III.3 Sääntöjen esitys

Esitetään sääntö

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

listana

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

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

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

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

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

Tämän vuoksi ulointa funktoria kutsutaankin *predikaatiksi*: faktaa vastaavan väitelauseen verbiksi (*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)kant* josta kysellään nykyiseen tilanteeseen soveltuvia sääntöjä.

Valitaan tälle kannalle yksinkertaisin mahdolline esitys: lista kirjoitusjärjestyksessä.

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

III.4 Ratkaisun etsintä

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

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

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

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

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

Sääntöjä käytetään tässä *takaperin*: Haluan *todistaa* uuden faktan (pään, **niin**-osan), joten yritän todistaa sen ehtona olevat faktat (vartalon, **jos**-osan).

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

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

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

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

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

Samalla nähdään että jokaisella käytöllä haluta
z koska tehdään ketjua

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

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

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

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

Ratkaisun etsijää ohjelmoitaessa on tehtävä 2 päätöstä:

- *Miten seuraava ratkaistava alimaali valitaan?*

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

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

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

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

  Maali on siis *pino* ratkaisua odottavia alimaaleja.

  Sääntöä ratkotaan siis sen *vartalon kirjoitusjärjestyksessä*.

  Silloin maalit ovat esimerkiksi

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

  ja etsintä päättyy onnistumiseen eli tyhjään maaliin.

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

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

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

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

  + Tehokkaampi etsintäohjelma.

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

  − Sääntöjen järjestys tulee tärkeäksi.

    Väärällä järjestyksellä etsintä voi jäädä *ikuiseen silmukkaan*!

  Tekoälyssä tarkastellaan tätä (ja) fiksumpia etsintäperiaatteita.

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

(**esi-isa** x y)

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

Etsintä voi siis *epä*onnistua säännön valinnassa, päätyä umpikujaan ja peruuttaa. Kalvojen II.8.8 laiskat listat mahdollistavat tällaisen etsinnän ohjelmoimisen *listana onnistumisia*:

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

Vastauksena palautetaan siis pari

(seuraava ratkaisusidonta . lupaus muista pareista)

tai () jos ratkaisusidontoja ei enää löydy.

Lisätään vielä välineet etsinnän edistymisen seuraamiseen.

(Listaus jatkuu yli kalvo

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

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

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

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

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

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

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

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

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

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

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

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

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

; A given <goal> is solved by a given <guide> as follows.
(define rule:solve
  (lambda (<guide>
           <goal>)
    (define engine
      (lambda (guide    ; The <rule>s not yet tried for this goal.
               goal     ; This goal.
               binding  ; Collected binding.
               choices) ; Delayed other solutions.
        (cond ((goal:none? goal)
               ; Solved! Report collected binding and other soluti
               (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> inste
                       (engine (guide:rest guide)
                               goal
                               binding
                               choices))))))))
      ; Initially neither bindings nor other solutions.
      (engine <guide>
              <goal>
              binding:empty
              (delay '())))))

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

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

(Listaus jatkui yli kalvorajojen.)

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

Samalla saadaan esimerkki kalvojen II.9.7 tulost ja kalvojen II.9.8 syötekirjastofunktioiden alkeiskäytöstä.

(Listaus jatkuu yli kalv

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

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

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

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

(Listaus jatkui yli kalvorajan.)

III.5 Järjestelmän käyttöä

Olkoon sukupuu esitetty seuraavina faktoina:

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

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

Niillä voidaan määritellä säännöillä käsitteet

`(mother` x y`)` : x on y:n äiti.

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

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

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

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

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

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) gonzalon 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"

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

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 ka
samoin ajattelevaa tutkijaa löysi toisensa:

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

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

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

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

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

IV.2 Perussyntaksi

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

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

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

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

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

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

Funktori f kirjoitetaan **atomi**na eli

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

Sääntö kirjoitetaan muodossa

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

listamuodon

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

sijasta.

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

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

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

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

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

Esimerkiksi Prolog-ohjelmassa

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

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

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

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

- Faktan `Y` on eri kuin säännön `Y`.
- Säännön pään `X` on sama kuin vartalon `X`.

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

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

IV.3 Prolog vs. perinteinen ohjelmoin

Lähestytään Prolog-ohjelmointia vertaamalla sit aiemmin tunnettuihin ohjelmointitapoihin:

Aliohjelmatulkinta: Predikaatin määritelmä

nimi(syötehahmo(t)$_1$,tuloshahmo(t)$_1$):-
    aliohjelmakutsu$_{1,1}$,...,kutsu$_{1,n_1}$.
nimi(syötehahmo(t)$_2$,tuloshahmo(t)$_2$):-
    aliohjelmakutsu$_{2,1}$,...,kutsu$_{2,n_2}$.
⋮
nimi(syötehahmo(t)$_m$,tuloshahmo(t)$_m$):-
    aliohjelmakutsu$_{m,1}$,...,kutsu$_{m,n_m}$.

määrittelee samannimisen aliohjelman *tapauksit*

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(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 (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 e tavoilla ja "tuloksilla".

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

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

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

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

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

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

  Onnistuessaan kutsu$_{i,j}$ laajensi saamansa sidonnan $\sigma_{i,j}$ sidonnaksi $\sigma_{i,j+1}$, jonka 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 kutsu$_{i,j-1}$. Palautetaan sidonnaksi se 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 u niin tämä sen kutsu sidonnalla σ_{nimi} epäonn

- Siis Prolog-ohjelman suorituksessa on 2 vaihdetta:

  eteenpäin rungossa oikealle kohden päättä 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 eteenpäin. Näin yritetään jotakin toista vaihtoehtoa umpikujaan vieneen sijasta.

  Nämä vaihteet näkyvät kalvojen III Scheme-toteutuksessa seuraavasti:

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

  Taaksepäin mennään lunastamalla viimeisi 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 tot 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 vä tietoa vasemmalta oikealle, kuten `Z` predikaatissa `ancestress/2`.

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

- Kaksi eri alaviivaa tarkoittavat siis *eri* mielenkiinnottomia asioita, vaikka ne olisiva samassa lauseessa:
  `ancestress(X,Y):-mother(X,_),ancestress(_,Y`
  olisi väärin, koska sen mukaan "`X` on `Y`:n esi 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).
```


Kirjallisuutta

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

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

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

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

CM87 W.F.Clocksin&C.S.Mellish: *Programming Prolog. Third, Revised and Extended Edit* Springer-Verlag, 1987.

CM98 G.Cousineau&M.Mauny: *The Functional Approach to Programming*. Cambridge University Press, 1998.

DD98 C.J.Date&H.Darwen: *Foundation for Object/Relational Databases: The Third Manifesto.* Addison Wesley, 1998.

DEC96 P.Deransart,A.Ed-Dbali&L.Cervoni: *Prolog: The Standard. Reference Manual.* Springer-Verlag, 1996.

D96 R.K.Dybvig: *The Scheme Programming Language: ANSI Scheme. Second Edition.* Prentice Hall, 1996.

H92 D.Harel: *Algorithmics. Second Edition.* Addison Wesley, 1992.

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

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

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

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

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

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

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

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