

D:n ja Erlangin tunnukset sekä kontrollirakenteet

Hansi Keijonen, Jari Koskinen, Eero Laine

6. helmikuuta 2013

1 Tunnusten näkyvyysalueet ja sidonta

(D ja Erlang vierekkäin)

lohkorakenne, litteä vs syvä

sidonta, staattinen vs dynaaminen

ensimmäisen toisen ja kolmannen luokan arvot

sulkeumat

esimerkit

em. vertailu

2 Kontrollin ja laskennan ohjaus - Erlang

Jotta voidaan ymmärtää Erlangin kontrollirakenteita, pitää ensimmäisenä selvittää hahmontunnistuksen mekanismi. Hahmontunnistus perustuu siihen, että funktioon määritellään useita funktiokutsuja siten, että jokaisessa kutsussa parametrit ovat lukumäärältään samat, mutta ovat arvoiltaan erilaiset. Ts. kun funktiota kutsutaan tietyllä parametrilla, käydään järjestyksessä läpi eri funktion kutsuvaihtoehdot ja kun ensimmäinen parametriin sopiva kutsuvaihtoehto löytyy, toimitaan sen määrittelyn mukaan. Helpoimmin asia selviää esimerkin avulla:

```
1: tervehdi(mies, Nimi) -> io:format("Terppa, herra %s!", [Nimi]);
2: tervehdi(nainen, Nimi) -> io:format("Terppa, rouva %s!", [Nimi]);
3: tervehdi(_, Nimi) -> io:format("Terppa %s!", [Nimi]) .
```

Kutsutaessa tervehdi-funktiota tervehdi(nainen, "Tuppurainen"), tutkitaan ensiksi, sopiiko parametrit rivin 1 parametrিসointivaihtoehtoon. Koska parametrit olivat epäsojivaiset, yritetään kutsua sojittaa rivin kaksi parametrিসointivaihtoehtoon. Tällä kertaa parametrit sopivat ja ohjelma tulostaa "Terppa, rouva Tuppurainen!". Jos funktiota olisi kutsuttu tervehdi(hermafrodiitti, "Höttölä"), olisi vasta rivin 3 parametrit osuneet oikeaan, ja silloin olisi toimittu tämän rivin määrittysten mukaan. Rivin 3 toinen parametri on universaalimerkki, johon voi sojittaa minkä tahansa arvon.

Valintojen tekemiseen Erlangissa on em. hahmonsovituksen lisäksi mahdollista käyttää if-lauseita, guardeja sekä in case of -rakennetta.

Erlangissa if-rakenne ilmaistaan hieman esim. Javasta ja D:stä poikkeavalla tavalla, jota valotan seuraavassa esimerkissä.

```
1: vertaa(N,M) ->
2:   if N>M -> 1;
3:       N<M -> -1;
4:       true -> 0
5:   end .
```

Esimerkkifunktiossa siis verrataan kahta lukua ja toimitaan eri tavoin sen mukaan, mikä on vertailun tulos. Riveillä 2 ja 3 tutkitaan ovatko N ja M erisuuruiset, mutta rivillä 4 on hieman erikoiselta vaikuttava true, joka vastaa esim. D:n else:ä. Koko if-rakenne päätetään end sanaan ja pisteeseen.

Guardit ovat yksi Erlangin erikoisuus, joka on tuttu myös Haskellista. Guardit ovat keino tehdä hahmonsovituksesta hieman ilmaisuvoimaisempaa. Sen sijaan, että täysi-ikäisyyden osoittava ohjelma toteutettaisiin näin:

```
taysi_ika(1) -> false;
taysi_ika(2) -> false;
taysi_ika(3) -> false;
.
.
.
taysi_ika(16) -> false;
taysi_ika(17) -> false;
taysi_ika(_) -> true;
```

, toteutetaan iän tarkastaminen guardilla

```
taysi_ika(N) when N < 18 -> false;
taysi_ika(_) -> true .
```

Tällä tavoin voidaan jättää kirjoittamatta useita funktiokutsuvaihtoehtoja. Tärkeää on muistaa, että guard-rakenteen tulee aina palauttaa true jollain vaihtoehdolla, muuten kääntäjä ilmoittaa virheestä.

Erlangissa on myös case...of -rakenne, joka mahdollistaa monipuoliset valintarakenteet, johon voidaan yhdistää myös guardeja. Esimerkki selventää rakennetta helpoiten:

```
1: biitsi(Lampotila) ->
2:   case Lampotila of
3:       {celsius, N} when N >= 20, N <= 45 -> 'ihan jees';
4:       {kelvin, N}   when N >= 293, N <= 318 -> 'tieteellisesti jees';
5:       {fahrenheit, N} when N >= 68, N <= 113 -> 'amerikkalaisittain jees';
6:       _ -> 'ei niin jees'
```

Ohjelma siis saa parametrinaan tuplen, jossa on lämpötila-asteikko ja astemäärä. Sen jälkeen `case...of`-rakenteessa riveillä tutkitaan, minkä periaatteen mukaisesti astemäärää tutkitaan. Lopulta guardin avulla pääte-tään, onko parametrina saatu lämpötila sopiva rantaelämään. Mielenkiin-toinen kysymys on se, että mihin `case...of`-rakennetta tarvitaan, koska se on hyvin samanlainen funktiokutsujen hahmonsovituksen kanssa. Valinta kannattaa perustaa henkilökohtaisiin mieltymyksiin.

Erlangissa toistoa ja rekursiota ei voi erottaa toisistaan, koska rekursio on ainoa tapa suorittaa toistorakenteita. Erlangissa ei ole siis `for`- tai `while`-lauseita esim. D:n tai Javan tapaan. Listan läpikäynti Erlangissa perus-tuu kielen ominaisuuteen, että esim. kokonaislukuja sisältävä lista `[1,2,3,4]` voidaan ilmoittaa myös konstruktorioperaattoria `'|'` käyttäen `[1|[2,3,4]]`, eli luku 1 liitetään listaan `[2,3,4]`. Lista voidaan ilmoittaa myös `head`- ja `tail`-funktioita käyttäen `[Head|Tail] = [1,2,3,4]` jolloin `Head = 1` ja `Tail = [2,3,4]`. Jos halutaan käydä lista läpi yksinkertaisesti siten, että laskemme jokaisen listan alkion yhteen, voidaan se Erlangissa tehdä esim. seuraavalla tavalla:

```
1: summa([]) -> 0;
2: summa(Lista) -> head + summa(Tail) .
```

Tässä funktiossa siis hahmonsovituksen avulla toteutetaan rekursiivinen listan läpikäynti. Rivillä 1 on toteutettu alkeistapaus, jossa lista on tyhjä ja jolloin alkioden summa on 0. Rivillä kaksi summataan listan ensimmäi-nen alkio (`Head`) loppulistan (`Tail`) alkioden summan kanssa. Tällä tavoin voidaan tehdä monipuolisia listan läpikäyntejä.

Rekursio on Erlangissa yvin tavanomainen tapa tehdä asioita, semminkin kun muita toistorakenteita ei ole. Perinteinen rekursiolla tehtävä funktio on kertoma $n!$, joka voidaan ilmaista yhtälöparilla

$$n! = \begin{cases} 1 & \text{jos } n=0 \\ n * ((n-1)!) & \text{jos } n>0 \end{cases}$$

Tämä on on helposti ja ulkoasultaan elegantisti toteutettavissa Erlangilla hahmonsovitusta ja rekursiota hyväksikäyttäen:

```
kertoma(0) -> 1;
kertoma(N) when N > 0 -> N*(kertoma(N-1)) .
```

Jälleen aluksi tarkistetaan alkeistapaus, joka on tässä tapauksessa luvun 0 kertoma ja jos tämä rivi ei ota kiinni, suoritetaan seuraavan rivin rekur-siivinen kutsu, jossa parametrilla N kerrotaan rekursiivisesti kutsuttavan `kertoma(N-1)` tulos. Kun rekursiokutsut saavuttavat tapauksen `kertoma(0)`, palautuu tulos pitkin kutsuketjua alkuun. Tämän lähestymistavan huono puoli on se, että tietokoneen muistissa joudutaan pitämään koko ajan jokai-sen kutsun 'välitulos'. Esimerkiksi edellisessä kertomaesimerkissä paramet-rilla 3 laskenta menisi seuraavasti:

```

kertoma(3) = 3*(kertoma(2))
            = 3*(2*(kertoma(1)))
            = 3*(2*(1*(kertoma(0))))
            = 3*(2*(1*1))
            = 3*(2*1)
            = 3*(2)
            = 6

```

Joutuisimme pitämään muistissa kaikki nuo 7 välivaihetta. Tässä tapauksessa tämä ei olisi ongelma, mutta jos parametrina annettaisiin joku todella suuri luku, muistin riittävyys muodostuisi ongelmaksi. Tämän ongelman voi ohittaa häntärekursiolla. Häntärekursiivinen kertomafunktio toteutetaan toteutamalla myös apufunktio hantakertoma(N,Acc) johon otetaan mukaan vielä toinen parametri Acc, johon tallennetaan laskennan välivaiheiden tulokset:

```

kertoma(N) -> hantakertoma(N,1) .

```

```

hantakertoma (0,Acc) -> Acc;
hantakertoma(N,Acc) when N > 0 -> hantakertoma(N-1, Acc*N) .

```

Nyt laskennan kulku on seuraavanlainen:

```

kertoma(3) = hantakertoma (3,1)
            = hantakertoma (3-1, 3*1)
            = hantakertoma (2-1,2*3)
            = hantakertoma (1-1,1*6)
            = 6

```

Tässä tapauksessa joudumme pitämään ainoastaan kaksi lukua muistissa koko laskennan ajan kun normaalissa rekursiossa joudumme pitämään kaikki rekursiokutsuketjun välivaiheet muistissa. Jos unohdetaan se, kuinka paljon luvun 3 ja 1000 000 esittäminen kuluttaa muistia, kertoma(3) ja kertoma(1000000) vaativat saman vakiomäärän muistia.

3 Kontrollin ja laskennan ohjaus - D

3.1 Valinta

If-lause on toteutettu D:ssä melko tyypillisellä tavalla:

```

if (a==b) {
    writeln("A ja B ovat samoja.");
}
else {
    writeln("A ja B eivät ole samoja.");
}

```

Toisin kuin muilla kielillä, D:llä ei ole “tyhjän lauseen” (empty statement) konstruktia. Koodinpätkä

```
if (a == b);
```

ei siis ole D:n hyväksymää. Sen sijaan tulisi kirjoittaa

```
if (a == b) {}
```

Else-lause on aina sidottu lähimpään if-lauseeseen. Esimerkiksi:

```
if (a == b)
  if (b == c)
    writeln("B ja C ovat samoja.");
  else
    writeln("B ja C eivät ole samoja.");
```

Useat laskeutuvat (cascading) if-else -lauseet toteutetaan kuten C-kielessä.

D:n static if -lause on C-kielen #if-rakennetta muistuttava kääntämisaajan (compile-time) selektori. Tullessaan static if -lauseen kohdalle kääntäjä arvioi ohjaavan lausekkeen. Jos lauseke ei ole tyhjä, vastaava koodi käännetään. Muussa tapauksessa ainoastaan mahdollisen else-lause käännetään.

Static if -lauseen testaama lauseke voi olla mikä vain if-testattava lauseke, jota voidaan arvioida kääntämisen aikana. Mahdollisia lausekkeisiin kuuluvat myös funktiokutsut.

Goto-lause toimii D:ssä allaolevalla tavalla:

```
goto <nimi>;
```

D:ssä goto-lauseen käyttöön ei liity paljoa rajoituksia, mikä D:n pääkehittäjä Andrei Alexandrescun mukaan tekee gotosta ongelmallisen työkalun. Goto voi hypätä eteen- tai taaksepäin ja sisään ja ulos lauseista. Symbolin <nimi> tulee kuitenkin olla näkyvässä funktiossa, missä gotoa kutsutaan.

3.2 Toisto

While- ja do-while -lauseet ovat hyvin samankaltaisia, kuin vaikkapa Java-kielessä. Myös myös for- ja foreach-lauseet noudattavat Javasta tuttua muotoa, mutta foreach-lauseen voi ilmaista D:ssä myös for-lauseella allaolevalla tavalla:

```
{
  auto __n = <lauseke1> ;
  auto <symboli> = true ? <lauseke1> : <lauseke2>;
  for ( ; <symboli> < __n; ++ , symboli ) <lause>
}
```

___n on kääntäjän generoima symboli, joka ei voi koskaan sekaantua muihin symboleihin. SELITÄ LISÄÄ??!! Uloimmat aaltosulkeet varmistavat, että koko rakenne pysyy yhtenä lauseena. ?-merkkialkiolla ei ole suorituseaikaista tarkoitusta, mutta se estää sekaannusta mm. eri tarkkuuksia edustavien lukujen välillä.

Taulukoiden käsittelyyn foreach-lauseesta on olemassa allaoleva muoto:

```
foreach (<symboli> ; <lauseke>) <lause>
```

Lausekkeen tulee olla taulukko, SLICE? tai käyttäjän määrittelemä tyyppi. Kun <lauseke> on arvioitu kerran, viite siihen tallennetaan väliaikaisesti. <symboli> sidotaan vuorotellen jokaiseen taulukon elementtiin, ja <lause> suoritetaan. <symboli>:n yhteyteen voi määritellä myös sen tyypin.

3.3 Rekursio

Rekursiosta on hyötyä D:ssä alustaessa monimutkaisia immutaabeleita olioita. Alla on esimerkki taulukosta alustetusta listasta:

```
class List {
    private int arvo;
    private List seuraava;
    this(int[] data) immutable {
        pakota(data.length);
        arvo = data[0];
        if (data.length == 1) return;
        next = new immutable(List)(data[1 . $]);
    }
}
```

Alustaakseen listan lopun konstruktori soveltaa itseensä rekursiota lyhyemmällä taulukolla. Listan alustaminen silmukassa ei kääntyisi, koska listaa ei voi lukea samalla, kun sitä luo. Rekursion käyttäminen ratkaisee ongelman. HIRVEÄTÄ SOOPAA, TEE TÄLLE JOTAIN