälleen yksi esimerkki symbolisesta laskutoimituksesta. . .

Kalvoilla II.5.5 esitellyssä Scheme-ydinkielen suorituskoneissa käytettiin muuttujien korvaamiseen arvoillaan sanakirjoja eikä β -reduktion mekanismeja $t[x \leftarrow t']$.

Näillä menetelmillä on kuitenkin sama intuitio:

- λ -laskennassa formaali parametri x korvataan *nyt heti* arvolla t' termissä t
- Suorituskoneissa merkitään sanakirjaan että jos termin t parametriin x viitataan *myöhemmin*, niin silloin pitääkin toimia ikään kuin sen paikalle olisi jo korvattu arvo t' .

Sanakirja on siis se *korvaushistoria*, joka olisi syntynyt tekemällä vastaavat β -reduktiot.

Merkintää $\lambda x. \dots$ käytetään usein myös muiden funktiomerkitöjen kuten $x \mapsto \dots$ tai $f(x) = \dots$ sijasta.

Se esittää silloin yleistä parametrinvälitysmekanismia.

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 (lat.) toistaa.

Recurrere (lat.) juosta takaisin.

Toistaminen on juoksemista takaisin *alkuun*.

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)))))
```

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 parametrin *n* arvoilla tämä kirjanpito ei enää mahdu muistiin; silloin joudutaan ohjelmaa muuttamaan kirjanpidon pienentämiseksi.

Esimerkiksi laskettaessa kutsun (fac 3) arvoa aktiivinen sievennyskohta etenee yhä syvemmälle lausekkeessa:

```

(fac 3)
= ((lambda (n) (if (zero? n) 1 (* n (fac (- n 1))))) 3)
= (if (zero? 3) 1 (* 3 (fac (- 3 1))))
= (if #f      1 (* 3 (fac (- 3 1))))
= (* 3 (fac (- 3 1)))
= (* 3 (fac 2))
:
= (* 3 (* 2 (fac 1)))
:
= (* 3 (* 2 (* 1 (fac 0))))
:
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6

```

Sen eteen jää kirjanpito: polku

$$n \cdot \underbrace{((m-1) \cdot ((m-2) \cdot \dots \cdot ((m-p+1) \cdot (\text{fac } m-p))) \cdot \dots \cdot 1)}_{(\dagger)}$$

keskeneräisiä kertolaskuja, joka on juostava takaisin.

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 lambdaista poistumista” .

Tarkastellaan kalvojen II.6.4 lambda-lausekkeen yleistä muotoa:

$$(\text{lambda } (x_1 \dots x_k) d_1 \dots d_m e_1 \dots e_m)$$

missä kukin

- d_i on kalvojen II.6.2 paikallinen `define`-määritelmä.

Kalvojen II.6.2 ja II.6.4 nojalla `lambda` siis avaa uuden paikallisen määrittelyalueen.

- e_j on suoritettava lauseke.

Näistä viimeinen eli e_m antaa funktion arvon.

Edelliset eli e_1, \dots, e_{m-1} suoritetaan suoritetaan *sivuvaikutustensa* (syöte/tulostus, uudelleensijoitus, ...) vuoksi. Sivuvaikutuksiin palataan kalvoilla II.9; vielä nyt Scheme-ohjelmissamme $m = 1$.

Koko tämän `lambda`-lausekkeen viimeinen vaihe on laskea kutsulle arvo alilausekkeella e_m . Se ja vain se on siis loppuyhteydessä.

Jos kalvojen II.5.4 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

$((\lambda (var_1 \dots var_k) laus_1 \dots laus_m) val_1 \dots val_k)$

niin

1. selvitetään ne (lausekkeen $laus_m$ ali)lausekkeet jotka ovat loppuyhteydessä tähän sisempään lambdaan ja
2. otetaan ne myös ulomman lambdaan loppuyhteyteen

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

(Vertaa kalvojen II.6.5 β -reduktio.)

Siis kalvojen II.6.4 let-rakenteessa

$(let ((var_1 val_1) \dots (var_k val_k)) laus_1 \dots laus_m)$

loppuyhteydessä on vain $laus_m$.

Näillä säännöillä voi (johdettuja lausekkeitä 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.

Etuna on, että ohjelmoija voi kirjoittaa induktiivisen määritelmän sellaisenaan, ja Scheme-toteutus pyrkii suorittamaan sen muistia säästäen. Vasta jos siinä ei onnistuta, joutuu ohjelmoija muokkaamaan koodiaan.

Tällaista "epäaitoa" loppurekursiota kutsutaan usein *häntä- tai takarekursioksi* (tail recursion) ja Scheme-kielen standardin sanotaan vaativan sen *poistoa* (elimination) jota pidetään tavallisesti (ohjelmoijien tai) ohjelmointikielitetutusten *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.

Kalvojen II.6.4 nimetön lambda-funktio ei nimittäin voi olla rekursiivinen (ilman temppuilua) – millä nimellä se kutsuisi itseään?

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

$(\text{let } \textit{nimi} ((\textit{var}_1 \textit{val}_1) \dots (\textit{var}_k \textit{val}_k)) \textit{laus}_1 \dots \textit{laus}_m)$

jossa lisäksi rungon lausekkeisiin \textit{laus}_i – muttei muualle – näkyy k -parametrinen funktio *nimi*.

Tämän paikallisen funktion kutsu $(\textit{nimi} \textit{val}'_1 \dots \textit{val}'_k)$ tekee rungon uudelleen arvoilla $\textit{var}_i = \textit{val}'_i$.

Loppukutsuna se siis vastaa silmukan seuraavaa kierrosta. Sitä voi myös käyttää tavallisena rekursiokutsuna muissa kuin loppuyhteyksissä.

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`, joka peittyä näkyvistä.

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 (\dagger) eli $m \cdot (m - 1) \cdot (m - 2) \cdot \dots \cdot (m - p + 1)$ 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

$n! = (\text{fac2 } n) = (\text{fac-acc } n \ \alpha) = n! \cdot \alpha$ 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

$\beta = (\text{if } (\text{zero? } m) \ \gamma \ \delta)$ 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

$$\delta = m! \cdot a = (\text{fac-acc } (- m 1) \eta) \underset{\text{ind.ol.}}{=} (m - 1)! \cdot \eta$$

kunhan valitaan $\eta = m \cdot a = (* m a)$.

Yhdistetään kaikki lausekkeet α, \dots, η paikoilleen:

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

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

- Iteratiivinen kuten pitikin:

```
(fac2 3 1)
= (fac2 (- 3 1) (* 1 3))
= (fac2 2 3)
⋮
= (fac2 (- 2 1) (* 3 2))
= (fac2 1 6)
⋮
= (fac2 (- 1 1) (* 6 1))
= (fac2 0 6)
⋮
= 6
```

- Korrekti jo *konstruktionsa nojalla*.

Vaikka tämä vaatikin hieman matematiikkaa, oli se silti helpompaa kuin

- keksiä ensin algoritmi ja todistaa se(n idea) myöhemmin oikeaksi
- kokeilla ja testata ja korjata ja testata ja korjata ja...

Kalvojen II.3 funktionaalisessa ohjelmointitavassa säilytettiin Leibnizin periaate, ja sen nojalla voitiin tehdä *yhtälöpäättelyä* "tämä lauseke = tuo lauseke".

Tilaperustaisen ohjelmoinnin

"tämä käskysarja = tuo käskysarja" on vaikeampaa, koska

- on vaikeampi nähdä milloin kaksi eri sarjaa jotavat samaan lopputulokseen
- lopputulosten samuus voi riippua alkutilanteesta.

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)

$$f(n) + \frac{1}{f(n-1) + \frac{1}{f(n-2) + \dots + \frac{1}{f(1)}}}$$

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` kerääjäfunktion `a` invarianttinaan ”kun parametrina annetaan nykyisen rekursiokutsun antama tulos, saadaan vastauksena koko rekursion tulos”: $(\text{cf-acc } m \ a) = (a \ (\text{cf-aux } m)) = (\text{cf } f \ n)$.

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

Aluksi $(\text{define } (\text{cf2 } f \ n) \ (\text{cf-acc } n \ \alpha))$ missä

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

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

Jatkamalla kalvojen II.7.2 tapaan saadaan

$$(\text{define } (\text{cf-acc } m \ a) \ (\text{if } (\text{zero? } m) \ \gamma \ \delta))$$

missä γ edustaa ei-rekursiivista haaraa 0. Tällaisessa 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

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

on kerääjäfunktion `a` päivitetty muoto seuraavaa iteraatiokierrosta varten:

- Parametrina `v` saadaan (aikanaan) tämän rekursiokutsun (\dagger) vastaus $v = (\text{cf-aux } (- \ m \ 1))$.
- Tätä vastausta `v` jatkokäsittellään ensin paikallisesti lausekkeella

$$w = \frac{1}{f(m) + v} = (\text{cf-aux } m).$$

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

Toistetaan sama järkeily formaalisti yhtälöillä:

$$\begin{aligned}
 & (\text{cf-acc } m \ a) \quad \text{missä } m > 0 \\
 & = (a \ (\text{cf-aux } m)) \\
 & \quad \text{kerääjän } a \text{ oikeellisuusehto etuperin} \\
 & = (a \ (/ \ 1 \ (+ \ (f \ m) \ (\text{cf-aux } (- \ m \ 1)))) \\
 & \quad \text{funktion } \text{cf-aux} \text{ määritelmä kun } m > 0 \\
 & = ((\text{lambda } (v) \ (a \ (/ \ 1 \ (+ \ (f \ m) \ v)))) \ (\text{cf-aux } (- \ m \ 1))) \\
 & \quad \text{kalvojen II.6.5 } \beta\text{-reduktio takaperin} \\
 & = (\text{cf-acc } (- \ m \ 1) \ \underbrace{(\text{lambda } (v) \ (a \ (/ \ 1 \ (+ \ (f \ m) \ v))))}_{\eta}) \\
 & \quad \text{kerääjän } a \text{ oikeellisuusehto takaperin}
 \end{aligned}$$

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 rekursiokirjanpitoon, kuten pitikin.

Abstrahoidaan ratkaisua edelleen:

- Nyt `cf-acc` palauttaa arvonaan kerääjäfunktion `a`.
- Tuloksen `a` kokoamiseen käytetään matematiikasta tuttua funktioiden f ja g yhdistämisoperaattoria $f \circ g: x \mapsto f(g(x))$.
- Tuloksen `a` alkuarvona on operaattorin \circ neutraalialkio `id`: $x \mapsto x$.
- Tulokseen `a` lisätään murtojatkeessa toistuva lauseke.

```
(define (o f g)
  (lambda (x)
    (f (g x))))
(define id
  (lambda (v) v))
(define (cf2b f n)
  (define (cf-acc m a)
    (if (zero? m)
        a
        (cf-acc (- m 1)
                 (o a
                   (lambda (v)
                     (/ (+ (f m)
                           v))))))))
  ((cf-acc n id) 0))
```

Oleellisesti sama funktio kuin kalvojen II.7.3 `fac2!`

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* (continuation) ja ohjelmointityyliä jossa ohjelmoija hallinnoi jatkeita itse – järjestelmän sijasta – *jatkeidenvälitystyyliseksi* (*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 *toteutustason* 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 voidaankin (vaikkapa) käsitellä virhe:

```
(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)))
```

Tässä siis luodaan jatke, joka sisältää virheenkäsittelyn: jos jakajaksi denom on tulossa 0, niin *oikaistaan* ohi lopun "tulevaisuuden" a ja palataan suoraan kutsujaan virhearvolla #f.

Näin voidaan toimia silloin, kun virhe ilmenee rekursiosta paluumatkalla.

Jos virhe ilmenee jo rekursioon edetessä, niin käsittely on vielä helpompaa: koska rekursio on muutettu iteraatioksi, niin riittää yksinkertaisesti poistua silmukasta — jälleen oikaisten ohi jatkeen.

Oletetaan esimerkissämme lisäksi, että myös parametrina saatu funktio f voi antaa tuloksenaan numeron sijasta virhearvon $\#f$:

```
(define (cf4 f n)
  (define (cf-exit m a)
    (if (zero? m)
        (a 0)
        (let ((fm (f m)))
          (if fm
              (cf-exit (- m 1)
                        (lambda (v)
                          (let ((denom (+ fm v)))
                            (if (zero? denom)
                                #f
                                (a (/ denom))))))
              #f))))))
  (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 Scheme-kielessä voi ilmaista pakeneminen kesken rekursion ohjelmoijan ylläpitämällä 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` pakatoi 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. Kirjastofunktioikutsulla

```
(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 funktionaalisisessa ohjelmoinnissa) *listat* ovat keskeinen rakenteinen tyyppi, Scheme-kielessä jopa miltei ainoa [AS96,§2.2;KCR98,§6.3.2;D96,§6.2].

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

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

II.8.1 Tyhjä lista

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

Schemessä tyhjää listaa merkitään '()'.
(Tämä on kuitenkin vanha tapa, nykyään tyhjä lista merkitään ())

Jo kalvoilla II.5 nähdyn heittomerkin ''' tarkoitukseen palataan kalvoilla II.8.4.

(DrScheme-toteutuksen kielitaso "full scheme" näyttää kuitenkin sallivan sen jättämisen pois.)

Testi `null?` tutkii onko sen argumentti tyhjä 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.

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

tuottaa "sisältä ulos" luettuna

1. 1-alkioisen listan liittämällä tyhjän listan alkuun alkion 4
2. 2-alkioisen listan alkioista 3 ja 4 liittämällä askeleessa 1 tehdyn listan alkuun alkion 3
3. listan alkioinaan 2, 3 ja 4
4. ja lopuksi listan alkioinaan 1, 2, 3 ja 4.

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

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

- Jokainen jakamaton tietoalkio on omassa laatikossaan.
- Kutsu (`cons a l`) luo uuden laatikon jossa 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 `(\dagger)` tuottaa aidon listan.

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

Aidon n -alkioisen listan *tulostusasu* on

$$(\text{alkio}_1 \text{ alkio}_2 \text{ alkio}_3 \dots \text{alkio}_n)$$

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

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

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 tällaisia listankaivuoperaatioita käytetään usein, on vakiokirjastossa

```
(define (cs1 ... skr x) (cs1r (cs2r ... (cskr x)))
```

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

Erityisesti n . alkion kaivaa `cadnr`.

Nyt meillä on (minimi)välineet *listarekursioon*.

Tehdään esimerkkinä alkioit kahdentava listakuvaus

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

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 (\dagger).

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

Kalvojen II.4.3 symbolityypin vakioihin heittomerkkiä kuitenkin jo tarvittiin erottamaan ne kalvon II.5.1 muuttujista.

Heittomerkki siis pysäyttää arvon laskennan siihen vakioon, joka löytyy heittomerkin takaa. Se vakio voi olla rakenteinenkin.

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

Esimerkiksi siis funktiossa

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

ylempi `'x` on symbolinen vakio `x`

alempi `x` on parametrin `x` arvo

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

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

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

Nurin väännetyn heittomerkin sisällä *pilkku* lausekkeen `e` edessä eli `,e` (lyhenteenä ilmaukselle `(unquote e)`) *pakenee* lainausmerkkien sisältä lausekkeen `e` ajaksi, eli laskee sen arvon ja liittää sen synnyttävän 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.

Heittomerkillä *nurinpäin* on kätevää ilmaista sellaisia rakenteita, jotka ovat melkein mutta ei ihan vakioita.

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

II.8.5 Samansisältöisyys ja samuus

Lukujen yhtäsuuruuden eli samuuden tutkimiseen oli operaatio =.

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

`(equal? a b)` palauttaa toden intuitiivisesti silloin kun lausekkeiden *a* ja *b* arvot ovat *tulostasultaan* 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 $\#t$.

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

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 kun saattaa sijaita useassa eri laatikossa.

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

Yleensä `equal?` lienee luonnollisin vaihtoehto.

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

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

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

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

Funktioilla (eli lambda-lausekkeiden arvoilla) ei ole

- (toteutusriippumattonta) 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 jakoi Scheme-lausekkeen kahteen osaan:

Ulkopuolella oleva lausekkeen osa suoritetaan koko lausekkeen lausekkeen arvon laskemiseksi.

Tällä puolella esiintyy kalvojen II.5.1 muuttujia.

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

Tällä puolen esiintyy kalvojen II.4.3 symbolityypin vakioita.

On harmillista että (Lisp-historiallisista syistä)

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

vaikka ne ovat eri käsitteitä.

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

tunnuksina erottamaan eri tyyppiset lausekkeet toisistaan

niminä kohdealueen käsitteille.

Tällöin ohjelmakoodi voidaan edelleen lukea reseptinä vastaavalle symbolinmanipulointitehtävälle:

"Syötelauseke `exp` on muotoa

vakio numerona

muuttuja symbolina

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

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

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

```

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

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

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

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

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

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

(define (addend s) (cadr s))

(define (augend s) (caddr s))

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

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))

```

II.8.7 Assosiaatiolistat

Joskus halutaan ohjelmassa liittää datassa tavatulle (syöte)symbolille jokin "arvo" jota ei kalvojen II.8.4 mukaisesti järjestelmä tee. Tämä kirjanpidon voi ohjelmoija hoitaa suoraan listalla pareja

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

jota kutsutaan *assosiaatiolistaksi* [KCR98,§6.3.2].

Syötesymbolia *key* listan *L* mukaan vastaava arvo löytyy kirjastokutsulla (*assq 'key L*) joka palauttaa *ensimmäisen* parin (*key . arvo*) 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.

Kalvojen II.5.5 sanakirjat olivat assosiaatiolistoja.

II.8.8 Laiskat listat

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

Anon.

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

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

1. Tee järjestetty lista $2, 3, 4, \dots, n$ 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 $3, 5, 7, 9, 11, \dots$

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

```

(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 seuralistan tarvitsisimme jotta siihen jäisi lopuksi (ainakin) n lukua.

Tai jos *jatkaisimme seuralistaa tarpeen mukaan!*

Laiska lista on

joko tyhjä lista () (kuten tavallista)

tai pari (**alkio** . **lupaus**) missä

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

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

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

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

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

Laiskoilla listoilla ohjelmointikuri on seuraava:

Alkion lisääminen (eli "cons") hoituu siten, 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 – sitähan ei ole "luvattu".

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

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

Koska nyt seuralistaamme pidennetään vain tarpeen mukaan, voimme abstrahoida ylärajan kokonaan pois ja ohjelmoida *äärettömän pitkillä* numerolistoilla

2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ...; 3, 5, 7, 9, 11, ...;
5, 7, 11, ...; ...

```
(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

`fromlow = (low . lupaus parista fromlow+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 $p = (\text{delay } e)$ lausekkeelle e laskettaisiin arvo a uudelleen joka kerran kun p lunastetaan operaatiolla ($\text{force } p$).

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 lausekkeet olisivat näkymättömien `delay`-operaatioiden sisällä, joihin ohjelmaa suoritettaessa kohdistettaisiin näkymättömiä `force`-operaatioita *mutta vasta silloin kun niiden arvoa tarvitaan*.

Suoritusmekanismissa varsinkin kalvojen II.5.3 *funktiokutsu* muuttuisi:

Ennen kutsun (`lauseke0 ... lausekek`) *jokainen* lauseke laskettiin arvoonsa.

Nyt vain *ensimmäinen* (eli `lauseke0`) `force`-laskettaisiin (jotta selviäisi kutsuttava funktio). Muut lausekkeet (sen parametrit) `delay`-viivytettäisiin tehtävässä kopiesa.

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 kutsuttavaan 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 on kalvojen II.6.5 `lambdakalkyylin` "oikea" eli "ulkoa sisään"-sievennysjärjestys.

Schemellä jokainen näistä on kohtuullisen vaivatonta.

Tarkastellaan ohjelmointiesimerkkinä kalvojen II.6.5 ”ulkoa sisään” -sieventäjää beta.

Siinä on käytetty

- näiden kalvojen laiskan suorituksen ideaa
- toteutettuna näiden kalvojen delay/force-mekanismilla.
- kalvojen II.5.5 sanakirjaidea
- toteutettuna kalvojen II.8.7 assosiaatiolistoilla.
- kalvojen II.8.6 ideaa symboleista tunnuksina eli niminä
- toteutettuna erillisinä apurutiineina.
- poistumista virhetilanteessa kesken rekursion
- toteutettuna kalvojen II.7.5 järjestelmäjatkeilla.


```

; Tyypittömän lambda-laskennan beta-redusoija.
; 30.09.2002 Matti Nykänen
;
; Lambda-laskennan <termi> on (pitkässä muodossa):
; * muuttuja (VAR <symboli>).
;   Ulkoasu (lyhyessä muodossa) on pelkkä <symboli>, ei kuitenkaan '^'.
; * vakio (CON <datum>).
;   Ulkoasuna ovat <numero>t, <merkkijono>t ja ne <symboli>t joita ei ole
;   abstrahoitu ja jotka eivät ole (tulostukseen varattua) muotoa
;   'x<indeksi>'.
; * abstraktio (ABS <symboli> <termi>).
;   Ulkoasu on (^ <symboli_1> ... <symboli_n> <termi>).
; * sovellus (APP <termi_1> <termi_2>).
;   Ulkoasu on (<termi_1> <termi_2> <termi_3> .... <termi_n>).
;   Myös ylimääräiset sulut eli (<termi>) sallitaan ulkoasuna.
; * funktioarvo (FUN <symboli> <termi> <ympäristö>):
;   * <symboli> on <symboli> joka nimeää parametrinä toimivan muuttujan
;   * <termi> on funktion määrittelevä lauseke jossa <symboli> on vapaa
;   * <ympäristö> antaa vastineet <termi>n muille vapaille muuttujille.
;   Funktioarvoja ei syötetä eikä tulosteta, joten niillä ei ole omaa
;   ulkoasua.
;
; Tämä <ympäristö> taas on assosiaatiolista <symboli>selta
; muuttujanimeltä vastaavalle <arvo>lle. Tämä <arvo> on <lupaus>
; yrittää sieventää vastaava <termi> funktioarvoksi.
;
; Jos sievennysyritys epäonnistuu, eikä saada haluttua funktioarvoa,
; niin palautetaan (pitkässä muodossa) kesken jäänyt sovellus
; (<muu kuin funktioarvo> <termi_1'> ... <termi_k'>)
; missä kukin <termi_i'> on vastaava argumentti<termi_i> mutta täysin
; normalisoituna.
; Tämä voidaan sitten tulostaa (lyhyessä muodossa).

```

```

(define (beta syote-termi)
  (define (tee-vakio datum)
    (list 'CON datum))
  (define (on-vakio? termi)
    (eq? (car termi) 'CON))
  (define (tee-muuttuja symboli)
    (list 'VAR symboli))
  (define (on-muuttuja? termi)
    (eq? (car termi) 'VAR))
  (define (tee-abstraktio symboli termi)
    (list 'ABS symboli termi))
  (define (on-abstraktio? termi)

```



```

                                                    ymparisto))))))
((on-funktioarvo? termi)
 termi)))
(define (normaalimuotoon termi)
  (cond
    ((on-vakio? termi)
     termi)
    ((on-sovellus? termi)
     (tee-sovellus (normaalimuotoon (sovellus-termi_1 termi))
                   (normaalimuotoon (sovellus-termi_2 termi))))
    ((on-funktioarvo? termi)
     ; Hack: totetaan uusi <vakio> vanhan <muuttuja>n tilalle, koska
     ; niiden lopullisissa tulostusasuuissa ei ole eroa.
     (let ((alias-symboli (uusi-symboli)))
       (tee-abstraktio
        alias-symboli
        (normaalimuotoon (funktioarvoksi
                          (tee-sovellus termi
                                         (tee-vakio alias-symboli))
                          ymparisto-tyhja)))))))
(define uusi-symboli
  ; Tämä (ja vain tämä) funktio on tilaperustainen.
  (let ((indeksi -1))
    (lambda ()
      (set! indeksi (+ indeksi 1))
      (string->symbol (string-append "x" (number->string indeksi))))))
(define (ulkoinen-muuttuja? datum)
  (and (symbol? datum)
       (not (eq? datum '^))))
(define (pitka-termi lyhyt-termi)
  (call-with-current-continuation
   (lambda (virhe)
     (let pidenna ((lyhyt lyhyt-termi)
                   (muuttujat '()))
       (cond
        ((ulkoinen-muuttuja? lyhyt)
         (let ((alias (assq lyhyt muuttujat)))
           (cond
            (alias
             (tee-muuttuja (cdr alias)))
            ((let ((tekstina (symbol->string lyhyt)))
              (and (char=? (string-ref tekstina 0)
                           #\x)
                   (string->number
                    (substring tekstina
                               1
                               (string-length tekstina))))))
           (virhe
            "Kirjoita <symboli>set vakiot muuten kuin 'x<numero>'!"))))

```

```

        (else
          (tee-vakio lyhyt))))))
((and (list? lyhyt)
      (not (null? lyhyt)))
 (cond
  ((null? (cdr lyhyt))
   (pidenna (car lyhyt) muuttujat))
  ((eq? (car lyhyt) '^)
   (if (null? (caddr lyhyt))
       (virhe "Ei <abstraktori>a ilman <muuttuja>a!")
       (let abstrahoi ((lyhyet (cdr lyhyt))
                       (muuttuneet muuttujat))
         (cond
          ((null? (cdr lyhyet))
           (pidenna lyhyet muuttuneet))
          ((ulkoinen-muuttuja? (car lyhyet))
           (let ((alias (uusi-symboli)))
              (tee-abstraktio alias
                              (abstrahoi (cdr lyhyet)
                                           (cons
                                            (cons (car lyhyet)
                                                  alias)
                                            muuttuneet))))))
          (else
           (virhe
            "Abstrahoida voi vain <symboli>sia muuttujia!"))))))))
 (else
  (let sovellla ((lyhyet (cdr lyhyt))
                 (pitkat (pidenna (car lyhyt)
                                   muuttujat)))
    (if (null? (cdr lyhyet))
        (tee-sovellus pitkat
                       (pidenna lyhyet muuttujat))
        (sovellla (cdr lyhyet)
                   (tee-sovellus pitkat
                                   (pidenna (car lyhyet)
                                           muuttujat))))))
  ((or (number? lyhyt)
        (string? lyhyt))
   (tee-vakio lyhyt))
  (else
   (virhe
    "Anna <vakio>na <numero>, <merkkijono> tai muu <symboli> kuin '
(define (lyhyt-termi pitka-termi)
  (cond
   ((on-muuttuja? pitka-termi)
    (muuttuja-symboli pitka-termi))
   ((on-vakio? pitka-termi)
    (vakio-datum pitka-termi))

```

```

((on-sovellus? pitka-termi)
  (let sovellla ((pitka (sovellus-termi_1 pitka-termi))
                (lyhyt (list (lyhyt-termi
                              (sovellus-termi_2 pitka-termi)))))
    (if (on-sovellus? pitka)
        (sovellla (sovellus-termi_1 pitka)
                  (cons (lyhyt-termi (sovellus-termi_2 pitka))
                        lyhyt))
        (cons (lyhyt-termi pitka)
              lyhyt))))
((on-abstraktio? pitka-termi)
  (cons '^
        (let abstrahoi ((pitka pitka-termi))
          (if (on-abstraktio? pitka)
              (cons (abstraktio-symboli pitka)
                    (abstrahoi (abstraktio-termi pitka)))
              (list (lyhyt-termi pitka)))))))
(let ((syotetty (pitka-termi syote-termi)))
  (if (string? syotetty)
      syotetty
      (lyhyt-termi (normaalimuotoon (funktioarvoksi syotetty
                                     ymparisto-tyhja))))))

```

(Listaus jatkui yli kalvorajojen.)

II.8.9 Parametrilistan otto

- `(lambda (x1 ...xn) ...)` antaa funktion joka vaatii *tasan* n argumenttia kuten kalvoilla II.6.4 kerrottiin.
- `(lambda (x1 ...xn . y) ...)` antaa funktion joka vaatii *ainakin* n argumenttia.

Kutsussa

$$((\text{lambda } (x_1 \dots x_n . y) \dots) a_1 \dots a_m)$$

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

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

- (λ () ...) 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 ".

(Parametri- ja argumenttilistojen ei tarvitse olla todellisia fyysisiä listoja.)

II.8.10 Parametrilistan anto

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

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

Kirjastofunktiokutsu $(\text{apply } f L)$ 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.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.8.8 kerrottiin delay-kirjanpidosta joka tallettaa lupauksen 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.

Mukaan tulee

- laskulaitteen muistin
- ympäröivän maailman

tilan käsite.

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 muuttujien arvoista pidetään kirjaa [AS96,§3.2].

Kehys (frame) varastoi äärellisen kuvauksen muuttujilta arvoille:

- Kuvaus on määritelty vain äärellisen monelle muuttujalle.
- Arvo jolle muuttuja kuvataan on joko nuoli kalvojen II.8.2 laatikkoon tai kalvojen II.6.4 `lambda`-lausekkeen arvona saatu funktio (josta enemmän kalvoilla II.9.2).

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 muuttujan esiintymälle s silloin kun lauseketta e suoritetaan [AS96,kuva 3.1]:

1. Ensin katsotaan onko ensimmäisen kehyksen kuvaus määritelty muuttujalle 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: muuttuja s ei olekaan määritelty lausekkeessa e .

Kalvojen II.5 käsitteillä

- kehys on se jono pareja $\langle \text{nimi}, \text{arvo} \rangle$ jonka *yksi* funktionkutsu lisää sanakirjaan
- ympäristö on sanakirja, paitsi että sama kehys voi olla jaettu usean eri sanakirjan kesken.

(Itse asiassa toteutuksen ei tarvitse selata ympäristöjä suoritusaikana: käännoäsaikana voidaan laskea etukäteen kuinka kaukaa arvo löytyy.)

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 perustyyppin *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 näkyvyysäännöt.
- Viite proseduurin parametreihin ja runkoon.

Kalvojen II.8.5 funktioiden *eq?*-vertailu testaa (yksinkertaisimmillaan) näitä osia.

Määrittely globaalissa ympäristössä (define s e)

- lisää tämän globaalin ympäristön kuvaukseen uuden muuttujan 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 ($e_0 \dots e_k$) suoritetaan nykyisessä ympäristössä p seuraavasti:

1. Evaluoidaan lausekkeille e_0, \dots, e_k arvot a_0, \dots, a_k nykyisessä ympäristössä p .
2. Tarkistetaan että a_0 on proseduuuri jolla on parametrit x_1, \dots, x_k (tai jokin muu kalvojen II.8.9 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]

Loppukutsuille kalvoilta II.7.1 odottava p poistetaan odottajien ketjusta *ennen* evaluointia, ja raportoidaan vastaus suoraan näin löytyneeseen ympäristöön, eli siihen joka odotti ympäristöä p .

Järjestelmäjatke kalvoilta II.7.5 on juuri tämä evaluoitavien lausekkeiden α "kuka odottaa arvoa keneltä" -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 muuttujaa 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].

- Parhaillaan luotavan paikallisen muuttujan alustuslausekkeessa *voi mainita* samaa ja muita parhaillaan luotavia muuttujia.

Jopa vuorottainenkin rekursio on siis mahdollista.

- Niiden arvoja *ei voi käyttää* koska niiden alustuslausekkeitä ei vielä ole suoritettu. [KCR98,§5.2.2]
- Siis mainintojen on oltava sellaisissa paikoissa, jotka suoritetaan vasta, kun kaikki alustukset on suoritettu:

Kalvojen II.6.4 `lambda` tai kalvojen II.8.8 `delay`n sisällä.
- Kalvojen II.6.2 paikalliset määritelmät ovat vaihtoehtoista syntaksia `letrec`-rakenteelle.

Ehtolause kalvolta II.5.4 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 muuttujalle 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 muuttujalle s katsottaisiin arvo nykyisessä ympäristössä p .

Sijoituksen kohdetta s ei evaluoida: sen on siis oltava sijoitettavan muuttujan nimi eikä mitään muuta.

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

Muutos näkyy juuri niihin evaluointiaskeleisiin joissa viitataan muuttujan 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, kuten kalvolla II.7.1 jo vihjattiin. Nämä lausekkeet 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 t1; ...; tn end`.

Sen Scheme-muoto on lauseke `(begin t1 ... tn)` mikä on pelkkä järjestelmän tarjoama lyhennysmerkintä lausekkeelle

$$((\text{lambda } () t_1 \dots t_n))$$

missä lausekkeista tuotetaan ensin kalvojen II.8.9 ”`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` *konstruoii* 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!` muuttujan arvon muuttamiseen.

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

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

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

Nämä ovat tavallisia kirjastofunktioita:

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

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

Näillä funktioilla saa aikaan yllätyksiä:

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

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

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

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

II.9.5 Yksiulotteiset taulukot

Kalvolla II.9.4 annettiin funktiot parin lokeroiden sisältöjen muuttamiseksi.

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

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

(*vector* $e_1 \dots e_n$) 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 $(\text{vector-length } v) - 1$).

`(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 merkkijonotyyppin (`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 $v_1 \dots v_k$)` palauttaa merkkijonon, joka on saatu yhdistämällä jonot v_1, \dots, 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 charo? 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

Perustulostukseen 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."

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

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

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

(current-input-port) on kuin current-output-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 fakta₁ **ja** ... **ja** fakta_k **niin myös** uusfakta missä kukin *fakta* on yksinkertainen väitelause.

III.1 Termien esitys

Sääntökielellemme 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

ihminen(x).

Esitetään rakennetermit Scheme-listoilla

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

missä ominaisuuden nimi eli *funktori* on symboli.

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

```

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

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

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

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

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

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

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

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

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

```

III.2 Samastus

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

Esimerkiksi termi

(ihminen (sokrates))

on termin

(ihminen x)

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

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

(kuolevainen x)

haluttu tulostermi

(kuolevainen (sokrates)).

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

```
; 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 occur 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 jatkuu 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).


```

; 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>, respectively):
  ; 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))

```

III.3 Sääntöjen esitys

Esitetään sääntö

jos fakta₁ **ja** ... **ja** fakta_k **niin myös** uusfakta
listana

$$\left(\underbrace{\text{uustermi}}_{\text{pää}} \underbrace{\text{termi}_1 \dots \text{termi}_k}_{\text{vartalo}} \right)$$

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 (*mitä tekee?*).

Alitermit ovat silloin muita lauseenjäseniä: subjekti (*kuka tekee?*), objekti (*kenelle tekee?*) tai predikatiivi (*on mikä?*), jne.

Säännöt

1. Sokrates on ihminen
2. kaikki ihmiset ovat kuolevaisia

esitetään siis muodossa

1. ((ihminen (sokrates)))
 - vartalo puuttuu koska ei **jos**-esiehtoja!
2. ((kuolevainen x) (ihminen x))
 - **jos** x on ihminen **niin** x on kuolevainen.
 - x on kuolevainen **jos** x on ihminen.

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 $x = (\text{sokrates})$.
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 *todistaa* uuden faktan (pään, **niin**-osan), joten yritän todistaa sen ehtona olevat faktat (vartalon, **jos**-osan).

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

$$\begin{aligned} &((\text{esi-isa } x \ y) \\ & \ (\text{isa } x \ y)) \end{aligned}$$
$$\begin{aligned} &((\text{esi-isa } x \ y) \\ & \ (\text{isa } x \ z) \\ & \ (\text{esi-isa } z \ y)) \end{aligned}$$

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

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

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

$$(\text{isa } x \ z_1), (\text{isa } z_1 \ z_2), (\text{isa } z_2 \ z_3), \dots, (\text{isa } z_p \ y)$$

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):

$$((\text{esi-isa } X_0 \ X_1) \ (\text{isa } X_0 \ X_2) \ (\text{esi-isa } X_2 \ X_1))$$

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 päättyy 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 epäonnistua säännön valinnassa, päätyä umpikujaan ja peruuttaa. Kalvojen II.8.8 laiskat listat mahdollistavat tällaisen etsinnän ohjelmoimisen *listana onnistumisia*:

Pidetään etsinnän kuluessa yllä (eräänlaisena kalvojen II.7.2 ja II.7.3 kerääjäparametrina) lupaus 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.


```

                (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 jatkuu 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 jatkuu yli kalvorajan.)

III.5 Järjestelmän käyttöä

Olkoon sukupuu esitetty seuraavina faktoina:

$(\text{daughter } x \ y \ z)$: x on äidin y ja isän z tytär.

$(\text{son } x \ y \ z)$: x on äidin y ja isän z poika.

Niillä voidaan määritellä säännöillä käsitteet

$(\text{mother } x \ y)$: x on y :n äiti.

$(\text{grandmother } x \ y)$: x on y :n äidinäiti.

$(\text{ancestress } x \ y)$: x on joku y :n esiäiti.

$((\text{mother } x \ y) (\text{daughter } y \ x \ z))$

$((\text{mother } x \ y) (\text{son } y \ x \ z))$

$((\text{grandmother } x \ y) (\text{mother } x \ z) (\text{mother } z \ y))$

$((\text{ancestress } x \ y) (\text{mother } x \ y))$

$((\text{ancestress } x \ y) (\text{mother } x \ z) (\text{ancestress } z \ y))$

Käytetään vielä faktoja Englannin kuningatar Viktorian jälkeläisistä:

```
((well (albert)))
((son (duke_leopold) (queen_victoria) (albert)))
((ill (duke_leopold)))
((daughter (alice) (queen_victoria) (albert)))
((daughter (empress_victoria) (queen_victoria) (albert)))
((daughter (beatrice) (queen_victoria) (albert)))
((daughter (princess_alice) (princess_helen) (duke_leopold)))
((well (n.n.)))
((son (rupert) (princess_alice) (n.n.)))
((ill (rupert)))
((well (louis)))
((daughter (alix) (alice) (louis)))
((son (frederick) (alice) (louis)))
((ill (frederick)))
((daughter (irene) (alice) (louis)))
((well (nicholas)))
((son (alexis) (alix) (nicholas)))
((ill (alexis)))
((son (waldemar) (irene) (prince_henry_of_prussia)))
((ill (waldemar)))
((son (henry) (irene) (prince_henry_of_prussia)))
((ill (henry)))
((well (emperor_frederick)))
((son (prince_henry_of_prussia) (empress_victoria) (emperor_frederick)))
((well (prince_henry_of_prussia)))
((well (prince_henry_of_battemberg)))
((daughter (queen_victoria_of_spain)
            (beatrice)
            (prince_henry_of_battemberg)))
((son (leopold) (beatrice) (prince_henry_of_battemberg)))
((ill (leopold)))
((son (maurice) (beatrice) (prince_henry_of_battemberg)))
((ill (maurice)))
((well (king_alfonso)))
((son (alfonso) (queen_victoria_of_spain) (king_alfonso)))
((ill (alfonso)))
((son (gonzalo) (queen_victoria_of_spain) (king_alfonso)))
((ill (gonzalo)))
```

Sitten voidaan esittää kysymyksiä kuten ”ketkä olivat (Espanjan prinssi) gonزالon esiäidit”:

NEXT GOAL [() EXITS]? ((ancestress x (gonzalo)))

x = (queen_victoria_of_spain)

ANOTHER ANSWER [y/n]? y

x = (queen_victoria)

ANOTHER ANSWER [y/n]? y

x = (beatrice)

ANOTHER ANSWER [y/n]? y

NO (MORE) ANSWERS.

NEXT GOAL [() EXITS]? ()

EXIT.

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"

algoritmi = logiikka + kontrolli

jonka mukaan ohjelmoinnissa voidaan ja pitää erottaa toisistaan

- itse ratkaistavan ongelman kuvailu
- menetelmä jolla kone ratkaisee ongelman.

Sääntöjärjestelmä erotteli säännöt ja hakukoneen.

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

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

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

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

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

Funktori f kirjoitetaan **atomina** eli

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

Sääntö kirjoitetaan muodossa

$$p :- q_1, q_2, q_3, \dots, q_m.$$

listamuodon

$$(p' \ q'_1 \ q'_2 \ q'_3 \ \dots \ q'_m)$$

sijasta.

Pää p on aina rakenteinen termi eli muotoa $r(\tau_1, \dots, \tau_k)$; 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 .

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

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

IV.3 Prolog vs. perinteinen ohjelmointi

Lähestytään Prolog-ohjelmointia vertaamalla sitä aiemmin tunnettuihin ohjelmointitapoihin:

Aliohjelmatulkinta: Predikaatin määritelmä

```
nimi(syötehahmo(t)1, tuloshahmo(t)1):-  
    aliohjelmakutsu1,1, ..., kutsu1,n1.  
nimi(syötehahmo(t)2, tuloshahmo(t)2):-  
    aliohjelmakutsu2,1, ..., kutsu2,n2.  
:  
nimi(syötehahmo(t)m, tuloshahmo(t)m):-  
    aliohjelmakutsum,1, ..., kutsum,nm.
```

määrittelee samannimisen aliohjelman *tapauksittain*:

```
if syöte sopii hahmoihin 1 then  
    Yritä suorittaa kutsut 1. Vastaa tulos 1 jos onnistuit.  
else if syöte sopii hahmoihin 2 then  
    Yritä samoin kutsuja ja tuloksia 2.  
:  
else if syöte sopii hahmoihin m then  
    Yritä samoin kutsuja ja tuloksia m.  
else  
    Tämä aliohjelmakutsun yritys epäonnistui.  
end if
```

Hahmojen sovitus eli parametrinvälitys tehdään kalvojen III.2 samastuksella.

Prolog on kuitenkin tätä aliohjelmatulkintaa rikkaampi kieli:

- Syöte- ja tuloshahmojen välillä ei olekaan eroa. Sama parametripositio voi olla joskus syötettä, joskus tulostetta.

Esimerkiksi kysymyksellä $?- p(s(o), s(s(o)), Z)$. haetaan sopivaa tulosarvoa muuttujaan Z positiossa 3, kun taas kysymys $?- p(X, Y, s(s(s(o))))$. hakee sopivia tulosarvoja muuttujiin X ja Y positioissa 1 ja 2.

Samastus sitoo kutsuhetken hahmoissa olevia muuttujia siten kuin on tarpeen kutsun onnistumiseksi. Näin ollen "tulosta" voi tehdä "syötteen" mihin tahansa sellaiseen kohtaan, jossa on vapaa muuttuja. Jopa $?- p(X, Y, Z)$. on sallittu kysymys.

Suorituslähtöisesti voi myös ajatella, että aliohjelman "syöte" on se samastin σ (kalvoilta III.2) jonka kutsuja on koonnut kutsuhetkeen mennessä, ja "tulos" on sama σ laajennettuna uusilla sidonnoilla (ja muuttujilla) juuri sen verran kuin aliohjelman onnistumiseen tarvittiin.

- Eri **if**-haarat eivät suljekaan toisiaan pois.

Jos yhden haaran i yritys epäonnistuu, niin siirrytäänkin kokeilemaan seuraavaa haaraa $i + 1$ (eli ulos ohjaavaa **else**-sanaa ei olekaan).

Silloin eri **if**-haarat voivat käsitellä myös (osittain) *samaa* hahmoa, kukin vuorollaan.

- Aliohjelma voi silloin myös *onnistua* useilla eri tavoilla ja "tuloksilla".

Logiikkaohjelmointi pitää kaikkia eri tapoja onnistua samanarvoisina — eri ratkaisuin sääntöohjelman kuvaamaan ongelmaan.

Eri **if**-haaroja voi käyttää missä järjestyksessä tahansa. Tärkeää on vain, että ohjelma sisältää tarpeeksi tietämystä ongelman ratkaisemiseksi.

Prolog suorittaa sääntöohjelmaa, eli etsii ratkaisu(j)a, yhdellä tietyllä järjestelmällisellä tavalla.

Eri **if**-haaroja kokeillaan niiden kirjoitusjärjestyksessä. Silloin tärkeäksi tulee kuitenkin myös kontrolli.

- Kun siis aliohjelman *nimi* **if**-haara i tekee aliohjelmakutsun $\text{kutsu}_{i,j}$ sidonnalla $\sigma_{i,j}$ niin kutsu voi joko

onnistua jolloin jatketaan eteenpäin seuraavaan kutsuun $\text{kutsu}_{i,j+1}$.

Onnistuessaan $\text{kutsu}_{i,j}$ laajensi saamansa sidonnan $\sigma_{i,j}$ sidonnaksi $\sigma_{i,j+1}$, jonka $\text{kutsu}_{i,j+1}$ vuorostaan saa. (Ensimmäisen sidonnan $\sigma_{i,0}$ antaa haaran i pään sovitus kutsuhetken sidonnasta σ_{nimi} .)

Jos seuraavaa kutsua ei ole, vaan törmätään pisteeseen '.', niin on löydetty yksi tapa onnistua tässä kutsuvassa aliohjelmassa *nimi*. Palautetaan kutsujalle sidontana $\sigma_{i,j}$ ja tieto että seuraava haara h_{nimi} olisi ollut $i + 1$.

Epäonnistua jolloin jatketaan taaksepäin edelliseen kutsuun $\text{kutsu}_{i,j-1}$. Palautetaan sidonnaksi se $\sigma_{i,j-1}$, joka oli kerätty tähän onnistuneeseen kutsuun mennessä. Sitten yritetään ratkaista tämä edellinen onnistunut kutsu toisin sen seuraavasta haarasta $h_{\text{kutsu}_{i,j-1}}$.

Jos edellistä kutsua ei ole, vaan törmätään kaulaan ':-', niin yritetään seuraavaa haaraa $i + 1$ sidonnalla σ_{nimi} .)

- Jos seuraavaa haaraa ei ole, vaan pudotaan aliohjelman *nimi* määritelmän alareunasta ulos, niin tämä sen kutsu sidonnalla σ_{nimi} epäonnistui.
- Siis Prolog-ohjelman suorituksessa on 2 vaihdetta:

eteenpäin rungossa oikealle kohden päättävää pistettä.

Jos ollaan umpikujassa, eikä päästä enää eteenpäin, niin sitten mennään...

taaksepäin rungossa vasemmalle ja kaulan kohdalta alas (lopuksi "ali laidan").

Kun on menty askel taaksepäin, yritetään taas eteenpäin. Näin yritetään jotakin toista vaihtoehtoa umpikujaan vieneen sijasta.

Nämä vaihteet näkyvät kalvojen III Scheme-toteutuksessa seuraavasti:

Eteenpäin edetään Schemen normaalissa suorituserjestyksessä, mutta aina kun valitaan jokin sääntö, tehdään myös lupaus tutkia tarvittaessa, mitä tapahtuisi, jos sitä ei valittaisikaan.

Taaksepäin mennään lunastamalla viimeisin näistä lupauksista.

Ensimmäinen Prolog-esimerkkimme olkoon kalvojen III.5 sukupuu.

- Faktat on järjestetty uudelleen siten, että saman predikaatin eri haarat ovat peräkkäin — muuten toteutuksemme SWI-Prolog antaa varoituksia.

Muuttujattomat faktat ovat *vakiodataa* joiden järjestyksellä ei ole niin väliä — se vaikuttaa vain eri vastausten löytymisjärjestykseen.

Muuttujalliset faktat ovat taas *ohjelman ei-rekursiivisia haaroja* — niiden järjestyksellä on siis väliä!

Esimerkiksi ohjelma

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

käyttäytyy eri tavoin kuin kalvojen IV.2 ohjelma: vaikka ohjelmien määrittelemät oikeat vastaukset ovat samat, tämä ohjelma joutuu hakoteille etsinnässä ?- p(X,o,Z). josta aiempi ohjelma selviää.

- Predikaatissa `mother/2` on muuttujan `Z` tilalla erityinen alaviivamuuttuja `'_'` — muuten toteutus antaisi varoituksen ”muuttuja `Z` on turha”.

Tämän muuttujan arvoa *ei tarvita*:

- Se ei esiinny lauseen päässä, eli se ei ole syöte/tulosmuuttuja, kuten `X` ja `Y`.
- Se ei toistu lauseen sisällä, joten se ei välitä tietoa vasemmalta oikealle, kuten `Z` predikaatissa `ancestress/2`.

Prolog-hahmoissa alaviivaa käytetäänkin merkitsemään paikkoja, joissa on jotakin, mikä ei tällä hetkellä kiinnosta ohjelmoijaa.

- Kaksi eri alaviivaa tarkoittavat siis *eri* mielenkiinnottomia asioita, vaikka ne olisivatkin samassa lauseessa:
`ancestress(X,Y):-mother(X,_),ancestress(_,Y).`
 olisi väärin, koska sen mukaan ”`X` on `Y`:n esiäiti, jos `X` olisi jonkun äiti, ja `Y`:llä olisi joku mahdollisesti muu esiäiti”.

daughter(alice,queen_victoria,albert).
daughter(alix,alice,louis).
daughter(beatrice,queen_victoria,albert).
daughter(empress_victoria,queen_victoria,albert).
daughter(irene,alice,louis).
daughter(princess_alice,princess_helen,duke_leopold).
daughter(queen_victoria_of_spain,beatrice,prince_henry_of_battemberg).

ill(alexis).
ill(alfonso).
ill(duke_leopold).
ill(frederick).
ill(gonzalo).
ill(henry).
ill(leopold).
ill(maurice).
ill(rupert).
ill(waldemar).

son(alexis,alix,nicholas).
son(alfonso,queen_victoria_of_spain,king_alfonso).
son(duke_leopold,queen_victoria,albert).
son(frederick,alice,louis).
son(gonzalo,queen_victoria_of_spain,king_alfonso).
son(henry,irene,prince_henry_of_prussia).
son(leopold,beatrice,prince_henry_of_battemberg).
son(maurice,beatrice,prince_henry_of_battemberg).
son(prince_henry_of_prussia,empress_victoria,emperor_frederick).
son(rupert,princess_alice,n_n).
son(waldemar,irene,prince_henry_of_prussia).

well(albert).
well(emperor_frederick).
well(king_alfonso).
well(louis).
well(n_n).
well(nicholas).
well(prince_henry_of_battemberg).
well(prince_henry_of_prussia).

mother(X,Y):-daughter(Y,X,_).
mother(X,Y):-son(Y,X,_).

grandmother(X,Y):-mother(X,Z),mother(Z,Y).

ancestress(X,Y):-mother(X,Y).
ancestress(X,Y):-mother(X,Z),ancestress(Z,Y).

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.
- Ohjelma haarautuu tapauksiin parametrihahmojen mukaan.
- Kukin tapaus on oleellisesti vastaava tasapaino(tus)ehto.
- Tietorakenneoperaatiot esitetään suoraan näillä parametrihahmoilla.
- Tietorakenteiden käsittely hahmonsovituksella on myös moderneissa funktionaalisissa kielissä kuten Haskell [T99] ja ML [CM98]. Niissä eri haarat ovat tietenkin toisensa poissulkevia, koska kullakin funktiokutsulla on vain yksi arvo.

/*

* 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(TU,Y,TY,Z,TZ),
    t(T0,W,TX,X,t(TU,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 jatkuu yli kalvorajojen.)

Esimerkkinä Prologin hyödyllisyydestä etsintäongelmissa olkoon lauselogiikan kaavojen toteutumisongelma:

- Syötteenä saadaan jakamattomista atomikaavoista P_i konnektiiveilla 'ja', 'tai' ja 'ei' yhdistelty kaava.

Esimerkiksi

$$\underbrace{(\text{sataa tai paistaa})}_{P_1} \quad \underbrace{\text{muttei}}_{\text{eli "ja ei"}} \quad \underbrace{\text{sada}}_{P_1}.$$

- Kysytään milloin se on totta.

Esimerkiksi silloin kun ei sada mutta paistaa.

Eli kun P_1 on valetta mutta P_2 totta.

- Riittää kirjoittaa induktiivinen *totuusmääritelmä* rekursiivisena Prolog-ohjelmana:

"Kaava ' F ja G ' on tosi jos sekä F että G on tosi", jne.

- Kun atomikaavan P_1 yhteen esiintymään laitetaan arvo tosi niin myös sen toinen esiintymä tulee kiinnitetyksi.

```

/*
onko(Kaava,Totuusarvo) on totta kun

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

2. Lauselogiikan Kaava on esitetty seuraavasti:

    a. Jakamattomat atomikaavat P_i on esitetty termeinä x(V_i)
       missä V_i on oma Prolog-muuttuja kullekin i.

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

    c. Disjunktiot 'F\/G' eli 'F tai 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).

```

IV.4 Kirjastopalveluita

Tutustutaan Prolog-standardin peruspalveluihin listankäsittelyyn, aritmetiikkaan, samastukseen, negaatioon sekä syöttöön ja tulostukseen [B01,§2.7,3.1-3.2,3.4,5,6,7.2-7.3]. 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*: [täma,on,listavakio].

Tyhjä lista on siis [].

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

Listoja voi käsitellä kalvojen IV.5.1 *listahahmoilla*:

$$[A_1, \dots, A_k | D]$$

on hahmo sellaiselle listalle, jossa on ainakin k alkioita.

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 . alkioita 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-kirjastofunktioiden cd^{k-1} ar ja cd^k r.

Niillä voidaan myös *rakentaa* listoja lukemalla ne toisin päin: muodosta uusi lista lisäämällä listan D alkuun alkioita A_k, \dots, 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 derivaattatermien avulla.

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

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 laskettuun 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 aritmeettisiä 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.5 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"), jota käytettiin jo kalvojen IV.4.1 esimerkissä. (On myös $==/2$ eli "täysin samat ilman samastustakin".)
- $\neq/2$ luetaan "eivät samastu". (On myös $\neq/2$.)

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

Prolog tarjoaa myös (jo kalvoilla IV.3 nähdyn) *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:

$$f(t_1, \dots, t_k) == [f, t_1, \dots, t_k]$$

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

IV.4.4 Negaatiosta

Kalvojen IV.3 sukupuuesimerkissä olisimme halunneet kirjoittaa säännöt "mies on terve (eli `well`) jos hän *ei* ole sairas (eli `ill`)". Kuitenkin kirjoitimme nämä faktat erikseen.

Syynä on se, että *Prolog-kielen negaatio (eli "ei...") on jotakin muuta kuin odottaisi*. Tähän palataan myöhemmin tutustuttuamme kielen merkitysoppiin tarkemmin.

Prolog-kielessä on negaatio `not(p)` missä p on kiellettävä alimaali. Sukupuuesimerkin sääntö olisi siis `well(X) :- not(ill(X))`.

(Voi kirjoittaa myös ilman sulkuja `not ill(X)`.)

Tämä `not/1` on eräänlainen korkeamman kertaluvun predikaatti, koska se ottaa argumentikseen toisen suoritettavan eli todistettavan faktan p . Prolog on kuitenkin "filosofialtaan" ensimmäisen kertaluvun kieli, jossa on vain termejä ja niiden predikaatteja. Omien korkeamman kertaluvun laajennusten ohjelmointia olisi siis syytä välttää "eleganssisyistä".

Prolog-negaatio $\text{not}(p)$ on *negaatio epäonnistumisena (Negation as Failure)*: yritetään osoittaa p , ja jos tämä ei onnistu, niin silloin katsotaan, että on osoitettu $\text{ei-}p$.

Negaatio epäonnistumisena *toimii kun p ei sisällä vapaita muuttujia sillä hetkellä kun kutsu $\text{not}(p)$ tehdään.*

Muuten tulos saattaa olla intuitiomme vastainen, joten tästä negaationkäyttöperiaatteesta kannattaa pitää kiinni! Siis: *tuota ensin muuttujan X lopullinen arvo ja vasta sitten testaa sitä arvoa negatiivisella ehdolla $\text{not}(q(X))$!*

Monissa Prolog-toteutuksissa negaation nimi onkin $\backslash+$ kirkostamaan sitä, ettei kyseessä olekaan "oikea" (eli kielellisen intuitiomme mukainen) negaatio.

IV.4.5 Haun ohjailusta

Kalvojen IV.4.4 negaatio epäonnistumisena voitaisiin ohjelmoida predikaattina

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

- `call(P)` tulkitsee muuttujan `P` arvon faktana, ja tekee vastaavan aliohjelmakutsun.

Joissakin toteutuksissa riittääkin kirjoittaa suoraan `P`.

Vertaa Scheme-kirjastofunktio `eval` kalvoilla II.9.2.

- Huutomerkki `'!'` eli *katkaisupredikaatti* (`Cut`) ohjaa Prolog-ohjelman suoritusta eli todistuksen hakua.

Kalvojen IV.3 aliohjelmatulkinnessa se voidaan lukea "jos suoritus peruuttaa **taaksepäin** yli aliohjelmassa nimeltä q esiintyvän huutomerkkin, niin suoritus putoaa suoraan ulos aliohjelman q alareunasta yrittämättä sen muita vaihtoehtoja".

- Nyt huutomerkki sanoo siis ”P onnistui yhdellä tavalla, älä turhaan etsi muita tapoja, tämä tapa kelpaa minulle”.

Huutomerkillä voi siis *parantaa tehokkuutta* kertomalla järjestelmälle, että nyt on löydetty se tulos, joka haluttiin — etsintä voidaan tältä osin lopettaa. Tähän palataan myöhemmin.

- `fail` epäonnistuu aina (eli sillä ei ole määritelmää).

Negaation määritelmän lisäksi `fail` on hyödyllinen esimerkiksi silloin, kun Prolog-ohjelman täytyy tehdä sivuvaikutuksenaan jotakin. Tähän palataan myöhemmin tietokantojen yhteydessä.

Siis `not`-predikaattimme luetaan seuraavasti:

” Jos aliohjelmakutsu `P` onnistuu, niin rajoita `not(P)` siihen ratkaisuun, jota ei ole.

Muuten `not(P)` onnistuu.”

Siispä se on ”negation as failure”.

IV.4.6 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 syötteen ja tulostuksen oikea käyttö vaatii Prolog-suoritusjärjestyksen tuntemista.

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 (pisteinen) pysyy luettuna.

Jos X onkin termi u , niin t luetaan (pisteinen) ensin, ja sitten yritetään samastaa t ja u . Jos samastus onnistuu, niin kutsukin onnistuu; muuten kutsu epäonnistuu – mutta molemmissa tapauksissa t on luettu (pisteinen)!

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-kalvoja II.9.9) lukeekin `read`illa (vertaa Scheme-kalvoja II.9.8) termejä, ja liittää ne nykyisen ohjelman loppuun.

IV.5 Semantiikka

Logiikkaohjelmoinnissa ohjelmien merkitystä voidaan kuvailla kolmesta eri näkökulmasta:

Suorituslähtöisesti muiden ohjelmointikielten tapaan [B01,§2.4].

”Mitä askeleita ohjelma tekee?”

Kalvojen IV.2 ohjelma **sum** siirtää toisen argumenttinsa kolmanteen, ja sen päälle kaikki ensimmäisen *s*-funktioit.

Näin tehtiin kalvoilla IV.3. Kalvojen III tulkki on tämän näkökulman toteutus.

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.5.1 Suoritussemantiikka

Tarkastellaan predikaatin p/k määritelmää n säännöllä:

$$\begin{aligned} p(t_{(1,1)}, \dots, t_{(1,k)}) & :- q_{(1,1)}, \dots, q_{(1,m_1)}. \\ & \vdots \\ p(t_{(n,1)}, \dots, t_{(n,k)}) & :- q_{(n,1)}, \dots, q_{(n,m_n)}. \end{aligned}$$

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

Jokaisessa haarassa $1 \leq i \leq n$ termit $t_{(i,1)}, \dots, t_{(i,k)}$ määräävät *hahmon* johon aliohjelmakutsun $p(\tau_1, \dots, \tau_k)$ argumenttien τ_1, \dots, τ_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 $\tau \theta$) vastineen.

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

Etsitään ensimmäinen mahdollinen haara i , eli sellainen jolla operaation

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

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, \dots

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

Entä jos jokin seuraava kutsu $q_{(i,j+1)}$ ilmoittaaakin, 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 $p(\tau_1, \dots, \tau_k)$ 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 $p(\tau_1, \dots, \tau_k)$ ei pysty laajentamaan sidontaa θ ”. Tämä aiheuttaa peruutusta vuorostaan kysyjässä.

Vastaavasti myös kysyjältä voi tulla pyyntö ”vaikka kutsu $p(\tau_1, \dots, \tau_k)$ laajensikin sidontansa θ onnistuneesti sidonnaksi θ_{m_i} haaralla i , ei tulos sopinutkaan myöhempään laskentaan, 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.5.2 Mallisemantiikka

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

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

(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 $\tau_1, \dots, \tau_k \in U_L$ (missä $k > 0$), niin myös $f(\tau_1, \dots, \tau_k) \in U_L$.

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

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

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

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 saamalla arvoilla*.

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

Haluamme näet pitää totena *ehtolauseetta*

jos q_1 **ja** \dots **ja** q_n **niin myös** p

jonka loogikko kirjoittaisi *implikaationa*

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

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 deklaratiiivinen 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.5.1 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: $p(o, s^b o, s^b o) \in M_{\text{sum}}$ kaikilla $b \in \mathbb{N}$.

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

Minimaalisuudesta: muuta ei mallissa M_{sum} ole.

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

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

$a = 0$: Seuraa suoraan faktasta.

$a > 0$: Seuraa induktio-oletuksesta

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

ja siihen sovelletusta säännöstä.

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

Miten ρ ilmestyi joukkoon M_{sum} ?

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

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

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

Minimaalisuuden nojalla *muuta tapoja ei ole*.

IV.5.3 Todistussementtiikka

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

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

Kalvojen IV.5.2 mukaan Prolog-ohjelmalla L on *pienin* malli M_L joka on samalla sen merkitys.

Kysymykseen "Onko maalifakta $p \in M_L$?" – eli kalvojen IV.5.1 aliohjelmakutsuun p – voi siis yrittää vastata *todistamalla* p väitteistä L .

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

Kalvojen IV.5.2 *todistuksen etsinnän* epätäydellisyyteen se ei auta: yleisessä tapauksessa on mahdotonta tunnistaa millään algoritmilla, että todistusta *ei* ole.

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

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

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

sivuehto: p_1 ja q'_1 ovat samat

johtopäätös:

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

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.5.2.
- 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.5.2 sääntö

$$p :- q_1, \dots, q_m.$$

tulkittiin loogisesti

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

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

$$p.$$

on

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

eli

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

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

$$\forall \vec{X}. q_1 \wedge \dots \wedge q_m \rightarrow \text{epätosi}$$

eli

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

(missä ' \neg ' on negaatio eli "ei"), eli

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

(missä ' \exists ' on "on olemassa" -kvanttori).

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

Tämä on *maali*

$$?-q_1, \dots, q_m.$$

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: $p: -q_1, \dots, q_m.$

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

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

Ensin säännön muuttujat on kuitenkin nimetty erilleen maalin muuttujista.

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

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.5.1 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.5.1 suoritussemantiikassa, jossa sääntöjä käytetään "takaperin" eli **niin**-osasta **jos**-osaan.

Kalvojen IV.5.2 mallisemantiikassa sääntöjä käytettiin "etuperin". Vastaava todistussemantiikka saadaan lukemalla Prolog-lause

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

(ei enää totuudesta puhuvana väitteenä vaan) todistuvuudesta puhuvana päättelysääntönä

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

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

Muuttujat tulkitaan kalvojen IV.5.1 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.5.1 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 $i = 1, \dots, n$.
- Oletukset yritetään todistaa kirjoitusjärjestyksessä $j = 1, \dots, m_i$.

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

Sehän on ohjelmointikielen suorituskoneen abstrakti määrittely ”vaihtoehtottomana logiikkana”! Esimerkkinä kalvojen II.5 Scheme-ydinkielen määrittely päättelysäännöillä.

Vastaava malliteoria on kielen denotationaalinen eli tarkoitesemantiikka.

IV.6 Negaatio tarkemmin

Kalvojen IV.5 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 kalvojen IV.4.4 negaatio epäonnistumisena [B01,§5.3-5.4]: vartalossa voi esiintyä alimaali

$\text{not}(p)$

jonka haluttu luenta on ”väite p ei päde”.

Seuraavaksi tarkastellaan mikä on sen todellinen luenta kalvojen IV.5.1—IV.5.3 näkökulmista.

IV.6.1 Suoritussemantiikassa

Kalvojen IV.5.1 suoritussemantiikassa kutsu $\text{not}(p)$ tehdään seuraavasti:

1. Yritetään suorittaa negation sisällä oleva kutsu p nykyisellä sidonnalla θ_i .
2. Jos sisäkutsu p epäonnistuu, niin koko kutsu $\text{not}(p)$ onnistuu.

Sidontana pysyy θ_i .

(Eihän epäonnistunut p voinut kasvattaa sidontaa θ_i . Myöskään myöhempi peruutus takaisin kutsuun $\text{not}(p)$ ei tuota uusia vaihtoehtoja.)

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

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.5.3 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?"

Tämän vuoksi 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.6.2 Mallisemantiikassa

Kalvojen IV.6.1 negatoitujen muuttujien ongelmaa voi yrittää selittää kalvojen IV.5.2 mallisemantiikalla.

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

Negatiivinen muuttujaton $?- \text{not}(q)$. 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.6.3 Todistussementtiikassa

Kalvojen IV.5.3 todistussementtiikassa kalvoilla IV.6.1 valittu negaation toteutus on uusi päättelysääntö

Väite $\text{not}(p)$ 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.7 Haun ohjailu

Kalvojen IV.4.4 Prolog-negaatio toteutettiin kalvoilla IV.4.5 ohjailemalla Prolog-hakukoneen toiminta kirjastopredikaateilla `!/0` ja `fail/0`.

Tarkastellaan katkaisupredikaattia `!/0` tarkemmin. Olkoon ratkaistavana maali

$$?-g, \dots$$

ja kokeiluvuorossa ohjelman sääntö

$$p: -q_1, \dots, q_m, !, r_1, \dots, r_n.$$

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, \dots).

- Suoritus etenee normaalisti eteenpäin yli katkaisukohtaan vartalon loppuosaan r_1, \dots, r_n sidonnan θ_m muuttumatta.

- Sivuvaikutuksena etsintäkone hävittää kaikki kutsun g jäljellä olleet vaihtoehtoiset ratkaisutavat:
 - vartalon alkuosan q_1, \dots, q_m
 - loput ohjelmasta.
- Jos suoritus myöhemmin peruuttaa taaksepäin yli katkaisukohtaan, 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.4.5 negaation toteutus on esimerkki säännöstä (uutta) muotoa

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

”**Jos** P onnistuu **niin** epäonnistuu **muuten** onnistuu.”

Tähän on tarjolla syntaktista sokeria muodossa
 $P \rightarrow \text{fail} ; \text{true}$.

Tässä väliin kirjoitettu funktori $\rightarrow/2$ on
jos. . . niin. . . .

Väliin kirjoitettu funktori $;/2$ taas on **tai**.

- + 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 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

```
put21(c(T0,X,TX),Y,TY,  
      t(T0,X,TX,Y,TY)).  
put21(b(T0,X,TX),Y,TY,  
      b(b(T0,X,TX),Y,TY)).  
put21(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

```
put21(c(T0,X,TX),Y,TY,  
      t(T0,X,TX,Y,TY)):-  
      !.  
put21(T,Y,TY,b(T,Y,TY)).
```

punaisella katkaisulla.

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

Toinen tapa ohjailta hakua on kirjoittaa sellaisia Prolog-ohjelmia, joissa sääntöjä valitaan sen mukaan, millaista *tyyppiä* eri muuttujien arvot nyt ovat.

Prolog on samaan tapaan piilevästi tyyplitetty kieli kuin Scheme (kalvot II.4).

Tyyppien tutkimiseen on seuraavat kirjastopredikaatit [B01,7.1]:

`atom(X)` kertoo, onko *X* jakamaton *atomi* —
Prolog-vastine kalvojen II.4.3
Scheme-symboleille.

Kalvojen IV.4.1 tyhjä Prolog-lista `[]` on myös atomi.

`integer(X)` kertoo, onko *X* kokonaisluku
(kalvot IV.4.2).

`float(X)` kertoo, onko *X* liukuluku.

`number(X)` kertoo, onko *X* kokonais- tai liukuluku.

`atomic(X)` kertoo, onko *X* atomi tai luku.

`compound(X)` kertoo, onko X rakenteinen termi, eli muotoa $f(\dots)$.

Myös kalvojen IV.4.1 epätyhjät Prolog-listat ovat rakenteisia. Niissä funktorina f on sama `'./2` joka nähtiin jo kalvojen II.8 Scheme-listoissa.

`var(X)` kertoo, onko X tällä hetkellä sellainen muuttuja, jota ei vielä ole sidottu.

Jos X on sidottu, mutta vain johonkin toiseen vapaaseen muuttujaan Y , niin tätä yhteyttä " X ja Y ovat vapaat vapaat mutta samat" ei vielä pidetä sidontana.

Näin `var(X)` kertoo esimerkiksi, voiko X toimia tulospositiona — jolloin se ei voi olla syötepositio. . .

`nonvar(X)` kertoo käänteisen ominaisuuden: joko X ei ole muuttuja tai sitten se on sellainen muuttuja, joka on tällä hetkellä sidottu johonkin "oikeaan" arvoon, eli atomiin, numeroon tai rakenteiseen termiin.

IV.8 Prolog ja tietokannat

Kalvojen III yksinkertaisessa Prolog-tulkissa säännöt ja faktat 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.

Lisäksi useissa Prolog-toteutuksissa on mahdollisuus käyttää jotakin ulkoista relaatiotietokantaa muuttujattomien faktojen tallennukseen. Esimerkiksi SWI Prolog tarjoaa Open DataBase Connectivity (ODBC) -standardin mukaisen kirjaston.

Prolog on siis hyvä korkean tason kieli sellaisiin tiedonhallintatehtäviin, joissa tarvitaan relaatio-operaatioiden lisäksi päättelykykyä.

IV.8.1 Sisäinen tietokanta

[B01,§7.4]

$\text{assert}(C)$ (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 $\text{assert}/1$ ei anna uusia "vastauksia", mutta sen lisäämä C jää tietokantaan.

$\text{assertz}(C)$ on kuin $\text{assert}/1$ paitsi että C lisätäänkin edellisten *perään*.

$\text{assert}(C)$ 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 $\text{assert}/1$ on $\text{assertz}/1$.)

`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.

Tässä on *toisto epäonnistuen* (failure-driven loop): tietokannalla on *tila joka muuttuu sivuvaikutuksena*, joten `fail`-epäonnistumisen aiheuttama peruutus toistaa tätä sivuvaikutusta.

Myös kalvojen IV.4.6 tulostus tapahtui sivuvaikutuksin.

Prolog-standardissa (mutta ei välttämättä kaikissa toteutuksissa)

- *consultoitu* tai kirjastopredikaatti on *staattinen* eikä muokattavissa suoritusajana ilman edeltävää kutsua `dynamic/1`.

Intuitiivisesti, suorituskelpoinen ohjelmakoodi pidetään sittenkin erillään vapaasti muokattavasta tietokantadatasta, kunnes erikseen julistetaan aikomus muokata jotakin osaa koodista.

- vaikka predikaatin määrittelyä muutettaisiin 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, ja sen tulos vaikuttaa jo saman päivityksen seuraaviin askeliin.)

IV.8.2 Datalog

Kalvojen IV.8.1 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”:

$pomo(P, A) :- johtaa(P, Y), töissä(A, Y).$

$pomo = \pi_{1,3}\sigma_{2=4}(johtaa \times töissä)$

```
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 ($\sigma_{...=...}(... \times ...)$).
- Vartalossa esiintyvän muuttujan esiintyminen myös päässä vastaa projektiota ($\pi_{...}$).

Jos taas (nimetön) muuttuja esiintyy päässä muttei vartalossa, on sille vaikea keksiä tietokantatulkintaa; siksi sellaisia ei nyt sallita.

Samaa ”relaationaalista” ajattelua voi käyttää usein myös Prolog-ohjelmissa: vartalon kukin predikaatinkutsu hakee vastaavasta (ehkä intensionaalista) 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.

Samana predikaatin eri säännöt vastaavat yhdistettä:

käläkättäjä(X) :- töissä(X, nokia).

käläkättäjä(X) :- töissä(X, ericsson).

$$\text{käläkättäjä} = \pi_1 \sigma_{2=\text{nokia}} \text{töissä} \\ \cup \pi_1 \sigma_{2=\text{ericsson}} \text{töissä}$$

```
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.4.4 suositeltiin, että Prolog-negaatiota

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

sovellettaisiin vain muuttujattomiin p .

Datalogissa riittää, että kaikki negatoidussa p esiintyvät muuttujat esiintyvät samassa vartalossa myös negatoimattomina (ennen itse negaatiota).

Silloin negaatio vastaa joukkoerotusta:

$$\text{työläinen}(X) :- \text{töissä}(X, Y), \text{not}(\text{johtaa}(X, Y)).$$
$$\text{työläinen} = \pi_1(\text{töissä} \setminus \text{johtaa})$$

```
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)":

$\text{omistaa}(A,B) :- \text{valvoo}(A,B).$

$\text{omistaa}(A,B) :- \text{valvoo}(A,C), \text{omistaa}(C,B).$

Nyt haetaan relaatioiden välisen yhtälön

$$\text{omistaa} = \text{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\text{valvoo} \times \text{omistaa})$$

ratkaisua $\text{omistaa} = \dots$

Kaikista ratkaisuista halutaan *pienin* koska

- seurataan kalvojen IV.5.2 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 := $\text{valvoo} \cup \pi_{1,4}\sigma_{2=3}(\text{valvoo} \times \text{ennen});$

until ennen = sitten

(Parempia mutta mutkikkaampia algoritmeja on!)

Tämä operaatio on itse asiassa ekstensionaalisen relaation *valvoo* *transitiivinen sulkeuma* (transitive closure, TC): $\langle a, b \rangle \in \text{omistaa}$ jos ja vain jos on ketju $\langle a, c_1 \rangle \in \text{valvoo}, \langle c_1, c_2 \rangle \in \text{valvoo}, \dots, \langle c_k, b \rangle \in \text{valvoo}$.

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.6).

Patologinen esimerkki on

$p: \neg \text{not}(q)$.

$q: \neg \text{not}(p)$.

jossa negaatio osallistuu rekursioon.

Kalvojen IV.5.2 mukaan \emptyset ei vielä ole malli: $\text{not}(q)$ on totta mutta p ei.

Hieman suurempi $\{p\}$ on jo malli: $\text{not}(p)$ ei ole totta.

Mutta myös $\{q\}$ on malli, eikä siis ole vain yhtä pienintä mallia.

Datalog-ajattelussa saadaan yhtälöpari

$$p = \text{full} \setminus q$$

$$q = \text{full} \setminus p$$

missä $\text{full} = \{\langle \rangle\}$ 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
\vdots	\vdots	\vdots

Myös vastaava Prolog-suoritus jää ikuisen silmukkaan.

Prolog ja datalog ovat uusien *deduktiivisten* eli (epätriviaalia) päättelyä suorittavien tietokantojen keskeisiä perusasioita.

IV.8.3 Vastausjoukot

Kalvojen IV.8 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* θ . 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.

$$L := \text{SELECT } X \\ \text{FROM } P$$

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 $L = []$.

$\text{bagof}(X, P, L)$ 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 .

```
 $L_a$  := SELECT X
        FROM P
        WHERE Y = a
```

Nykyinen a löytyy muuttujan Y arvona. Peruutus siirtyy seuraavaan a , tai epäonnistuu, jos niitä ei enää ole.

Notaatiolla $\text{bagof}(X, Y^P, L)$ voi määrätä muuttujan Y käyttäytymään sittenkin kirjastopredikaatin `findall/3` tapaan.

$\text{setof}(X, P, L)$ on muuten kuin `bagof/3`, mutta L on järjestetty ja sisältää kunkin X vain kerran.

```
 $L_a$  := SELECT DISTINCT X
        FROM P
        WHERE Y = a
        ORDER BY X
```

Kirjastossa on termien vertailuun predikaatti `@</2` (sekä `@=<`, `@>`, `@>=`, `==` ja `\==`).

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.Clocksinn&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.): Revised⁵ Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11(1), 1998 **ta** *ACM SIGPLAN Notices* 33(9), 1998.

- L87 J.W.Lloyd: *Foundations of Logic Programming. Second, Extended Edition.* Springer-Verlag, 1987.
- P02 B.C.Pierce: *Type Theory and Programming Languages.* MIT Press, 2002.
- S98 S.Slade: *Object-Oriented Common Lisp.* Prentice Hall, 1998.
- SS86 L.Sterling&E.Shapiro: *The Art of Prolog.* MIT Press, 1986.
- S97 B.Stroustrup. *The C++ Programming Language. Third Edition.* Addison-Wesley, 1997.
- T99 S.Thompson: *Haskell: The Craft of Functional Programming. Second Edition.* Addison-Wesley, 1999.