KOIRAMAINEN OHJELMOINTIKIRJANEN

Kanaherkun tuoksuinen johdatus funktionaaliseen ohjelmointiin

Juuso Vuorinen

Juuso Vuorinen

KOIRAMAINEN OHJELMOINTIKIRJANEN

Kanaherkun tuoksuinen johdatus funktionaaliseen ohjelmointiin

Ideal Learning Oy
Tampere 2020

Tämä teos on lisensoitu Creative Commons Nimeä-EiKaupallinen-JaaSamoin 4.0 Kansainvälinen -lisenssillä.

Sisällys

Luku 1. Johdanto 7

ATK:n paluu 7

Algoritminen ajattelu ja reseptit 7

Ylhäältä alas tulkittavista ohjeista yhteistoiminnan kuvaamiseen 8

Resepteistä tiedonkäsittelykoneisiin 9

"Käymme yhdessä ain, käymme aina..." 10

Tyylillä on väliä 11

Matematiikan oppimisesta on paitsi iloa myös hyötyä 13

Ohjeita kirjan lukijalle 13

Luku 2. Koneet 17

Tiedon käsittelyn kuvaamista ei voi automatisoida 17

Funktio tiedonkäsittelykoneena 17

Ensimmäinen koneemme 18

Tuotantoketju ja koneiden keskinäiset riippuvuudet 22

Ei miten, vaan mitä 24

Funktion arvon selvittäminen 25

Luku 3. Raaka-ainetta koneeseen 28

Raakaa-aineita moneen lähtöön 28

Usean eri raaka-aineen annosteleminen koneelle 28

Tyypillisiä ohjelmoinnin raaka-aineita 30

Luku 4. Luvut koneiden raaka-aineina 31

Funktio on funktio myös ohjelmoidessa 35

Lukujen vertaaminen 40

Luku 5. Merkit ja merkkijonot funktioiden argumentteina 42
Merkkien vertaaminen 42
Merkkijonot funktioiden argumentteina 45
Merkkijonojen vertaaminen 47
Lisää älyä 48
Luku 6. Hahmonsovitusta ja kanaherkkuja 51
Luku 5. Kanaherkkupusseista listoihin 55
Listat ja listoihin liittyvät funktiot 55
Listat ja hahmonsovitus 58
Luku 6. Filter-funktio 60
Kahden funktion yhdistäminen yhdeksi 64
Luku 7. Map-funktio 70
Luku 8. Fold-funktio 81
Lähteet 88
Lukusuosituksia 88

Luku 1. Johdanto

ATK:n paluu

Ohjelmointi on katsottu Suomessa niin tärkeäksi asiaksi, että aihe lisättiin Suomen peruskoulujen opinto-ohjelmiin 2010-luvun lopulla. Erityisesti peruskoulun opettajille on jo hyvän aikaa järjestetty erilaisia koulutuksia aiheesta. Varsinkin matematiikan opettajien keskuudessa on pohdittu, millainen ohjelmointityyli tukee matemaattisen ajattelun kehittymistä.

Funktionaalinen ohjelmointityyli on vastaus moneen 2000-luvun ohjelmointiongelmaan. Se ei tee ohjelmoinnista yhtään sen helpompaa kuin ennenkään, mutta funktionaalisen lähestymistavan avulla on helpompi suunnitella laadukkaampia ohjelmistoja.

Aiheesta kiinnostuneelle nousee helposti kysymys, miten ohjelmoimaan voisi oppia ja mitä ohjelmointi ylipäänsä on. Tämän kirjan luettuaan lukija saa yhden näkökulman ohjelmointiin. Se on tuskin yhtään parempi tai huonompi kuin mikään muukaan vallitsevista näkökulmista. Yhdestä asiasta lukija voi kuitenkin olla varma. Funktionaalinen ohjelmointi tarjoaa algoritmiseen ajatteluun liittyvät työkalut mahdollisimman pelkistetyssä ja yksinkertaisessa muodossa. Vähemmän on enemmän.

Algoritminen ajattelu ja reseptit

Lukuisissa verkkokeskustelussa tai aiheeseen liittyvissä artikkeleissa viljellään sanaparia algoritminen ajattelu kuvaamaan logiikkaa tai vaiheita, joiden lopputuloksena ohjelmakoodin nähdään syntyvän. Vaiheiden tai askelten korostaminen on omiaan johtamaan siihen, että oppija alkaa nähdä ongelmanratkaisun koodiriveinä, joita suoritetaan algoritmin vaiheiden mukaisesti. Hyvin usein algoritmista ajattelua verrataan ruokareseptiin – ruoka-annos syntyy reseptin mukaisesti ja reseptin mukaan toimiminen nähdään ajassa etenevinä erillisinä tapahtumina.

Reseptirinnastukseen liittyy kuitenkin ongelma, joka piilee sen pykälittäisyydessä. Rinnastus korostaa työvaiheita ja tekemistä – otetaan puuro uunista ja ripotellaan kanelia päälle. Tiedon käsittelyyn liittyvien ongelmien helposti tulkittavat kuvaukset eivät kuitenkaan esitä peräkkäin suoritettavia toimintosarjoja, vaan pikemminkin kuvaavat ongelman sellaisenaan. Kyse ei nykyaikaisissa tiedonkäsittelytehtävissä ole niinkään tietokoneelle annettavista käskyistä vaan tiedonkäsittelysuunnitelman kuvaamisesta.

Ylhäältä alas tulkittavista ohjeista yhteistoiminnan kuvaamiseen

Ohjelmoinnin ymmärtämiseksi ja algoritmisen ajattelun oivaltamiseksi ei tarvita muuttujia tai mekaanisen tiedonkäsittelyjärjestelmän fyysisiin ulottuvuuksiin viittaavia käsitteitä. Kun algoritmia suunnitellessa keskustelu kääntyy muistiin ja muuttujiin, olemme jo kulkeneet hyvän matkaa kohti käsitemaailmaa, joka liittyy tietokoneen toimintaan, ei niinkään itse tiedonkäsittelyongelman kuvaamiseen.

Kun ohjelmointia opiskellessa tulee tarve käyttää sellaisia ajatuksia, joissa "laitetaan numero jemmaan" tai "laitetaan näppäimistölle kirjoitettu sana muistiin" olemme lähestyneet ajattelumallia ja käsitteitä, joita ei nykyaikaisessa ohjelmointityössä välttämättä tarvita.

Tämä ei kuitenkaan tarkoita, että reseptivertauskuva olisi erityisen huono malli algoritmisen ajattelun kuvaamiseksi, vaan sitä, että reseptivertauskuva on huono ongelmanratkaisun kuvausmalli, jos sen kuvausvoima halutaan ulottaa myös sellaisiin tiedonkäsittelytilanteisiin, joihin se soveltuu huonosti. Tässä kirjassa algoritminen ajattelu ymmärretään erityisesti ohjelmoijan työkaluna, ei kaikenkattavana ajattelumallina, jolla ratkaista ongelma kuin ongelma.

Onko ohjelmointi viisasta rinnastaa resepteihin, jos rinnastus kuvaa huonosti vallitsevaa todellisuutta? Yhtäältä reseptivertaus paljastaa hieman liikaa tiedonkäsittelyn konehuoneesta ja toisaalta piilottaa aivan liiaksi sen maailman, jossa moni tiedonkäsittelyn ammattilainen tällä hetkellä toimii. Miten reseptivertaus istuu esimerkiksi siihen tiedonkäsittelyn todellisuuteen, johon kuuluu oleellisena osana jatkuva tietovirtojen käsittelyn ohjaaminen? Vastaus: ei mitenkään.

Lisäksi reseptivertaus korostaa pieniä vaiheita, yksityiskohtia, tekemistä ja toimintaa: uunin laittaminen päälle, maidon laittaminen lämpiämään hellalle, ruoka-aineiden sekoittaminen oikeassa suhteessa keskenään, uunin lämpiämisen odottaminen 200 asteeseen ja niin edelleen. Reseptivertaus korostaa yksittäisiä vaiheita yhden ongelman ratkaisemiseksi ja jättää hyvin vähälle huomiolle sen, että ohjelmien kuvaaminen on yhä useammin yhdessä toimivien tietoa käsittelevien yksiköiden (funktio) yhteistoiminnan kuvaamista.

Tietojärjestelmän laatu ei riipu valitusta ohjelmointiparadigmasta, vaan siitä miten eri paradigmoja sovelletaan kulloiseenkin ohjelmointiongelmaan. Jos niin peruskoulussa kuin yliopistoissa ja korkeakouluissakin pääasiassa tarjottavat rinnastukset ohjaavat vahvasti imperatiiviseen paradigmaan funktionaalisen paradigman kustannuksella, edellä mainittua valintaa ei päästä tekemään. Siksi tarvitaan uteliasta mieltä ja tietoa muistakin vaihtoehdoista.

Resepteistä tiedonkäsittelykoneisiin

Reseptirinnastuksen rinnalla tai jopa sen tilalla voisimme käyttää tiedon käsittelyn kuvaamiseen tiedonkäsittelykonetta. Jokainen kone saa jotain työstettävää ja jokainen kone myös saa jotain aikaiseksi.

Koneen työstämä raaka-aine on aina tietoa jossain muodossa ja koneen lopputulos on niin ikään tietoa. Ohjelmointi voitaisiin ymmärtää reseptien sijaan tiedonkäsittelykoneiden kuvaamisena. Tiedonkäsittelykoneiden, joiden tuotanto riippuu siitä, millaista on syötettävä raaka-aine eli tieto.

Idea ei ole uusi, sillä funktiokoneen käsitettä on käytetty esimerkiksi matematiikan opetuksessa jo pitkään ja se tuntuu sopivalta rinnastukselta myös johdatellessa ajatuksia funktionaaliseen ohjelmointiin.

Kuten television kokkikilpailuista tiedämme, kokin tila voi vaikuttaa syntyvään soppaan, vaikka raaka-aineet olisivat kaikilla yhtä hyvät. Kokki on voinut olla kilpailua edeltävänä iltana juhlimassa ja kisapäivänä päänsärky on yllättänyt.

Funktionaalisessa ohjelmoinnissa ei ole tilaa. Kärjistäen voidaan sanoa, että koska tilaa ei ole, ei ole päänsärkyäkään.

"Käymme yhdessä ain, käymme aina..."

Ohjelmoinnin rinnastaminen tiedonkäsittelykoneeseen ruuanvalmistusreseptien sijaan toimii myös siksi, että sen avulla on helppo kuvata asioiden tapahtumista rinnakkain. Tehtaaseen rakannettavan uuden tuotantolinjan rinnastaminen rinnakkaiseen suorittamiseen lienee helpompaa kuin selittää neljä ”tilallista” kokkia valmistamassa rinta rinnan samaa ateriaa. Uusi tuotantolinja lisätään edellisten ”rinnalle” ja nyt tehtaassa saadaan samassa ajassa enemmän aikaan. Ongelma kumpuaa siitä, että jokainen kokki on erilainen: yhdellä on päänsärky, toisella on käsi kipeä, kolmannella on huimausta ja neljäs on hiukan huppelissa.

Kokkien sijaan tehtaaseen voidaankin rakentaa samanlaiset rinnakkain toimivat tehtaan lattiaan pultatut tuotantolinjat koneineen. Jokainen kone on aina ”iskussa” – samoilla raaka-aineilla koneista saadaan aina samat tuotokset. Koneella ei ole huonoja päiviä.

Rinnakkaisuuden kuvaaminen on erityisen tärkeää tietokonelaitteistojen kehittyessä niin, että ne tekevät yhä enemmän asioita samaan aikaan. Nykyaikaisissa tiedonkäsittelylaitteistoissa on monia rinnakkaisia tuotantolinjoja. Tietokoneille on viisasta tehdä sellaisia ohjelmia, jotka parhaiten soveltuvat niitä suorittaviin laitteisiin.

Rinnakkaisuuden tai samanaikaisuuden korostaminen tämän kirjasen esipuheessa voi alkuun kuulostaa lennokkaalta, mutta sitä se ei suinkaan ole. Rinnakkain suorittavat ja siis useita tuotantolinjoja edustavat tietokoneet ovat arkipäivää aina suurista tietokoneista sormenpään kokoisiin laitteisiin.

Tyylillä on väliä

Ohjelmointityylin valinnalla on siis merkitystä. Imperatiivinen tyyli on edelleen usein ainoa vaihtoehto, kun laadimme niin sanottuja matalan tason ohjelmia – ohjelmia, jotka ohjaavat välittömästi tietokoneen toimintaa. Imperatiivinen, komentava ohjelmointityyli on hyvä valinta myös silloin, jos pitää saada aikaan ohjelmia, joiden suorituskyvyn tulee olla erinomaisen hyvä. Käskyjä ja komentoja korostavan ohjelmointityylin mannekiiniksi ruokaresepti kokkeineen istuu siis mainiosti, sillä sellaisessa ohjelmointityylissä on aina tila mukana.

Jos meidän pitääkin kirjoittaa ohjelma mahdollisimman lyhyesti ja ilmaisuvoimaisesti niin, että ohjelmakoodin merkitys on mahdollisimman helposti tulkittavissa, valintamme on funktionaalinen ohjelmointityyli. Funktionaalinen ohjelmointityyli on usein nopein, selkein ja helpoin tapa kuvata minkä tahansa tiedonkäsittelyongelman ratkaisu.

Esimerkiksi tekoälyyn tai tilastomatematiikkaan liittyviä ongelmia on haastavaa ratkoa niin, että ratkaisua etsitään pohtimalla koodin etenemistä ja yksittäisen koodirivin suoritusta ja miettimällä kuinka monta muuttujaa ongelmaa ratkaistaessa tarvitaan. Tiedon käsittelyyn liittyvät ongelmat ratkeavat usein helpommin kuvaamalla ne funktionaalisella tyylillä.

Eräs funktionaalisen ohjelmoinnin vahvuuksista on, että funktionaalista ohjelmaa on helpompi tulkita kuin imperatiivista ohjelmaa. Tämä näkyy hyvin esimerkiksi korkeamman asteen funktioissa map, filter ja fold (reduce).

Imperatiivisessa ohjelmointikielessä joukkoja manipuloidaan tyypillisesti silmukan avulla. Tässä kohtaa näemme koodissa tyypillisesti for-silmukan tai vastaavan toistorakenteen. Koodin luettavuusongelmaksi muodostuu, että näemme saman näköisen for-silmukan oli kyse sitten tiedon suodattamisesta (filter) tai jokaista alkiota koskevasta muunnoksesta(map) tai jostain muusta, kuten fold(reduce) operaatiosta. For-rakenne ei sellaisenaan anna koodin lukijalle vihjettä tiedonkäsittelyoperaation luonteesta. Alla oleva kuva selventää asiaa.

imperatiivinen

```
merkkijono kirjaimet[] = {'a','b','c'}
merkkijono uudetKirjaimet[]
for (int x=0;x<3;x++) {
    uudetKirjaimet[]=toUpper(koirat[x])
}
```

```
merkkijono kirjaimet[] = {'a','b','c'}
merkkijono bKirjaimet[]
int z=0
for (int x=0;x<3;x++) {
    if (nimenEkaKirjain(kirjaimet[x])=='s' {
        bKirjaimet[z]=kirjaimet[x]
        z++
    }
}
```

funktionaalinen

```
kirjaimet' = ['a','b','c']
uudetKirjaimet = map (toUpper) kirjaimet'
```

```
kirjaimet' = ['a','b','c']
bKirjaimet = filter (=='b') kirjaimet'
```

Kuva 1.Imperatiivinen ja funktionaalinen ohjelmointityyli.

Koodin selkeydellä ja luettavuudella on yhteys tietojärjestelmän kehitys- ja ylläpitokustannuksiin. Jos koodia on vaikea tulkita, sitä on myös vaikea muuttaa ja kehittää. Pilvilaskennan yleistyessä palvelittomat ratkaisut ovat erinomainen sovellusalue funktionaaliselle ohjelmointityylille: pysyvyysratkaisu (esimerkiksi relaatiotietokanta) ja funktiot ovat erillään eikä muuttujille sinänsä ole tarvetta. Tilanhallinta on jätetty taustajärjestelmän huoleksi ja ohjelmistosuunnittelija voi keskittyä olennaiseen – laatimaan helposti tulkittavia ja ylläpidettäviä sovelluksia.

Tämän tekstin tarkoituksena ei ole väheksyä imperatiivista ohjelmointityyliä sen enempää kuin ylettömästi korostaa funktionaalisen ohjelmointityylin erityisluonnettakaan. Kirjan kirjoittaja ymmärtää myös funktionaalisen ohjelmointityylin ongelmat, jos ohjelmointityyliä sovelletaan sellaiseen käyttötilanteeseen, johon se ei sovi. Tekstin tarkoituksena on herättää lukija ajattelemaan ohjelmointia tiedon käsittelyn näkökulmasta ja pohtimaan millainen ohjelmointityyli omiin tarpeisiin sopisi parhaiten.

Matematiikan oppimisesta on paitsi iloa myös hyötyä

Funktionaalinen ohjelmointityyli tukee eri ohjelmointityyleistä parhaiten matemaattisen ajattelun kehittymistä ja on paras tapa oppia sekä ohjelmointia että matematiikkaa samalla kertaa. Tämä johtuu siitä, että funktionaalinen ohjelmointityyli on yleisesti tunnetuista ohjelmointityyleistä luonteeltaan lähinnä matematiikkaa – kaiken perustana on funktion käsite.

Funktionaalisia ohjelmointikieliä on kuitenkin useita ja matemaattisen ajattelun kehittämisen kannalta olisi hyvä valita yhtäältä parhaiten matemaattista ajattelua tukeva ja toisaalta mahdollisimman yksinkertainen kieli. Kieli, joka tukee erinomaisesti sekä ohjelmoinnin että matematiikan taitojen kehittymistä on Haskell. Selma-koira ja minä voimme vakuuttaa, että Haskell-kielen ainoa sivuvaikutus on, että sitä opiskellessa oppii matematiikkaa ihan huomaamatta.

Ohjeita kirjan lukijalle

Haskell-kieli kouluttaa käyttäjänsä esittämään ongelman ratkaisun selkeästi ja yksinkertaisesti. Kieli saa hyvän selkänojan matematiikasta. Haskell-ohjelmassa asiat esitetään lambda-laskennan periaatteiden mukaan laiskasti evaluoitavina lausekkeina.

Kirja johdattelee sinut Selma-koiran ja hänen ystäviensä myötä Haskell-kieleen ja funktionaaliseen tapaan ajatella niin algoritmeista kuin ohjelmoinnista yleensä. Osoitamme Selman ja hänen koiraystäviensä kanssa, että funktionaalinen ohjelmointi Haskell-kielellä on paitsi hauskaa myös haastavaa. Kirjanen voi innostaa myös funktionaalisen ohjelmointityylin opiskeluun yleensä, sillä monet funktionaalisia piirteitä sisältävät ei-funktionaaliset ohjelmointikielet, ovat lainanneet funktionaaliset piirteensä usein jostain puhtaasta funktionaalisesta ohjelmointikielestä, kuten Haskellista.

Kirja pyrkii johdattelemaan lukijaa tiettyyn tapaan ajatella. Lähdemme liikkeelle sort-funktiosta, jota käytämme esimerkkinä tietoa käsittelevästä funktiosta. Jatkamme laskutoimituksia kuvaavilla funktioilla, joissa käytämme raaka-aineena eli argumentteina lukuja. Funktio-sana korvataan kirjan alkuosassa usein sanalla kone ja argumentti sanalla raaka-aine. Lisäksi funktion arvon sijaan käytetään sanaa tuotos, mikä kuvaa funktion arvoa.

Sort-funktioon ja muutamiin lukuja käsitteleviin funktioihin tutustuttuamme, kokeilemme pieniä funktioita ja vertailemme merkkejä ja merkkijonoja toisiinsa. Pyrimme rakentamaan oppijalle ajatuksen, että koneet liittyvät toisiinsa. Lisäksi korostamme, että koneiden liitoskohtia pohtiessa tulee ottaa huomioon, että yhden koneen tuotos on toisen koneen raaka-ainetta.

Merkit ja merkkijonot esitellään, jotta lukija pääsee kokeilemaan vertailukoneita ja toteamaan, miten ohjelmassa voidaan haarautua ja saada ohjelma toimimaan eri tavoin riippuen millaisia raaka-aineita koneisiin milloinkin annostellaan. Merkit ja merkkijonot on hyvä esitellä aluksi myös siksi, että niiden avulla voi tehdä pieniä ohjelmointiharjoituksia ja opetella uusia taitoja.

Hahmonsovitusta käsitellään hyvin pintapuolisesti, mutta sen jättäminen pois olisi johtanut siihen, että moni verkosta löytyvä pieni koodiesimerkki ei olisi lukijalle auennut – hahmonsovitus on niin perustavaa laatua oleva osa Haskell-kieltä, että sitä ei voi lyhyessä johdannossakaan hyvällä omalla tunnolla täysin ohittaa.

Kirjan loppuosa on omistettu lähinnä lista-tyyppisen tiedon ja muutaman tärkeimmän korkeamman asteen funktion esittelyyn. Lista on ohjelmoinnin tärkeimpiä ellei tärkein tietorakenne. Listoja esittelevät esimerkit muuttuvat vaikeammiksi kirjan loppua kohti ja fold-funktiota esittelevä esimerkki omine tietotyyppeineen on aloittelijalle usein haastavaa pohdittavaa.

Korkeamman asteen funktiot kannattaa opetella järjestyksessä yksinkertaisemmasta monimutkaisempaan. Kannattaa aloittaa filter-funktiosta, sitten tutkia map-funktiota ja vasta viimeisenä pohtia ongelmia, jotka luontevimmin ratkeavat fold-funktiolla. Kyseisiä funktioita viisaasti yhdistelemällä voi ratkaista ongelman kuin ongelman.

Rekursion käsitteeseen tai Haskell-kielen tyyppijärjestelmään ei paneuduta. Rekursio ei ole funktionaalisen ohjelmoinnin oppimisen ehto – korkeamman asteen funktion käyttäjä on usein myös rekursiivisen funktion käyttäjä enemmän tai vähemmän tietoisena siitä, että toisto perustuu rekursioon.

Tietoisuus rekursiosta toistomekanismina ei erityisesti hyödytä tavanomaisia ohjelmointiongelmia ratkovaa ohjelmistosuunnittelijaa tai harrastajaa. Puhtaan funktionaaliset tai funktionaalisia ominaisuuksia sisältävät ohjelmointikielet tarjoavat usein työkalut, jotka piilottavat rekursion korkeamman asteen funktioihin.

Tietoisuus ohjelman rekursiivisesta luonteesta voi kuitenkin olla hyödyksi esimerkiksi tilanteissa, joissa kohdataan suorituskykyongelmia tai vaikkapa pinon ylivuototilanne. Teoreettisesti suuntautunut saattaakin lähteä tämän kirjasen kahlattuaan tutkimaan rekursiota – käytännönläheisempi käyttää korkeamman asteen funktioita sellaisenaan.

Kirjassa esiintyviä pieniä ohjelmointitehtäviä ja näkyviä koodirivejä voi harjoitella Haskell-tulkilla. Haskell-tulkin käyttämiseen löydät apua katsomalla videon osoitteesta https://youtu.be/zUxXkrSEx0Y

Kirjan koodiesimerkeissä merkki > tarkoittaa, että koodirivi kirjoitetaan ja merkki => tarkoittaa, että komentorivi tulostaa kyseisen rivin näyttäessään funktion lopputuotoksen.

Tsemppiä ohjelmointihommiin!

Tampereella 1.2.2020

Juuso Vuorinen

Koska niin koirat kuin ihmisetkin tekevät viljalti virheitä ja toisinaan niistä oppivatkin, olisi mukavaa, jos laittaisit teosta koskevat kommentit, kehitysehdotukset ja muut ideat sähköpostitse osoitteeseen juuso.vuorinen@ideallearning.fi.

Luku 2. Koneet

Tiedon käsittelyn kuvaamista ei voi automatisoida

Tiedon käsittelyn taito lienee eräs tärkeimmistä ohjelmoijan perustaidoista. Tietoa käsitellessä tarvitsemme työkaluja - ne käsitteet, joiden varassa tiedonkäsittelykoneemme toiminta kuvataan.

Tiedon käsittelyn vaiheiden kuvaaminen ei siis tapahdu itsestään. Se on jotain, mitä emme voi automatisoida, koska se on automatisaation raaka-ainetta. Tiedon käsittelemiseksi meidän tulee määritellä, miten annetusta tiedosta luodaan uutta tietoa. Tätä käsittelyketjua voidaan kuvata tehtaassa toimivilla koneilla, jotka saavat jotain mielekästä aikaan.

Funktio tiedonkäsittelykoneena

Koneesta voimme käyttää myös nimitystä funktio. Funktiolle on ominaista, että jos haluamme saada arvon (koneen valmistama tai tuottama asia) meidän tulee antaa argumentti (raaka-aine).

Funktion soveltuvuus tiedonkäsittelykoneistomme peruskäsitteeksi on erityisen hyvä juuri siksi, että sen avulla tiedon käsittelyyn liittyvät kuvaustarpeemme tulevat erittäin hyvin tyydytetyiksi suhteessa nykyaikaisen tietojenkäsittelyn vaatimuksiin. Funktio on abstraktiona sopivasti irti tietokoneesta fyysisenä laitteena, mutta käsitteenä riittävän käytännöllinen tarjoamaan yksiselitteisen ja helposti tulkittavan tavan kuvaamaan tiedon käsittelyä.

Jos vertaamme funktionaalista ja imperatiivista lähestymistapaa, huomaamme, että funktionaalinen ohjelmointityyli sopii koneineen imperatiivista tyyliä paremmin juuri tiedonkäsittelyongelmien kuvaamiseen ja ratkaisemiseen. Ero on hiukan sama kuin numeroilla ja tukkimiehen kirjanpidolla.

Tukkimiehen kirjanpidolla kirjaa pitävä pärjää ilman numeroita. Jos tukkimies vie kirjanpitonsa arabialaisia numeroita ymmärtävälle kirjanpitäjälle, voi kirjanpidon laatiminen osoittautua vaikeaksi.

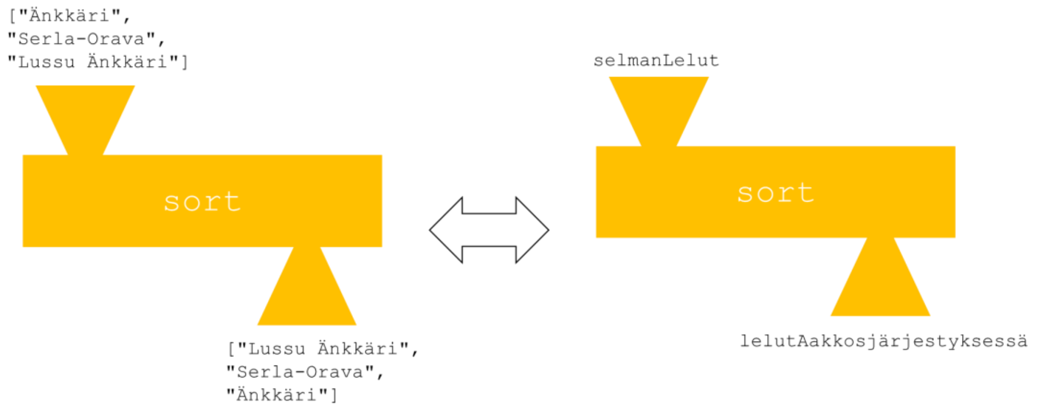
Kun kirjanpitäjä yrittää tulkita tukkimiehen kirjanpidossa näkyviä lukumääriä, hän joutuu ensin laskemaan viiden niput yhteen. Sen jälkeen kirjanpitäjän on vielä lisättävä edelliseen lukuun jäljelle jäävien viivojen summa. Voi olla, että kirjanpitäjä joutuu tekemään useita laskutoimituksia ja laittamaan lukuja muistiin päästäkseen lopulta oikeisiin tuloksiin arabialaisin numeroin.

Laskiessaan viiden nippuja kirjanpitäjälle voi tulla helposti virhe, sillä joku nipuista voi jäädä laskematta tai kirjanpitäjä laskee vahingossa yhden viiden nipun liikaa. Toisiaan lähellä olevien viivojen yhteen laskeminen on sekin vaivalloista ja virhealtista.

Voinemme hieman kärjistäen sanoa, että funktionaalista ohjelmointityyliä suosiva käyttää arabialaisia numeroita siinä, missä imperatiivisesti ohjelmoiva turvautuu tukkimiehen kirjanpitoon. Imperatiivisen ja funktionaalisen ohjelmointityylin erojen kuvaaminen on vaikeaa eikä kärjistys tee oikeutta kummallekaan ohjelmointityylille. Kärjistyksen tarkoitus on vain ja ainoastaan antaa lukijalle yksi ajatus siitä, millä tavalla näiden kahden eri ohjelmointityylin voi katsoa eroavan. Annetaanpa nyt Patelle ja Selmalle puheenvuoro ja lähdetään tutkimaan ensimmäistä konettamme eli sort-funktiota.

Ensimmäinen koneemme

”Katsotaanpa Pate `sort`-koneen kuvaa ja ohjelmakoodia”, sanoo Selma tomerasti. ”Ohjelmakoodin avulla voin järjestää leluni aakkosjärjestykseen:



Kuva 2. Kaksi tapaa käyttää sort-konetta

```
import Data.List.sort

selmanLelut = ["Änkkäri","Serla-Orava","Lussu Änkkäri"]

lelutAakkosjärjestyksessä = sort selmanLelut
```

”Mitä kuvat oikein tarkoittavat?” kysyy Pate.

”Odotahan, niin kerron”, jatkaa Selma. ”Kahteen suuntaan osoittavan nuolen vasemmalla puolella annostelemme raaka-aineet koneeseen sellaisinaan ja nuolen oikealle puolella olevassa kuvassa raaka-aineet annostellaan viittaamalla raaka-ainetta tarkoittavaan nimeen.”

”No mitä rivit tarkoittavat?” jatkaa Pate taas uteliaasti.

”Kuulehan, niin kerron”, sanoo Selma. ”Ensimmäinen rivi kertoo, että haluamme käyttää valmista konetta (funktiota) `sort`, joka osaa järjestää sanalistan aakkosjärjestykseen. Jos saatavilla on jo olemassa olevia koneita, joilla saadaan aikaan haluttu lopputulos, niin sellaisia koneita kannattaa jokaisen koiran käyttää sen sijaan, että rakentaisimme koneet alusta asti itse. `Sort` on esimerkki tällaisesta valmiista koneesta. Jotta saamme koneen käyttöömme, meidän pitää `import`-sanan avulla kertoa, että haluamme ottaa käyttöön listojen käsittelyyn erikoistuneen `sort`-nimisen koneen.

Voimme ajatella toisen ja kolmannen rivin asiat kahtena eri koneena. Toisella rivillä yhtäsuuruusmerkin oikealla puolella oleva kone (funktio) tuottaa listan sille annetuista alkioista siinä järjestyksessä, jossa ne ohjelmakoodissa hakasulkujen välissä luetellaan. Kolmannella rivillä yhtäsuuruusmerkin oikealle puolella oleva kone (funktio) tuottaa järjestyksessä olevan listan niistä alkioista, jotka ensimmäinen kone tuotti.

Toisella rivillä kuvataan ohjelmakoodin avulla lista, joka sisältää kolme alkiota: suosikkileluni. Kuten ohjelmakoodista näkyy, listan alkiot erotetaan toisistaan pilkulla ja listan molemmissa päissä on hakasulkeet. Lisäksi jokaisen lelua kuvaavan sanan ympärillä on lainausmerkit. Miltä tämä kaikki Pate näyttää?” kysyy Selma.

”Nyt en kyllä yhtään tajua”, sanoo Pate Selmalle. ”Miksi toisen ja kolmannen rivin koneet ovat yhtäsuuruusmerkin oikealla puolella niin eri näköisiä, vaikka molemmat ovat koneita? Kolmannen rivin koodin ymmärrän kyllä hyvin. Siinähän pyydetään selvästi annostelemaan sanalista `sort`-nimiselle koneelle ja sitten vain odotamme koneen käynnistämistä. Mutta missä on se kone (funktio) toisella rivillä, koska siinähän on vain yksi pötkö hakasulkeiden välissä: `["Änkkäri","Serla-Orava","Lussu Änkkäri"]`?” ihmettelee Pate. ”Missä on se kone, jonkin nimi voisi olla vaikkapa `luoLista`?” jatkaa Pate ihmetellen.

”Kuulehan Pate, niin kerron”, sanoo Selma. ”Sanoimme, että nimi `selmanLelut` edustaa listaa, joka sisältää lelut alkuperäisessä järjestyksessä. Emme kuitenkaan näe ensimmäisen ohjelmarivin yhtäsuuruusmerkin oikealle puolella nimettyä funktiota, kuten `luoLista`. Mistä tässä sitten on kyse? Olenko minä Pate huijannut sinua? En ole. Hakasulkeet edustavat tässä tapauksessa konetta, joka luo listan luetelluista leluista.

Yhtäsuuruusmerkin oikealla puolella on siis aina kone tai koneen nimi. Yhtäsuuruusmerkin vasemmalla puolella on aina koneen nimi.

Käsin kirjoitetut lelujen nimet ovat siis listan luovan funktion raaka-aine. Näin ollen yhtäsuuruusmerkin oikealla puolella oleva on kuin onkin funktio ja vasemmalla puolella nimi, joka edustaa funktiota eli funktion nimi. Joskus on niin, että koneet eivät ole ohjelmasta niin kovin hyvin tunnistettavissa – hyvä esimerkki tästä ovat hakasulkeet, jotka toimivat listoja luovana koneena.

Yhden kysymyksen Pate vielä haluaisin sinulle esittää. Mitä tapahtuisi, jos kirjoittaisit koodin `x = x` ja sitten käynnistäisit koneen kirjoittamalla komentoriville `x`?”

”Nytpä kysyitkin visaisen kysymyksen Selma. Odotahan, niin mietin. Jos komentoriville kirjoitetaan nyt `x`, niin se varmasti käynnistää koneen, jonka nimi on `x`. Se kone on sellainen, että se käynnistää saman nimisen koneen uudelleen, mikä johtaa taas koneen käynnistämiseen uudelleen, ikuisesti. Voisiko siinä käydä niin, että kone jää ikuisesti tuottamaan vain ja ainoastaan sen, että konetta käynnistetään ikuisesti maailman loppuun asti?”

”Hyvä Pate!” tokaisee Selma. ”Niin siinä tosiaankin käy. Kone ei varsinaisesti tuota mitään järkevää, mutta se jää kuitenkin ikuisesti käyntiin. Kokeillaanpa:

```
> x = x

> x
```

Voit nyt pysäyttää jumiin jääneen ohjelman suorituksen, jotta pääset jatkamaan ohjelmointiharjoituksia. Asia voi tuntua oudolta, mutta palaamme siihen myöhemmin. Sitä ennen sinä Pate ansaitset kyllä kanaherkun, kun olet noin urheasti jaksanut selvittää asioita.”

Tuotantoketju ja koneiden keskinäiset riippuvuudet

Funktio on siinä mielessä fiksu kone, että sen tuotantokoneisto käynnistyy vain, jos jokin muu kone (funktio) tilaa siltä raaka-ainetta. Toisen koneen raaka-ainetarve käynnistää ensimmäisen koneen tuotannon. Ensimmäinen kone (funktio) ei tuota mitään varastoon, vaan sen käynnistyminen on täysin riippuvainen siitä, tarvitaanko sen koneita (funktioita) osana jotain toista tuotantoketjua (muita funktioita).

Voimme ajatella, että `selmanLelut` on koneen nimi. `SelmanLelut`-niminen kone tuottaa listan, joka sisältää kolme Selma-koiran parasta lelua. Alla olevasta ohjelmakoodista näemme, kuinka koneen nimi `selmanLelut` on nyt korvattu itse koneella hakasulkeineen päivineen. `Sort`-koneen nimen jälkeen on siis alkuperäinen lelulista kirjoitettuna sellaisenaan. On oleellista ymmärtää, että koneen nimen voi aina korvata sillä koneella, jota nimi edustaa. Tästä syystä alla oleva kuvaus koneesta antaa saman tuotoksen kuin edellinenkin kuvaus:

```
1 import Data.List

2 lelutAakkosjärjestyksessä = sort ["Änkkäri","Serla-
Orava","Lussu Änkkäri"]

> lelutAakkosjärjestyksessä

=> ["Lussu Änkkäri","Serla-Orava","Änkkäri"]
```

Yhtäsuuruusmerkin vasemmalla puolella oleva nimi on siis lyhyempi tapa ilmaista yhtäsuuruusmerkin oikealla puolella oleva. Selman mielestä ei ole viisasta toistaa joka paikassa kolme alkiota sisältävää listaa, vaan on helpompaa kirjoittaa sen sijaan nimi `selmanLelut` siellä, missä tarvitaan `["Änkkäri","Serla-Orava", "Lussu Änkkäri"]`. Palatkaamme siis tutkimaan alkuperäistä ohjelmakoodiamme:

```
1 import Data.List.sort
2 selmanLelut = ["Änkkäri","Serla-Orava","Lussu Änkkäri"]
3 lelutAakkosjärjestyksessä = sort selmanLelut
> lelutAakkosjärjestyksessä
=> ["Lussu Änkkäri","Serla-Orava","Änkkäri"]
```

”Kirjoittamalla Haskell-tulkille funktion (koneen) nimen `lelutAakkosjärjestyksessä` ja painamalla etutassulla return-näppäintä tulemme käynnistäneeksi (ainakin) kaksi konetta.”, jatkaa Selma. ”Joko Pate arvaat mitkä kaksi konetta käynnistyvät ja missä järjestyksessä?” kysyy Selma.

”Tämä on helppoa”, jatkaa Pate. ”Ensin käynnistyy tietenkin se kone, joka luo `selmanLelut` listan ja kun se lista on luotu, niin lista annostellaan tuolle `sort`-koneelle, joka sitten luo uuden järjestetyn listan, joka sitten näkyy tuossa ruudullakin. Meidän pitää tietenkin ensin mennä tuotantoketjun alkupäähän ja käynnistää tuotanto ensimmäisestä tuotannon vaiheesta eli luoda alkuperäinen lista. Jotta `sort`-kone voi käsitellä lelulistaa, pitää lelulista siis ensin luoda. Tämä on ihan helppoa päättelyä tällaiselle lempääläläiselle vehnäterrierille”, jatkaa Pate ja iskee Selmalle silmää.

”Aivan oikein meni!” vastaa Selma. ”Sinustahan on tulossa oikea koodivelho, etten paremmin sanoisi”, Selma päättää.

”No voi olla, etten ihan vielä tänään käy velhosta, mutta ehkä jo muutaman kuukauden päästä. Mitä muuten tapahtuu, jos kirjoitan komentokehotteeseen `selmanLelut` ja painan etutassulla return-näppäintä?” jatkaa Pate juttua.

”Mitä arvelisit?” sanoo Selma. ”No ehkä `selmanLelut` käynnistäisi sen koneen, joka luo tuon alkuperäisen listan. Kokeillaanpa, miten siinä käy”, jatkaa Pate.

```
> selmanLelut
=> ["Änkkäri","Serla-Orava","Lussu Änkkäri"]
```

”No tuo olikin äkkiä kokeiltu”, hihkuu Pate. ”Kirjoitin nimen `selmanLelut` ja painoin return-näppäintä etutassulla. Tämä todellakin toimii!” jatkaa Pate innoissaan.

"Onko ohjelmoinnissa kyse siis niinkin yksinkertaisesta asiasta kuin yhden tai useamman koneen tehtävän kuvaamisesta ja koneiden yhdistämisestä?" kysyy vuorostaan Pate äimistyneenä.

"Kyllä vain", vastaa Selma ja jatkaa: ”Se on juuri sitä ja se on loppujen lopuksi jotain koiramaisen yksinkertaista.”

Ei miten, vaan mitä

"Meidän koiramaisiin tiedonkäsittelykoneisiimme liittyy vielä yksi piirre, josta haluan sinulle Pate kertoa. Me emme ole kiinnostuneet käykö koneemme sähkö- vai polttomoottorilla. Meidän koneissamme huomio kiinnittyy aina siihen, millaisia raaka-aineita ne käyttävät ja millaisia tuotoksia niistä syntyy. Tämän kun Pate vielä muistat, niin olet jo pitkällä ajatuksenjuoksussasi matkalla kohti taitoja, joiden avulla nykyaikaisia tiedonkäsittelykoneita luodaan", päättää Selma.

Ensimmäisen esimerkin kolmannen rivin ohjelmakoodista ilmenee hyvin funktionaalisen ohjelmoinnin perusajatus: Selma-koira ei ole funktionaalisesti ohjelmoidessaan kiinnostunut siitä, mitä tietokoneen muistissa tapahtuu, kun lelulistaa järjestetään. Selmaa kiinnostaa se, että lelut saadaan ylipäänsä järjestykseen. Vaikka tutkisimme `sort`-koneen Haskell-kielistä lähdekoodia `Sort`-niminen kone ei paljasta koneen käyttäjälle, miten lelut järjestetään tietokoneen näkökulmasta ja mitä muistiosoitteissa tapahtuu, kun järjestyalgoritmiin annostellaan järjestettävä lista – lähdekoodissa ei ole muuttujia. `Sort`-niminen kone tarvitsee raaka-aineekseen listoja ja se tuottaa järjestettyjä listoja. ”Lista sisään, lista ulos, siitä syntyy lopputulos”, riimittelee Pate.

Kolmannella koodirivillä nähdään sama ilmiö kuin toisellakin. Yhtäsuuruusmerkin vasemmalle puolelle kirjoitetaan sana, jonka avulla lyhennetään yhtäsuuruusmerkin oikealla puolella oleva. Selma voi nyt käyttää ilmaisua `lelutAakkosjärjestyk-sessä` lyhentämään yhtäsuuruusmerkin oikealla puolella olevan yhdeksi nimeksi: `lelutAakkosjärjestyksessä`. Kirjoittamalla `lelutAakkosjärjestyksessä` komentorivikehotteeseen saadaan näkyville koneen tuotos, aakkosjärjestykseen järjestetty lista:

```
> lelutAakkosjärjestyksessä

=> ["Lussu Änkkäri","Serla-Orava","Änkkäri"]
```

Funktion arvon selvittäminen

Aiemmin todettiin, että funktio saa jotain ja siitä tulee ulos jotain. Jotta funktiosta saataisiin jotain ulos, on funktion arvo eli tuotos selvitettävä. Funktion arvo selvitetään (evaluoidaan) vain, kun sitä tarvitaan. Pate voi pyytää tietokonetta selvittämään mikä on `selmanLelut` lista aakkosjärjestyksessä kirjoittamalla komentokehotteeseen nimen `lelutAakkosjärjestyksessä`, jonka jälkeen näemme listan aakkosjärjestyksessä. Koneiden käynnistäminen nimeä käyttämällä johtaa siihen, että koneet alkavat tuottaa jotain mielekästä eli arvoja.

"Onko tässä arvonselvittelyhommassa siis kyse jonkin koneen käynnistämisestä ensin ja sitten kun se on käynnistetty, niin muutkin koneet saattavat käynnistyä, mutta vain silloin, jos ne jotenkin liittyvät käynnistyneen koneen touhuihin vai miten se nyt oikein liittyy koneisiin?" kysyy Pate hämillään. "Kyllä vain", vastaa Selma. "Kyse on juuri siitä. Konetta ei ole mielekästä käynnistää ennen kuin joku haluaa jossain käyttää koneen tuotoksia olivatpa ne sitten koiran nappuloita, kanaherkkuja tai vaikkapa metelöiviä änkkärileluja. Meille ei kukaan lemmikkiruokavalmistaja valmista kanaherkkuja, ellemme me niitä halua ja mehän haluamme", jatkaa Selma ja lipoo kanaherkkuja kaipaavia huuliaan.

”Tästä lähtien voimme Pate kutsua koneitamme laiskoiksi, sillä ne eivät tee mitään, ellei niiden tuotoksia joku jossain tarvitse. Laiskuus on hyvä motto meille koirillekin, ainakin tällaisina kuumina kesäpäivinä”, sanoo Selma niin laiskasti kuin suinkin pystyy.

Kun pyydämme konettamme muuntamaan alkuperäisen lelulistan uudeksi järjestetyksi lelulistaksi, saamme lopputuloksena arvon (koneen tuotos), joka on Selman lelulista aakkosjärjestyksessä. Esimerkkikoodissa yhtäsuuruusmerkin vasemmalla puolelle oleva nimi `lelutAakkosjärjestyksessä` edustaa lelulistalle tehtyä muunnosta alkuperäisestä lelulistasta järjestettyyn lelulistaan. Kirjoittamalla nimen `lelutAakkosjärjestykessä` komentokehotteeseen ja painamalla return-näppäintä käynnistämme kaksikoneisen tehtaamme.

Aina, kun pyydämme tietokonetta selvittämään arvon, jota edustaa nimi `lelutAakkosjärjestyksessä`, tietokone selvittää ensin `selmanLelut` nimeen liittyvän arvon, sillä `selmanLelut` on osa arvonselvittelyketjua (koneita tuottamassa kukin omia tuotoksiaan ja siirtämässä tuotoksia ketjussa seuraavaan koneeseen). Halumme saada lelut aakkosjärjestykseen laukaisee siis tuotantoketjun, joka alkaa siitä, että alkuperäisestä lelulistasta luodaan arvo ja päättyy siihen, että alkuperäisestä listasta muodostettu arvo annostellaan `sort`-nimiseen koneeseen (funktioon), joka taas muodostaa uuden arvon eli järjestetyn listan.

Koneiden maailmasta tiedämme, että tuotannon eri vaiheissa tarvittavien koneiden kokoonpano riippuu haluamastamme lopputuloksesta. Funktiot edustavat tehdastuotannon maailmassa tuotantoketjuun kuuluvia koneita. On siis valittava sopivat koneet halutun lopputuotoksen mukaan. Tähän voimme käyttää olemassa olevia koneita tai rakentaa omia.

`Sort`-funktio tietää, miten se saa listan järjestykseen. Se on `sort`-funktion äly. Meidän on vain annettava funktiolle tieto siitä alkuperäisestä listasta, josta tulisi muodostaa uusi, aakkosjärjestyksessä oleva lista. Tuolle annettavalle tiedolle on myös toinen nimi: funktion argumentti, yksi tai useampia. Voidaan ajatella, että itse `sort`-funktio voi sisältää piilossa olevia funktioita (koneita), joiden yhteistoiminnan avulla järjestämätön tieto saadaan järjestetyksi. Kone tarvitsee siis raaka-ainetta eli argumentin tai useampia. Argumenttien tai tuotoksen eli arvon laatu eivät kuitenkaan vielä kerro koneen sisäistä toiminnasta. Ne paljastavat vain kaksi seikkaa. Yhtäältä sen, mitä raaka-ainetta kone tarvitsee ja toisaalta sen, mitä kone tuottaa.

Funktion argumentti tai argumentit ovat siis tehtaiden ja koneiden maailmassa raaka-ainetta, jota koneeseen annostellaan, jotta koneesta saataisiin jokin lopputuotos. On tärkeää huomata, että yhden koneen tuotos on toisen koneen raaka-aine. Sellukoneen raaka-aine on puu ja sellukoneen tuottama selluloosa on puolestaan paperikoneen raaka-aine. Yhdistämällä sellu- ja paperikoneen yhdeksi suuremmaksi tuotantoyksiköksi saamme koneen, jonka raaka-aine on puu ja tuotos paperi.

Lienemme nyt pohtineet riittämiin tiedonkäsittelykoneisiin liittyviä peruskäsitteitä. Nämä käsitteet ovat: funktio, funktion argumentti, funktion nimi ja funktion arvon selvittäminen eli funktion evaluointi. Näiden avulla voimme rakentaa koneistamme mitä mielikuvituksellisimpia tuotantolaitoksia.

Seuraavassa luvussa käsittelemme tarkemmin funktion argumentteja eli koneen raaka-aineita. Päädymme tutkimaan miten useampia funktioita (koneita) yhdistelemällä saadaan aikaan monimutkaisempia funktioita, joiden avulla voimme ratkaista yhä mutkikkaampia ongelmia.

Luku 3. Raaka-ainetta koneeseen

Raakaa-aineita moneen lähtöön

”Kuten varmaan jo huomasit Pate, `sort`-koneelle (syötettävä raaka-aine näyttää olevan lista. Se on lista siksi, että olemme käyttäneet sen määrittelemiseen hakasulkuja ja pilkkuja erottamaan listan alkiot toistaan. Raaka-aine eli tässä tapauksessa lista -tyyppinen arvo saadaan aikaan funktiolla. Hakasulut edustavat listan muodostavaa funktiota.

Kuten aiemmin totesimme, raaka-ainetta voidaan kutsua myös argumentiksi. Joskus tarvitsemme tuotantoprosessissamme useita eri raaka-aineita. Jos koneemme tarkoitus olisi laskea kaksi lukua yhteen, meidän pitäisi antaa yhteenlaskukoneellemme nämä luvut, samalla periaatteella kuin annoimme `sort`-koneelle listan leluja.”

Usean eri raaka-aineen annosteleminen koneelle

”Mutta hei!” huudahtaa Pate hengästyneenä. ”Tarkoitat siis sitä, että me voisimme antaa niitä eri raaka-aineita siinä listassa? Sittenhän meidän olisi helppo antaa mihin tahansa koneeseen niin paljon erilaisia raaka-aineita kuin ikinä haluamme? Jos haluamme laskea kaksi kokonaislukua yhteen, annamme yhteenlaskukoneelle listan, jossa on kaksi kokonaislukua. Voisiko se toimia niin?” jatkaa Pate innoissaan.

”Idea on kyllä hyvä ja sen saisi toimimaan, sillä kaksi kokonaislukua ovat keskenään samantyyppisiä asioita”, sanoo Selma. ”On nimittäin niin, että listalla on oltava aina samantyyppisiä asioita. Siellä on aina joko leluja tai koirien syntymävuosia mutta ei koskaan molempia samaan aikaan. Samalla listalla ei siis saa olla sekä sanaa ”Änkkäri” että lukua 2015. Ymmärrätkö nyt miksi koneeseen meneviä erityyppisiä raaka-aineita ei voi syöttää koneeseen yhdessä ja samassa listassa?” jatkaa Selma.

”Aaah. Ymmärrän”, sanoo Pate. ”Se on siis sama asia kuin että ostoslistallakin on vain niitä asioita, joita kaupasta ostetaan eikä esimerkiksi autojen rekisterinumeroita tai tuttujen koirien puhelinnumeroita? Jos haluamme rakentaa tehtaaseemme koneen, joka saa kaksi ihan erityyppistä raaka-ainetta, meidän on käytettävä kahta argumenttia. Nyt mä tajusin”, toteaa Pate.

”Kyllä. Olet ymmärtänyt aivan oikein Pate.”, jatkaa Selma.

”Voimme siis välittää koneillemme erityyppisiä raaka-aineita (argumentit). Argumentit voivat olla yksittäisiä, vaikkapa lukuja tai leluja Selman lelulistalta. Argumentti voi olla myös lista -tyyppinen, kuten olemme nähneet. Lista-tyyppisessä argumentissa voimme välittää useita yksittäisiä arvoja, mutta yksittäiset arvot voidaan välittää myös kahden eri argumentin avulla. Argumentin tyyppi on täysin riippuvainen koneesta (funktio), johon olemme raaka-ainetta (argumentit) annostelemassa.”, valistaa Selma.

”Ahaa…”, tuumii Pate. ”Eli jos haluaisimme laskea vaikkapa kymmenen luvun keskiarvon, olisi ihan viisasta antaa keskiarvonlaskukoneelle lista lukuja sen sijaan, että meillä olisi kymmenen argumenttia, joiden avulla välittäisimme raaka-aineet koneellemme? Näinkö se siis on?”

”Se on just näin”, toteaa Selma. ”Argumentteja pitää pohtia ihan samalla tavalla kuin joudumme pohtimaan sitä, millaisia putkia meidän pitää koneeseemme rakentaa, jotta saamme raaka-aineet syötettyä koneen sisään. Sementintekokoneeseen olisi hyvä tulla yksi putki, jossa kulkee vettä ja toinen putki, jossa kulkee hiekkaa ja kolmas putki, jossa kulkee ainetta, joka seoksen kuivuessa sitoo muut aineet kovaksi sementiksi. Emme kuitenkaan voi välittää koneelle vettä ja hiekkaa samassa putkessa, sillä putki voisi mennä tukkoon. Puhumattakaan, että johtaisimme veden ja sementtijauheen samassa putkessa. Kun tuotanto pysähtyisi, voisimme olla varmoja, että putki tukkeutuu ja sementti kovettuu putken sisään. Erityyppisille aineille pitää olla jokaiselle omat putkensa.”

”Onpa ohjelmointi hauskaa, sehän on kuin suunnittelisi tehtaita ja koneita vaan päivät ja yöt! Minä laittaisin ainakin omaan sementintekokoneeseeni hienot kromatut putket, joita pitkin olisi raaka-aineiden hyvä virrata koneeseen”, innostuu Pate.

Tyypillisiä ohjelmoinnin raaka-aineita

”Millaisia raaka-aineita ohjelmoidessa yleensä sitten tarvitaan?” kysyy Pate.

”Aika usein tarvitaan kokonaislukuja ja tekstiä, kuten Selman lelujen nimiä tai henkilöiden nimiä”, vastaa Selma. ”Joskus voidaan argumenteiksi tarvita myös desimaalikuluja tai vaikkapa murtolukuja. Lisäksi lähes aina tarvitaan totuusarvoja `True` ja `False`.

Erilaisista alkioista koostuvia joukkoja kutsutaan tyypeiksi. Tyypit ovat monille ohjelmointikielille jotain hyvin tyypillistä. Laajemmissa ohjelmissa käytetään paljon erilaisia itse määriteltyjä tyyppejä, joilla voidaan kuvata erilaisia asioita, kuten autoja, ihmisiä, ilmapalloja, kirjanpidon vientejä, palkkaa tai mitä tahansa muuta.”

”Miksi emme heti opettelisi miten ilmapallo -tyyppi määritellään? Mikään ei ole hauskempaa kuin ilmapallojen hätyyttäminen. Hauskinta niissä on se sähkö, joka saa saa takkuisemmankin huskyn karvat suoristumaan.”, veistelee Pate.

”Itse määriteltyjen tyyppien käyttäminen on kyllä hauskaa ja mielekästäkin. On kuitenkin niin, että funktionaalisen ohjelmoinnin perusasiat voi oivaltaa kyllä ilman niitäkin. Myönnän, että itse määritellyt tyypit avaavat kokonaan uuden maailman, sillä käytännön ohjelmointitöissä tarvitaan tämän tästä itse määriteltyjä tietotyyppejä.”

”Ok!” sanoo Pate ja jää miettimään, miten ilmapalloja voisi kuvata omilla tietotyypeillä.

Luku 4. Luvut koneiden raaka-aineina

”Kokonaislukujen tyyppi voi sisältää äärettömän määrän kokonaislukuja tai rajatuman lukujoukon”, aloittaa Selma. ”Kokonaislukujen tyypille on ominaista, että lukuja voidaan käyttää raaka-aineena erilaisille koneille, kuten yhteenlaskukoneelle. Kirjoitamme ensin koneen (funktion) nimen, joka on yhteenlaskun tapauksessa `+`. Laitamme vielä koneen nimen sulkeisiin ja lisäämme loppuun yhteenlaskettavat eli raaka-aineet (argumentit) välilyönnillä erotettuina.

```
>(+) 2 3
```

```
=> 5
```

Rakenne on täysin sama kuin `sort`-esimerkissämme siltä osin, että rivin alussa on koneen nimi ja raaka-aineet seuraavat:

```
> sort selmanLelut
```

Konetta tarkoittava nimi sattuu yhteenlaskukoneen tapauksessa olemaan + eikä kokonainen sana, kuten `sort`. Koodirivit eroavat myös niin, että + koneeseen syötetään kaksi lukua ja `sort`-koneeseen yksi lista. Koneisiin siis annostellaan erilaiset raaka-aineet.

Koska `sort`-koneen nimen jälkeen on `selmanLelut`-koneen nimi, niin voidaan ajatella myös, että +-koneen jälkeen tuleva luku 2 onkin itse asiassa sellaisen koneen nimi, joka tuottaa arvon 2. Kakkonenkin voidaan Pate siis ajatella koneeksi, joka tuottaa arvon 2.”

”Mistä voidaan tietää, että nimi `selmanLelut` tarkoittaa juuri listaa eikä jotain muuta?” kysyy Pate mietteliäänä. ”Eihän `selmanLelut` nimi paljasta mitään rakenteestaan – hakasulkeita tai muita listaan viittaavia piirteitä ei näy. Miten tämä nyt oikein toimii?”

”No johan sinä Pate kysymyksen lykkäsit. Lyhyt vastaus on, että viisas ohjelmointikieli osaa päätellä jo koneiden suunnitteluvaiheessa (koodia kirjoittaessa) ennen koneiden käynnistämistä, että `selmanlelut` on lista, joka sopii raaka-aineeksi `sort`-nimiseen koneeseen. Pitkä vastaus olisi niin pitkä, että siihen pitäisi palata kokonaan toisessa kirjasessa.”

”Voitaisiinko luvut 2 ja 3 niin ikään korvata joillain nimillä, jotka tarkoittavat joitakin lukuja? Onko siis niin, että mikä tahansa nimi kelpaisi, kunhan se tarkoittaisi jotain lukua?”, kysyy Pate.

”Juuri niin Pate”, vastaa Selma. ”Voimme määritellä kaksi nimeä `ekaluku` ja `tokaluku` ja korvata yhteenlaskukoneessa luvut 2 ja 3 näillä nimillä. Katsohan Pate.”

```
> ekaluku = (-) 5 2

> tokaluku = (*) 5 4

> (+) ekaluku tokaluku

=> 23
```

”Ja kuulehan Pate vielä tämä. On myös yhdentekevää, kuinka monen muun eri koneen työvaiheen jälkeen +-koneeseen syötettävät raaka-aineet siihen annostellaan. Pääasia on, että edellisistä koneista saadaan oikean tyyppisiä raaka-aineita yhteenlaskukoneeseen. Esimerkissä `ekaluku` syntyy vähennyslaskukoneen tuotoksena ja `tokaluku` syntyy kertolaskukoneen tuotoksena. Koska sekä kertolaskukoneesta että vähennyslaskukoneesta saadaan tuotoksina lukuja, sekä `ekaluku` että `tokaluku` kelpaisivat yhteenlaskukoneenkin eli jo meille tutun +-koneen raaka-aineiksi. Katsopa tätä, Pate. Eikös olekin mukavan ketjumaista?”

```
> yhteenlaskukone = (+) ekaluku tokaluku

> yhteenlaskukone

=> 23
```

”No todellakin on”, tuumaa Pate. ”Se tässä nyt kiinnostaa, että miten me saataisiin tuo `yhteenlaskukone` toiseen koneeseen jatkokäsittelyyn. Miten ketjua jatkettaisiin niin, että luku `yhteenlaskukone` voitaisiin taas annostella raaka-aineena johonkin sellaiseen koneeseen, joka osaisi sen käsitellä? Miten esimerkiksi yhteenlaskukoneen lopputulos saataisiin ketjutettua koneeseen, joka kertoisi yhteenlaskukoneen lopputuloksen kahdella. Ja jos olisi vielä niin, että haluaisimme käyttää yhteenlaskukoneen lopputulosta useammassa eri paikassa ohjelmakoodia, meidän kannattaisi nimetä tuo yhteenlaskukone. Kerrohan Selma minulle, miten se temppu tehtäisiin, niin lupaan tarjota sinulle jäätelön illalla.”

”Näin se Pate menisi. Katsohan tänne!

```
> kahdellakertoja = (*2) yhteenlaskukone

> kahdellakertoja

=> 46
```

Kerrotaanpa sama ohjelmoinnin käsittein ilman koneita. Nyt Pate tarkkana kuin porkkana!

`kahdellakertoja` on nimi, jota käyttämällä voimme kertoa kahdella funktion `yhteenlaskukone` arvon.

`yhteenlaskukone` on nimi, jota käyttämällä voimme laskea yhteen nimiä `ekaluku` ja `tokaluku` tarkoittavat arvot.

`ekaluku` on nimi, joka tarkoittaa arvoa, joka saadaan kun 5:stä vähennetään 2.

`tokaluku` on nimi, joka tarkoittaa arvoa, joka saadaan kun 5 kerrotaan 4:llä.

”Ja kuulehan Pate vielä tämä. Kuten esimerkistä näet, kun kirjoitat komentokehotteeseen `kahdellakertoja` ja painat etutassullasi return-näppäintä, tehtaan koneisto käynnistyy. Ensin selvitetään tietysti arvot `ekaluku` ja `tokaluku`, koska niitä tarvitaan seuraavassa vaiheessa. Sen jälkeen selvitetään arvo `yhteenlaskukone` ja viimeisenä selvitetään arvo `kahdellakertoja`.”

”Onko ohjelman toiminta siis eräänlaista koneiden tuotosten selvittelyä ja tuotosten siirtämistä aina seuraavaan käsittelyvaiheeseen?” kysyy Pate uteliaana.

”Se on Pate juuri sitä”, vastaa Selma. ”Mutta käsittelyvaiheesta toiseen siirryttäessä pitää tyyppien kanssa olla erityisen tarkkana. Jos kone odottaa, että se saa tiettyä putkea pitkin vettä, niin kone menee takuulla rikki, jos samassa putkessa johdetaankin koneeseen hiekkaa. Meidän pitää siis miettiä hyvin tarkkaan, mitä mihinkin koneeseen menee ja mitä mistäkin koneesta tulee ulos. Tämä siksi, että loppujen lopuksi meillä on aina koneita, jotka toimivat yhdessä muiden koneiden (funktioiden) kanssa. Muista, että yhden koneen lopputulos on jonkin toisen koneen raaka-ainetta!

Yhteenlaskukoneen nimi + voidaan lisätä myös raaka-aineiden väliin:

```
> 2+3

=> 5
```

Tämä helpottaa ohjelmakoodin tulkintaa, sillä yhteenlaskumerkki + on totuttu näkemään argumenttien välissä eikä sulkeissa kahden argumentin edessä.

Kokonaisluvut ovat hyvää raaka-ainetta myös jako- ja kertolaskukoneille. Kokonaisluku sopii raaka-aineeksi myös koneelle, jonka lopputuloksena on koneeseen raaka-aineena syötetty luku, jonka etumerkin kone muuttaa. Plussista tulee miinuksia ja miinuksista plussia.

Kokonaisluku on sopivaa raaka-ainetta myös koneelle, joka osaa korottaa kokonaisluvun haluttuun potenssiin. Kokonaisluku käy raaka-aineeksi myös koneelle, jonka tuotoksena on neliöjuuri. Kuulostaako tämä järkevältä Pate?”

”Kyllä, kaikki tähän mennessä tuntuu koiranjärkeen käyvältä. Yksi asia kuitenkin on, mitä en ollenkaan tajua. Nyt näyttää siltä, että me voimme käyttää kahdellakertomiskonetta vain ja ainoastaan niin, että raaka-aineena annostellaan aina yksi ja sama `yhteenlaskukone`. Eikö olisi järkevämpää, jos voisimme annostella kahdellakertomiskoneeseen muutakin kuin tuon iänikuisen `yhteenlaskukone`en tuottaman arvon?”

”Nyt olet todellakin asian ytimessä”, vastaa Selma innostuneesti.

”Totta kai voimme ja se käy helposti. Näin se käy.”

```
> kahdellakertoja = (*2)

> kahdellakertoja 5

=> 10
```

”Siis mitäs tuo nyt sitten oikein tarkoittaa”, ihmettelee Pate. ”Miten tuo oikein toimii?”, kysyy Pate.

”Se toimii ihan niin kuin muutkin koneet tähän asti. Katsohan. Kahdellakertomiskoneemme `kahdellakertoja` saa komentoriviltä annosteltuna raaka-aineena toisen kerrottavan 5 ja se tuottaa esimerkin tapauksessa arvon 10.”

Funktio on funktio myös ohjelmoidessa

”Koulumatematiikasta Pate muistanet yksinkertaisen funktion $y = 2x$ tai toisin ilmaistuna $f(x) = 2x$. Kun edellä mainittuun funktioon sovelletaan arvoa 5 näyttää se matematiikan kielellä tältä: $f(5) = 2 * 5 = 10$.

Jos nyt muutamme funktion nimen f nimeksi `kahdellakertoja`, matemaattinen esitys on kahdellakertoja (x) = 2x. Jos vielä lisäämme kertomerkin matemaattiseen esitykseen saamme kahdellakertoja (x) = 2*x. Jos nyt vertaamme ohjelmakoodia ja funktion määrittelyä huomaamme, että ne muistuttavat toisiaan.”

```
> kahdellakertoja = (*2)
```

kahdellakertoja (x) = 2*x

”Ne todellakin näyttävät aika lailla samoilta mutta kyllä tuossa Selma eroakin on”, sanoo Pate ja jatkaa. ”Matemaattisessa esityksessä on tuo äksä, mutta ohjelmakoodissamme sitä ei ole. Mistä tämä johtuu?”, kysyy Pate.

”Hyvä kysymys, Pate”, vastaa Selma. ”Se johtuu siitä, että koneemme osaa päätellä asioita. Kun kirjoitamme (*2) kone ymmärtää, että tarvitsemme äksän siihen koneeseen raaka-aineeksi eli toiseksi kerrottavaksi ilman, että asiasta tarvitsee erikseen ohjelmakoodissa mainita. Sen takia sitä ei tarvitse siihen erikseen kirjoittaa. Siitä huolimatta, niin halutessamme, voimme sen siihen kirjoittaa. Katsopa, niin vertaillaan taas ohjelmakoodia ja matemaattista esitystä.”

```
> kahdellakertoja x = (*2) x
```

kahdellakertoja (x) = 2*x

”No näyttävätpä ne tosi samanlaisilta nyt”, hihkuu Pate. ”Nehän ovat melkein kuin kaksi marjaa, mutta pieniä eroja kuitenkin vielä on. Saisiko niitä näyttämään vielä hieman enemmän samannäköisiltä?”

”No kyllä ne saa”, jatkaa Selma. ”Tältä ne lopulta voisivat näyttää”.

```
> kahdellakertoja x = 2*x
```

kahdellakertoja (x) = 2*x

”Ja jos vielä lisäämme sulut yhtäsuuruusmerkin vasemmalla puolella olevan x:n ympärille saisimme:”

```
> kahdellakertoja (x) = 2*x
```

kahdellakertoja (x) = 2*x

”Ja nyt sekä tapa merkitä, että merkitys ovat samoja.”, jatkaa Selma. ”Näyttääkö Pate siltä, että tällä ohjelmointikielellä ja matematiikalla olisi jotain yhteistä?”

”No kyllä tosiaankin näyttää!” huudahtaa Pate innoissaan. ”Tämähän ON matematiikkaa, eikä vain näytä siltä.”

”Just niin, Pate”, jatkaa Selma. ”Niin on, jos siltä näyttää ja nyt tosiaankin näyttää siltä”, jatkaa Selma iskien silmää.

Kahdellakertomiskoneemme on monessa mielessä fiksu. Yhtäältä kahdellakertomiskonetta kuvatessa (ohjelmakoodia kirjoittaessa) ei tarvitse erikseen kirjoittaa äksää tai muutakaan kuvaamaan raaka-ainetarvetta, koska kone itse tietää odottaa jotain raaka-ainetta, koska kahdella kertominen edellyttää toista kerrottavaa. Toisaalta kone osaa myös odottaa tietyn tyyppistä raaka-ainetta, ei mitä tahansa tavaraa - se siis tietää itse millaista raaka-ainetta se tarvitsee, koska se tietää, että kahdella kertominen tarvitsee raaka-aineena luvun.

Kokeile vaikka! Jos yrität antaa sille raaka-aineena luvun sijaan merkin ’a’, tulee ongelmia.”

```
> kahdellakertoja 'a'

=> * Couldn't match expected type `Int' with actual type
`Char'
```

”Kuten Pate huomaat, homma ei toimi. Yritimme laittaa hiekkaa vesiputkeen ja juuri sitä me ei saada tehdä. Tietokoneet eivät ymmärrä tällaisia kepposia. Kone haluaa raaka-aineena lukuja ja jos yritämme annostella kahdellakertomiskoneeseen vaikkapa kirjaimen ’a’, se toteaa, että kirjainta ei voi käyttää raaka-aineena sellaisessa koneessa, joka kertoo lukuja kahdella. Fiksu vehnäterrieri ei sotke vesiputkia hiekalla tai hiekkaputkia vedellä. Palaamme tähän asiaan vielä myöhemmin. Lupaan sen Pate!

Koneemme osaa siis itse päätellä monia asioita. Kone ei siis päästä hiekkaa vesiputkeen, vaan estää moisen katastrofin. Sellaisista koneista me koirat aivan erityisesti tykkäämme, sillä kuten tiedät Pate, meille koirille sattuu aika ajoin kaikenlaisia vahinkoja ja kommelluksia. Tämä on meille koirille ihan paras mahdollinen tapa kirjoittaa ohjelmia ja kuvata kaikenlaisia hienoja koneita, sillä koneet saavat aina prikulleen juuri ne raaka-aineet, joita ne osaavat hyödyntää!

Jos haluamme rakentaa `yhteenlaskukone`-nimisen koneen, joka saa raaka-aineena kaksi lukua (argumentit), voisimme tehdä sen näin.

Ensin määrittelemme itse koneen näin

```
> yhteenlaskukone a b = a+b
```

tai näin

```
> yhteenlaskukone' a b = (+) a b
```

Sitten käynnistämme koneen kirjoittamalla koneen nimen ja antamalla raaka-aineet (argumentit) 5 ja 9:

```
> yhteenlaskukone 5 9
```

```
=> 14
```

Näin määrittelimme kaksi argumenttia sisältävän nimen, joka tarkoittaa kahden luvun summaa. Nyt voimme kuitenkin annostella yhteenlaskettavat koneeseen Haskell-tulkkia käyttäen luettelemalla ne koneen nimen jälkeen. Voiko tämä Pate enää olla selkeämpää?"

"No ei kyllä voi", vastaa Pate. "Tämähän on niin selkää kuin olla ja voi. Nyt yhteenlaskukoneella tai kahdellakertomiskoneella voi laskea millä tahansa luvuilla, eikä vain joillain ennalta sovituilla."

"Koneemme ovat nyt paljon yleiskäyttöisempiä ja niitä on nyt helppo muidenkin koirien käyttää omissa hauskoissa ohjelmissaan", jatkaa Pate innoissaan.

"Hyviä kahdellakertomiskoneita ja yhteenlaskukoneita tarvitaan aina", lisää Pate ja jatkaa: "Muistan hyvin päivän, jona olin ollut kunnon koira ja totellut koko päivän ja melunnut mahdollisimman vähän. Saman päivän iltana minulle luvattiin hyvästä käytöksestä seuraavan kynsien leikkuun yhteydessä kaksinkertainen kanaherkkuannos. Jos minulla olisi jo silloin ollut kahdellakertomiskone, niin olisin voinut helposti laskea kuinka monta kanaherkkua olisi ollut odotettavissa.

Nyt tilanne on korjaantunut – osaan vaatia vähintään sen, mitä kahdellakertomiskone minulle tietyllä argumentilla tuottaa. Näistä koneistahan on uskomattoman paljon hyötyä ihan käytännön kanaherkkupuuhissakin. Ei huono! Lisää tällaisia hienoja koneita ja ne kromiputket!” hihkuu Pate.

”Olemme jo niin pitkällä funktionaalisen ohjelmoinnin peruskäsitteissä, että voimme jättää hyvästit konemaisille rinnastuksille ja käyttää varsinaisia termejä. Mitäs sanoisit Pate, jos tästä lähtien kutsuisimme koneita funktioiksi ja raaka-aineita argumenteiksi turvautumatta rinnastuksiin?”

”No kyllä se sopii”, vastaa Pate. ”Tuo matematiikan funktion käsite tarkoittaa selvästi samaa asiaa kuin koneemme, joten voimme unohtaa koneet ja tarttua funktioita ja argumentteja sarvista!”

”Käytämme kuitenkin vielä sanaa tuotos kuvaamaan toisinaan funktion arvoa, mutta hiljalleen voisimme luopua siitäkin rinnastuksesta ja puhua jatkossa vain funktion arvosta” jatkaa Selma. Näin meille jää vain kourallinen käsitteitä, joiden avulla voimme kuvata ohjelmaa ja sen toimintaa. Osaatko Pate sanoa, mitkä ne ovat?”

”Enköhän”, vastaa Pate. ”Tärkeimmät asiat ovat funktio, funktion argumentit ja funktion arvo. Funktion arvo saadaan jollain, mitä kutsutaan funktion evaluoinniksi. Funktio evaluoidaan vasta, kun se on saanut riittävästi argumentteja. Tämä tarkoittaa sitä, että funktiolle voidaan antaa argumentteja yksi kerrallaan ikään kuin osissa ja vasta, kun funktio on saanut riittävästi argumentteja se evaluoidaan. Haskell-kielessä evaluointi on laiskaa. Näin minä sen Selma tiivistäisin. Menikö edes sinne päin?”

”Kyllä vaan menikin”, vastaa Selma toiveikkaana. ”Nyt voimme siirtyä vertailufunktioihin ja alamme rakentamaan älyä ohjelmiimme. Sitä ennen nappaamme kuitenkin yhdet kanaherkut, sillä näin kovan älyllisen puristuksen päätteeksi kuuluu saada kunnon palkinto vai mitä tuumaat Pate?”

”Ilman muuta napataan”, jatkaa Pate ja lipoo jo huuliaan kirsun kiiltäessä kevätauringossa.

Lukujen vertaaminen

”Lukuja on hauska laskea yhteen, mutta yksi tärkeä juttu tästä hommasta kuitenkin vielä puuttuu”, tokaisee Pate. ”Miten voisin vertailla noita lukuja? Miten voisin esimerkiksi vertailla viime viikolla ja tällä viikolla syömieni kanaherkkujen lukumääriä? Olisi nimittäin mukava tietää tuliko tällä viikolla syötyä enemmän kanaherkkuja kuin viime viikolla. Meidän vehnäterrierien on hyvä pitää linjoistamme huoli, ettemme joudu dieetille. Kertoisitko Selma, miten voimme ongelman ratkaista.”

”Toki kerron”, jatkaa vuorostaan Selma. ”Lukuja voidaan tietenkin vertailla vertailufunktioilla. Lukuja vertailevalle yhtäsuuruudenvertailufunktiolle eli lukujenyhtäsuuruudenvertailufunktiolle annostellaan kaksi lukua argumentteina ja vertailufunktio tuottaa arvon tosi tai epätosi sen mukaan ovatko argumentit samat vai ei. Vertailufunktioita on monenlaisia. Tässä tutustumme niistä kahteen.

Vertailufunktion idea on, että se tutkii onko jokin väittämä tosi vai epätosi. Vertailufunktioita on erilaisia, mutta varmasti tunnetuin niistä on lukuja vertaileva yhtäsuuruudenvertailufunktio, joka osaa vertailla ovatko kaksi lukua samat vai ei.

Sellaisen vertailufunktion nimi on == ja sitä käytetään samalla periaatteella kuin yhteenlaskufunktiotakin. Lukujen yhtäsuuruutta vertailevan funktion argumentit ovat samat kuin yhteenlaskufunktiossakin eli kaksi lukua. Funktio tuottaa kuitenkin erilaisen tuotoksen kuin yhteenlaskufunktio. Lukujenyhtäsuuruudenvertailufunktion tuotos on joko arvo epätosi (`False`) tai arvo tosi (`True`) sen mukaan, millaiset argumentit vertailufunktioon milloinkin annostellaan.

Huomaa Pate, että niin lukujenyhtäsuuruudenvertailufunktiota kuin kaikkia muitakin vertailufunktioita voidaan käyttää myös niin, että argumentit annostellaan funktion nimen == ympärille. Katsotaanpa esimerkkiä:”

```
> (==) 1 1

=> True

> (==) 1 2
```

```
=> False

> 1==1        -- argumentit funktion nimen ympärillä!

=> True

> 1==2        -- argumentit funktion nimen ympärillä!

=> False
```

”Joo. Ihan kiva juttu Selma tuo lukujenyhtäsuuruudenvertailufunktiokin, mutta minähän halusin sellaisen lukujensuurempikuinvertailufunktion, jonka avulla voisimme selvittää, sainko viime viikolla vähemmän kanaherkkuja kuin tällä viikolla. Miten sellainen tehtäisiin?”

”Olin niin innoissani yhtäsuuruudenvertailufunktiosta, että unohdin Pate koko kysymyksen. No näinhän se menisi”, jatkaa Selma. ”Vaihdetaan == funktion tilalle > funktio ja annostellaan taas argumentit funktioon”

```
> (>) 2 1

=> True

> (>) 1 2

=> False
```

”Äsh! Olisihan tuo nyt pitänyt arvata koiramaisella arvauksella, että vaihtamalla funktion nimeä asia hoituu”, puistelee Pate päätään ja rapsuttaa korvaansa.

”Muista Pate, että lisäksi on vielä muitakin lukujen vertailufunktioita. Tunnetuimmat niistä jo nähtyjen lisäksi lienevät /= -funktio ja < -funktio. /= on erisuuri kuin -funktio, joka tuottaa tosi, jos argumentit ovat erisuuret. < -funktio tuottaa tosi vain, jos ensimmäinen argumentti on pienempi kuin toinen.”

Luku 5. Merkit ja merkkijonot funktioiden argumentteina

”Kokonaisluvut käyvät siis moneen, mutta mietihän Pate, millaisiin funktioihin kävisivät argumenteiksi merkit ’a’ tai ’6’? Merkit ovat ohjelmoinnissa tärkeitä, sillä monissa ohjelmointikielissä teksti muodostuu listasta merkkejä. Yksittäinen merkki on esimerkiksi ’ä’ tai ’3’. Kun ohjelmassa halutaan kuvata merkkiä, laitetaan merkin ympärille heittomerkit. Muista, että yksittäiset merkitkin ovat funktioita, vaikkakin aika pieniä sellaisia. Heittomerkit merkin ympärillä kertovat, että kyseessä on funktio, joka tuottaa yksittäisen merkin.

Voisimme myös kysyä, mitä sellaisia funktioita on tai voisi olla, jotka käyttävät argumentteinaan yksittäisiä merkkejä?

Ensin tulevat varmaankin mieleen funktiot, jotka muuttavat pieniä kirjaimia isoiksi kirjaimiksi tai isoja kirjaimia pieniksi kirjaimiksi. Katsohan Pate, miltä yksittäisiä merkkejä käsittelevät funktiot näyttävät. Tältä näyttää `toLower`-niminen funktio, joka muuntaa yksittäisen kirjaimen pieneksi kirjaimeksi:

```
> toLower 'S'

=> 's'
```

toUpper-niminen funktio sen sijaan muuntaa yksittäisen kirjaimen isoksi kirjaimeksi:

```
> toUpper 's'

=> 'S'
```

Merkkien vertaaminen

”Tiesitkö Pate, että yksittäisiä merkkejä argumentteinaan käyttäviä funktioita on lukuisia. Mielenkiintoinen on myös funktio, jonka avulla voimme tutkia onko ovatko argumentteina funktioon annostellut kaksi merkkiä samoja. Kyseessä on merkkienyhtäsuuruudenvertailufunktio, mihin voimme annostella kaksi merkkiä argumentteina ja selvittää ovatko merkit samoja.

Sellaisen merkkienyhtäsuuruudenvertailufunktion nimi on == ja sitä käytetään samalla periaatteella kuin lukujenyhtäsuuruudenvertailufunktiota. Funktion argumentit ovat kuitenkin eri tyyppiä kuin lukujenyhtäsuuruudenvertailufunktiolla. Molemmat funktiot tuottavat kuitenkin samoja arvoja tosi tai epätosi.

Merkkienyhtäsuuruudenvertailufunktion tuotos on aina joko arvo epätosi tai arvo tosi sen mukaan, millaiset argumentit vertailufunktioon milloinkin annostellaan. Katsotaanpa esimerkkiä:

```
> (==) 'a' 'a'

=> True

> (==) 'a' 'b'

=> False
```

”Yksi asia tässä Selma minua vähän vielä ihmetyttää”, sanoo Pate. ”Miksi me käytämme kirjainten vertaamisessa sanaparia yhtä suuri. Emme me kai voi sanoa, että jokin kirjain on isompi kuin jokin toinen kirjain. Vaikka kirjain olisikin ISOLLA kirjoitettu, emme me silti sano, että se on suurempi kuin jokin toinen kirjain. Emme ainakaan samassa tarkoituksessa kuin jos sanomme, että luku 5 on suurempi kuin luku 2. Mistä tässä on oikein kyse. Olisiko parempi sanoa, että kirjaimet ovat samoja tai että kirjaimet eivät ole samoja. En oikein tajua tätä yhtäsuuruusjuttua. Kertoisitko Selma minulle siitä vielä hieman tarkemmin.”

”Toki kerron kuomaseni”, jatkaa Selma. ”On totisesti hauskaa pohtia mitä tarkoittaa, että kahden asian sanotaan olevan samoja tai yhtä suuria.

Se on nimittäin täysin määrittelykysymys. Ihan samalla tavalla sinä Pate voisit laittaa ystäväsi paremmuusjärjestykseen. Onko Topi sinulle parempi kaveri kuin Selma vai ovatko ne sinulle yhtä hyviä kavereita? Kun haluat tietää kumpi kavereistasi on parempi vai ovatko kaverit kenties yhtä hyviä, tarvitset tietysti kaveruudenparemmuudenvertailufunktion ja kaveruudenyhtäsuuruudenvertailufunktion.

Nuo olisivat niin viisaita funktioita, että pystyisivät kertomaan kumpi kahdesta kaveristasi olisi parempi vai olisivatko kaverisi ehkä yhtä hyviä. Se olisi varsinaista tekoälyä, eikös vain?” jatkaa Selma ja iskee silmää. ”Sinun pitäisi Pate kuitenkin määritellä tuo paremmuusjärjestys nyt ystävien välillä. Ystävien laittaminen paremmuusjärjestykseen on kyllä sellaista puuhaa, että se kannattaa tehdä salassa ystäviltä, paitsi tietysti siltä parhaalta ystävältä. Kukapa ei haluaisi tietää olevansa jonkun paras ystävä.

Numeroiden osalta on Pate sovittu, että lukujonossa myöhemmin tulevat luvut ovat suurempia kuin aiemmin tulevat. Jonossa {1,2,3,4…} nelonen on suurempi kuin ykkönen, koska se tulee jonossa myöhemmin. Sama pätee kirjaimiin. Kun vertaamme kirjaimia ’a’ ja ’z’, ’z’ on suurempi, koska se tulee aakkosissa a:n jälkeen {’a’,’b’,…’z’}. Lukujonossa samalla kohtaa olevista luvuista toteamme, että ne ovat yhtä suuria.

```
> 'a'>'z'

=> False

> 'a'<'z'

=> True
```

Näillä tiedoilla pääsemme hyvin alkuun ja voimme jo rakentaa hyvin monenlaisia funktioita. Koska me koirat emme muista pitkiä salasanoja, niin mehän voisimme Pate rakentaa vertailufunktion, joka vertailisi kahta merkkiä ja kertoisi muistimmeko yhden merkin mittaisen salasanan oikein.

Funktion voisi nimetä niin, että nimi kuvaisi mahdollisimman hyvin funktion luonnetta, nimettäköön funktio `salamerkinvertailu`ksi. Kuulostaako hyvältä Pate? Onko riittävän koiramainen nimi?”

”Kuulostaa ihan huipulta, Selma”, jatkaa Pate. ”Nyt vaan kirjoittamaan ohjelma!”

```
> salamerkinvertailu = (==)'p'

> salamerkinvertailu 't'
```

```
=> False

> salamerkinvertailu 'p'

=> True
```

”Sehän toimii!”, hihkuu Pate. ”Jos annan argumenttina merkin t funktio vertailee sitä merkkiin p ja näin saan vastaukseksi epätosi. Jos taas annan argumenttina merkin p, funktio vertailee sitä merkkiin p ja näin saan vastaukseksi tosi . Tämähän on todellinen tietokone”, hehkuttaa Pate.

”No niinhän se tosiaan onkin”, sanoo Selma ja jatkaa: ”Koska salamerkinvertailu-funktio on saanut vasta yhden argumentin ’p’, se odottaa vielä toista argumenttia ennen kuin se voi vastata kysymykseen ovatko kaksi merkkiä samanlaisia vai erilaisia. Kun toinen argumentti sitten annostellaan koneeseen, funktion arvo selvitetään ja näin ollen saamme tietää ovatko arvot samat vai ei.”

.

Merkkijonot funktioiden argumentteina

”Merkkien muodostamaa yhden tai useamman merkin listaa kutsutaan merkkijonoksi. Vaikka merkkien listaa kutsutaankin merkkijonoksi, on se silti merkeistä muodostuva lista eikä jono. Sanoja ympäröivät lainausmerkit edustavat funktiota, joka luo yksittäisistä merkeistä muodostuvan listan. Keksisitkö Pate jotain fiksua käyttöä merkkijonoille?” kysyy Selma.

”Voin hyvin keksiäkin”, vastaa Pate miettien. ”Vaikka tassunjälki onkin jokaisen koiran nimikirjoitus, niin olisihan se hienoa, kun voisi vaikka herkkuluuhun painattaa omat nimikirjaimet. Tämän jälkeen kaikki koirat tietäisivät mikä olisi kenenkin luu, eikä tulisi turhia herkkuluuriitoja”, jatkaa Pate. ”Miten sellainen funktio sitten pitäisi tehdä?” kysyy Pate vuorostaan.

”Jos funktioon annosteltaisiin argumentteina etunimi ja sukunimi, voisi funktio poimia molemmista ensimmäiset kirjaimet ja tuottaa nimikirjaimet, joiden välissä olisi kirjaimia erottamassa piste. Koska merkkijonot ovat listoja pitäisi funktion osata ottaa molempien listojen alkupäästä alkiot ja yhdistää ne uudeksi merkkijonoksi. Sellaiset nimikirjaimet olisi mukava painattaa herkkuluun kylkeen.

Koska merkkijonot ovat listoja, pitäisi siis ensin löytää funktio, joka osaisi tuottaa listan ensimmäisen alkion. Sellainen funktio on `head`-funktio. Vieläkö Pate muuten muistat, miten saimme aikaan funktion, joka osasi luoda listan?”

”Eikös ne olleet ne hakasulkeet, joiden väliin lueteltiin ne listan alkiot”, vastaa Pate tomerana. ”Voisimme käyttää kahta `head`-funktiota, joista ensimmäinen saisi argumenttina etunimen ja toinen saisi argumenttina sukunimen. Kun annostelemme funktiot pilkulla erotettuina hakasulkujen väliin, niin meidän pitäisi saada tuotoksena lista, jossa nämä kirjaimet ovat. Sitähän me ajamme takaa eikös vaan? Mutta miltä tuollainen ohjelmakoodi Selma sitten näyttäisi?”

”Se näyttäisi tältä”, vastaa Selma.

```
> nimikirjaimet e s = [head e,head s]

> nimikirjaimet "Selma" "Erkkilä"

=> "SE"
```

”Mutta hei! Eikös meidän pitänyt myös lisätä piste noiden kirjaimien väliin, vai miten se nyt meni?” kysyy Pate tarkkaavaisena.

”Ai niin pitikin. Hyvä huomio Pate. Ohjelmakoodi pitää kirjoittaa määritysten mukaan.”, toteaa Selma. ”Ei hätää, näin asia korjataan. Ohjelmakoodin voisi kirjoittaa näin:”

```
> nimikirjaimet' e s = [head e,'.',head s]

> nimikirjaimet' "Pate" "Ilonen"

=> "P.I"
```

”Jos innostuisimme tekemään tekstinkäsittelyohjelman, vastaan tulisi varmasti tilanteita, joissa kaksi erillistä sanaa pitäisi liittää yhteen ja muodostaa näistä uusi sana. Silloin saattaisimme tarvita funktiota, joka tuottaisi kahdesta merkkijonosta yhden.

Sellainen funktio on ++ - funktio. ++ -funktio saa argumentteina kaksi merkkijonoa tuottaen kolmannen. Funktiota voi kokeilla näin:

```
> (++) "alkupää" "loppupää"

=> "alkupääloppupää"
```

tai näin

```
> "alkupää"++"loppupää"

=> "alkupääloppupää"
```

Merkkijonojen yhdistäminen on siis kahden listan yhdistämistä. Seuraavassa luvussa tutustumme listoihin tarkemmin. Kun osaamme käsitellä kokonaislukuja, merkkejä, merkkijonoja ja listoja, voimme luoda koiraystäviemme kanssa monenlaisia sovelluksia. Nyt kun Pate tiedät, miten `nimikirjaimet`-funktio tehdään, voit opettaa sen myös Topi-koiralle tai vaikka Orvokille.”

”No ilman muuta opetan `nimikirjaimet`-funktion myös Topille ja Orvokille, heti kun iltalenkillä näemme. Ei enää herkkuluuriitoja, vaan rauha maassa koirien kesken”, vitsailee Pate. ”Funktionaaliselle ohjelmoinnille tuntuu löytyvän vaikka kuinka paljon käytännön sovelluksia. Tietävätköhän ihmiset, miten paljon käyttöä näille jutuille oikeasti on? ”, jatkaa Pate ja pistää poskeensa mehevän kanaherkun.

Merkkijonojen vertaaminen

”Olisi muuten Selma hyvä, jos voisimme kerralla vertailla kokonaisia merkkijonoja eli merkkien muodostamia listoja. Onhan sellaisiakin funktioita olemassa?”

”Kyllä vain on ja montakin”, vastaa Selma. ”Jos annostelet funktioon koiramaisen vaikean useammasta merkistä muodostuvan salasanasi, niin onhan meidän tutkittava onko salasana oikein vai ei. Tuo tutkiminen edellyttää funktioon annostelemasi salasanan vertaamista funktioon jo annettuun salasanaan – siis kokonaisen sanan vertaamista toiseen kokonaiseen sanaan. Tältä se voisi näyttää.”

Määritellään salasanavertailufunktio näin:

```
> salasanavertailu = (==)"kanaherkku"
```

Annostellaan salasanavertailufunktioon ”nappulat”.

```
> salasanavertailu "nappulat"

=> False
```

Annostellaan salasanavertailufunktioon ”kanaherkku”.

```
> salasanavertailu "kanaherkku"

=> True
```

”Kahden merkkijonon yhtäsuuruus voidaan siis selvittää käyttämällä == -funktiota aivan samalla tavalla kuin merkkien yhtäsuuruutta tutkittaessa tai kahden luvun yhtäsuuruutta tutkittaessa. Koska salasanavertailufunktio on saanut vasta yhden argumentin ”kanaherkku”, se odottaa vielä toista argumenttia ennen kuin se voi vastata kysymykseen ovatko kaksi merkkijonoa samat vai ei. Kun toinenkin argumentti sitten annostellaan funktioon, funktion arvo voidaan selvittää ja näin ollen kertoa ovatko arvot samat vai ei.”

Lisää älyä

”Moi taas Selma! Aloitetaanko taas miettimään ohjelmointijuttuja. Mitäpä tänään olisi tarjolla?”

”Moi Pate!”, vastaa Selma iloisesti. ”Tänään voisimme käydä läpi sitä, miten saisimme funktioihimme hiukan enemmän älyä. Mitäs tykkäisit sellaisesta suunnitelmasta?”

”No se sopii hyvin! Meillä koirilla ei ole koskaan liikaa älyä kirjoittaa älykkäitä ohjelmia. Aloitetaan!”

”Näin tehdään”, jatkaa Selma. ”Älykkyys syntyy siten, että funktiomme voi saada aikaan erilaisia tuotoksia riippuen siitä, millaisia argumentteja niihin annostellaan. Nyt esittelen sinulle herkkujenarviointifunktion, joka tietää, kuinka herkullisia eri herkut ovat. Katsotaanpa ohjelmakoodia.”

```
arvioiHerkut herkku
 |herkku=="kanaherkku" = "Erinomainen ja maittava"
 |herkku=="maksalaatikko" = "Aika maukas"
 |herkku=="nappulat" = "Ei niin hyvä, mutta terveellinen"
```

”Hetkinen. Nyt on taas uusia merkkejä ja kaikenlaista”, haukahtaa Pate. ”Tuon == -nimen minä kyllä tunnistan. Sehän on yhtäsuuruudenvertailufunktio! Lisäksi tuttua on, että tuossa on taas tuo yhtäsuuruusmerkki ja sen oikealla ja vasemmalla puolella on asioita. Mutta mikä kumma on tuo pystyviiva noiden kolmen rivin edessä, sitä minä en ole koskaan nähnytkään?”

”Hyvinpä olet Pate lukenut kirjasen aiemmat osat”, kehuu Selma. ”Tuo pystyviiva on vahti. Sen avulla voimme tutkia arvoja ja päättää, mikä funktio milläkin arvolla valitaan seuraavaksi. Jos vertailufunktion tuotoksena on arvo tosi, niin valitsemme samalla rivillä yhtäsuuruusmerkin oikealla puolella olevan funktion. Jos funktion argumentti on ”kanaherkku”, niin yhtäsuuruudenvertailufunktio selvittää ensin onko argumentti ”kanaherkku” sama kuin ”kanaherkku” ja vertailu tuottaa tietysti arvon tosi ja seuraavaksi valitaan tuotokseksi ”Erinomainen ja maittava”. Kokeilepa, niin ymmärrät.”

```
> arvioiHerkut "kanaherkku"
=> "Erinomainen ja maittava"
> arvioiHerkut "maksalaatikko"
=> "Aika maukas"
```

”Vahtien avulla voimme tutkia arvoja ja ohjata ohjelmamme toimintaa. Katsopa vielä tämä toinenkin esimerkki.”

```
herkkujaViikossa x
 |x<1 = "Et ole syönyt ollenkaan herkkuja!"
 |x>=1 && x < 4 = "Olet syönyt jonkin verran herkkuja!"
 |x>=4 = "Olet syönyt aivan liikaa herkkuja!"
> herkkujaViikossa 4
=> "Olet syönyt aivan liikaa herkkuja!"
> herkkujaViikossa 0
=> "Et ole syönyt ollenkaan herkkuja!"
```

”No jopas on taas kaikenlaista!”, huokaisee Pate. ”Mikä on muuten tuo kaksi & merkkiä peräkkäin? Sitä en ole kuunaan nähnyt.”

”Olipa hyvä kysymys Pate!”, sanoo Selma. ”Kuten huomaat && -merkin molemmin puolin on vertailufunktio. && -merkki tarkoittaa myös vertailufunktiota. && tarkoittaa sellaista vertailufunktiota, joka tuottaa arvon tosi vain siinä tapauksessa, että sen molemmilla puolilla olevat vertailufunktiot tuottavat kumpikin arvon tosi.

&& on siis vertailufunktio, joka saa kaksi argumenttia, joita voimme tästä lähtien kutsua myös totuusarvoiksi (tosi tai epätosi). Meillä on olemassa toinenkin hyvin samantyyppinen vertailufunktio, jonka nimi on ||. || tuottaa arvon tosi vain siinä tapauksessa, että sen molemmilla puolilla olevista vertailufunktioista vähintään yksi tuottaa arvon tosi.

Tämä kaikki näyttää Pate vielä paljon selvemmältä, kun esitämme sen vanhassa tutussa muodossa eli ensin && -funktio ja sitten vasta kaksi muuta vertailufunktiota. Katsohan tätä!”

```
|(&&) (x>=1) (x<4) = "Olet syönyt jonkin verran herkkuja!"
```

”Nyt lienee päivänselvää, että && -funktioon annostellaan argumentteina kaksi vertailufunktiota. Sulkumerkkejä näyttää nyt olevan vähän enemmän kuin aikaisemmin, mutta niistä sinun ei kannata tässä kohtaa Pate murehtia. Logiikka paljastuu sinulle sulkeista huolimatta. && -funktio tuottaa oman logiikkansa mukaan joko arvon tosi tai epätosi kahden muun funktion tuotosten perusteella.

Jos ajattelemme evaluointihetkeä, niin kyseessä on siis kolmen funktion pötkö, joista ensimmäiseen annostellaan argumentteina kahden sitä seuraavan funktion tuotokset. Eikös ole helppoa!

Myös vahti eli |-merkki on funktio. Se on funktio, joka selvittää yhtäsuuruusmerkin oikealle puolella olevan arvon vain, jos sen oma argumentti on tosi. Tässä on nyt koiramaisen tärkeää huomata, että vahteja voi olla useampia kuin yksi. Jos meillä on kolme vahtia, kuten esimerkissämme, voimme haarautua ohjelmakoodissamme kolmeen suuntaan.

Mietipä tätä! Kun `arvioiHerkut` nimeä jossain kohtaa ohjelmaa käytetään, johtaa se siihen, että vain yksi kolmesta eri merkkijonon tuottavasta funktiosta valitaan. Arvoja tosi tai epätosi tuottavia funktioita kutsutaan myös predikaateiksi. Vahteja siis tarvitaan, jotta voimme haarautua ohjelmakoodissamme – valita yhden useammasta funktiosta, jonka arvo tulee selvitettäväksi seuraavana tuotantoketjussamme.

Näillä kahdella totuudenvertailufunktiolla && ja || ja vahteja ohjelmassasi viljelemällä voit lisätä ohjelmasi älyä vaikka kuinka paljon.”

Luku 6. Hahmonsovitusta ja kanaherkkuja

”Tiedätkö muuten Pate, montako koirarahaa nuo eri herkut maksavat?” kysyy Selma.

”En tiedä, koska en joudu niitä koskaan itse maksamaan.”, vastaa Pate ja hiukan luimistelee korviaan.

”Minäpä muuten tiedän. Katsopa, niin näytän taas ohjelmaa. Herkkuhinnaston tekeminen on kuin lasten leikkiä.”, hehkuttaa Selma.

```
hinta "kanaherkku" = 70
hinta "maksalaatikko" = 38
hinta "nappulat" = 22
```

”Siis onko herkkuhinnasto tosiaankin tuossa? Miten se sitten oikein toimii?” kysyy Pate.

”No herkkuhinnaston pitää olla tietysti koiramaisen helppokäyttöinen. Näin sitä Pate käytetään. Nyt kun herkkujen hinnat on määritelty, voidaan niitä kysyä kirjoittamalla hinta ja sen jälkeen herkun nimi.”

```
> hinta "kanaherkku"

=> 70

> hinta "maksalaatikko"

=> 38

> hinta "nappulat"

=> 22
```

”Se todellakin toimii”, sanoo Pate. ”Se näyttää todella yksinkertaiselta, mutta miten ne noin voi toimia? Miten voi olla kolme funktiota, joilla on sama nimi – `hinta`?”

”Kuulehan Pate! Oikeasti tuossa onkin vaan yksi `hinta`-niminen funktio. Sitten meillä on kolme vaihtoehtoa, jotka funktio tunnistaa eri argumenteiksi: ”kanaherkku”, ”maksalaatikko” ja ”nappulat”. Kirjoitetaanpa ohjelmakoodi vielä hieman selkeämmin käyttämällä case of -rakennetta, niin näet, että siinä todellakin on vain yksi `hinta`-niminen funktio.”

```
hinta herkunNimi = case herkunNimi of
     "kanaherkku" -> 70
     "maksalaatikko" -> 38
     "nappulat" -> 25
```

”Kuten näet Pate, meillä on todellakin vain yksi funktio, jonka nimi on `hinta`”, jatkaa Selma. ”Funktio saa argumenttinaan herkun nimen, joka voi vaihdella. Ohjelmointikielemme on niin viisas, että se osaa tunnistaa sisään tulevat argumentit ja valitsee oikean hinnan annostellun argumentin perusteella.

Kun argumentti eli herkun nimi annostellaan funktioon, alkaa se evaluointihetkellä sovittaa argumenttia lueteltuihin hahmoihin. Ensin tutkitaan, sopiiko argumentti hahmoon kanaherkku, jos sopii, niin jatketaan nuolen osoittamaan suuntaan. Jos argumentti ei sovi hahmoon, niin sovitetaan argumenttia seuraavaan hahmoon eli maksalaatikkoon. Jos argumentti sopii maksalaatikkoon, jatketaan nuolen osoittamaan suuntaan. Jos argumentti ei sovi maksalaatikkoonkaan sitä sovitetaan vielä nappuloihinkin. Jos argumentti sopii nappuloihin, jatketaan taas nuolen osoittamaan suuntaan. Jos argumentti ei sovi nappuloihinkaan, funktio tuottaa virheilmoituksen.

Osaisitko Pate muuten tehdä funktion, joka kertoisi meille herkun nimen, jos tietäisimme sen hinnan. Minäpä näytän, miten sellainen tehtäisiin, katsohan tänne!”

```
herkku herkunHinta = case herkunHinta of
     70 -> "kanaherkku"
     38 -> "maksalaatikko"
     25 -> "nappulat"
```

```
=> herkku 70
```

```
=> "kanaherkku"
```

”Nyt on Pate oleellista huomata, että me emme verranneet hintoja tai herkkujen nimiä, kuten vahtien kanssa. Me luotimme hahmonsovitukseen, jonka avulla saatoimme tunnistaa argumentin.”

”Häh!”, älähtää Pate. ”Eikös hahmonsovitus ole sitä, että tietokone voi tunnistaa kuvasta koiran ja kertoa sen ihmiselle?”

”No se on hahmontunnistusta, mutta idea on sama kuin hahmonsovituksessakin”, vastaa Selma. ”Tässä yhteydessä se on kuitenkin keino tehdä juuri tuollaisia temppuja, kuten äsken näimme. Jos funktion argumentti sopii hahmoon, niin valitsemme sen asian, joka on yhtäsuuruusmerkin tai case of -rakennetta käytettäessä nuolen oikealla puolella.

Hahmonsovitusjuttu on kuitenkin jotain paljon suurempaa, kuin mitä juuri näimme. Näytän myöhemmin, miten hahmonsovituksen avulla voitaisiin selvittää onko kanaherkkupussissa yksi vai useampi kanaherkku tai ei yhtään herkkua. Sitä ennen meidän pitää kuitenkin tutustua listan käsitteeseen.”

Luku 5. Kanaherkkupusseista listoihin

Listat ja listoihin liittyvät funktiot

”Listat ovat Pate ohjelmoijan tärkein työkalu. Listojen kanssa touhuaminen on hauskaa ja helppoakin. Tämä johtuu etenkin siitä, että meillä on paljon hienoja funktioita, joita voi soveltaa listoihin. Listan avulla on helppoa järjestellä niin kanaherkkuja kuin monia muitakin asioita. Aloitetaan siis laittamalla kanaherkut pussiin, jota ohjelmassamme voimme luontevasti kuvata listalla. Aluksi luomme tyhjän listan.”

```
> lista0 = []
```

”Kun haluamme lisätä siihen pari kanaherkkua, meidän tulee käyttää siihen tarkoitukseen sopivaa funktiota. Lisätään malliksi kanaherkut listalle yksi kerrallaan : -funktion avulla. : -funktio on funktio, jonka avulla voidaan lisätä listan alkuun uusi alkio ja samalla luoda kokonaan uusi lista, joka sisältää tuon lisätyn alkion ja jonka perässä on tuo aiemmin määritelty lista.”

```
> lista1 = "kanaherkku1":lista0

> lista2 = "kanaherkku2":lista1

> lista2

=> ["kanaherkku2","kanaherkku1"]
```

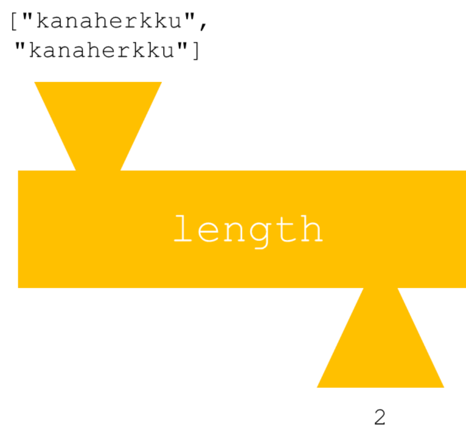
”Nyt Pate näemme, että listalla on kaksi kanaherkkua. Listaan loppuu voimme lisätä uusia alkioita ++ -funktiolla. Näin se käy:

```
> lista3 = lista2 ++ "kanaherkku3"

> lista3

=> ["kanaherkku2","kanaherkku1","kanaherkku3"]
```

Mutta mitäs jos haluamme tietää kuinka monta herkkua listalla on?”

”Jaa-a”, ihmettelee Pate. ”Voisiko siihen löytyä jokin pussissaolevienkanaherkkujen-lukumääränlaskufunktio, koska kaikkiin muihinkin asioihin on ratkaisuna aina jokin funktio?”

”Kyllä vaan Pate”, vastaa Selma. ”Olet todellakin hoksannut jutun jujun kuomaseni, ratkaisu on aina funktio, pitää vaan keksiä millainen. Katsopa tätä funktiota ja sen kuvaa. Funktioon annostellaan argumenttina lista kanaherkkuja ja funktio laskee, kuinka monta kanaherkkua argumenttina olevalla listalla on.”



Kuva 3. Kanaherkkuja laskeva funktio.

```
> length kanaherkkulista
=> 2
```

”`Length`-funktio on kätevä. Se saa argumenttina listan, jonka alkioiden lukumäärän se osaa laskea. Nyt on helppo pysyä kärryillä herkkujen lukumäärästä ja varmistaa, ettei naapurin koira ole käynyt sinun kanaherkkupussillasi salaa.

`Length`-funktion lisäksi on paljon muitakin funktioita, joiden argumentin tyyppi on lista. Muita tärkeitä listoja käsitteleviä funktioita ovat esimerkiksi `head`- ja `tail`-funktiot sekä !!-funktio.

!!-funktiolla saamme tuotettua listasta tietyn alkion. !!-funktion nimen perään annostellaan luku. Luvulla 0 tarkoitetaan listan ensimmäistä alkioita, luvulla 1 listan toista alkiota ja niin edelleen. Huomaathan Pate, että !!-funktio saa argumenttinsa funktion nimen edessä ja takana. Funktion nimen eteen laitetaan lista ja funktion nimen jälkeen laitetaan luku ja funktio tuottaa luvun mukaisen alkion.

```
> ["kanaherkku1","kanaherkku2","kanaherkku3"]!!0

=> "kanaherkku1"

> ["kanaherkku1","kanaherkku2","kanaherkku3"]!!1

=> "kanaherkku2"
```

`Head`-funktiolla saamme tuotettua listan ensimmäisen alkion:

```
> head ["kanaherkku1","kanaherkku2","kanaherkku3"]

=> "kanaherkku1"
```

`Tail`-funktiolla saamme tuotettua listan loppuosan ilman ensimmäistä alkiota:

```
> tail ["kanaherkku1","kanaherkku2","kanaherkku3"]

=> ["kanaherkku2","kanaherkku3"]
```

Huomaa, että `head` tuottaa alkion ja `tail` tuottaa listan.

```
> head ["kanaherkku1","kanaherkku2","kanaherkku3"]
```

”Aivan huippua”, toteaa Pate. ”Mutta nyt annan sinulle Selma sellaisen herkkuongelman, että jos siihen keksit ratkaisun, niin tarjoan sinulle kyllä kanaherkun kanaherkkupussistani. Oletetaan, että pussissani olisi muitakin herkkuja kuin kanaherkkuja. Miten saisimme tietää kuinka monta kappaletta kutakin herkkua olisi? Onnistuisimmeko kirjoittamaan sellaisen ohjelmakoodin?”

”No nytpä pistit pahan Pate!”, jatkaa Selma. ”Eiköhän tuohonkin kuitenkin keinot keksitä, pitää vaan keksiä lisää funktioita. Tuon ongelman voisi ratkaista `filter`-funktio. Siihen tutustumme seuraavassa luvussa.”

Listat ja hahmonsovitus

”Kuten lupasin, aion kertoa, miten listoja voi tutkia hahmonsovituksen avulla. Katsohan Pate seuraavaa esimerkkiä.”

```
selmanOstoslista = ["Kanaherkut", "Nappulat", "Maksalaa-
tikko","Koiransuklaa"]
tutkiOstoslista [] = "Ostoslista on tyhjä"
tutkiOstoslista [x] = "Ostoslistalla on vain 1 alkio"
tutkiOstoslista (x:xs) = "Ostoslistalla on enemmän kuin 1 al-
kio"
```

”Kuten näet, `selmanOstoslista` listalla on neljä alkiota. Jos `tutkiOstoslista`-funktiolle annostellaan argumenttina tyhjä lista eli `[]`, se osaa kertoa, että lista on tyhjä. Jos taas annostelemme `tutkiOstoslista`-funktiolle yhden alkion, jota kuvaa hahmo `[x]`, saamme funktion tuotokseksi ”Ostoslistalla on vai 1 alkio”. Jos lista ei ole tyhjä eikä siinä ole vain yhtä alkiota, valitsemme ”Ostoslistalla on enemmän kuin 1 alkio”, johon viittaa hahmo `x:xs`. Hahmo `x:xs`, viittaa ylipäänsä listaan, jossa on vähintään yksi alkio. Kokeile Pate kutsua funktioita tyhjällä listalla, listalla, jossa on arvoja ja listalla, jossa on useampia kuin yksi arvo, niin näet, miten se toimii. Tässä tapauksessa voisimme korvata viimeisen hahmon `x:xs` myös alaviivalla, joka tarkoittaa ”mikä tahansa arvo”, jolloin ohjelmamme näyttäisi tältä:

```
selmanOstoslista = ["Kanaherkut", "Nappulat", "Maksalaa-
tikko","Koiransuklaa"]
tutkiOstoslista [] = "Ostoslista on tyhjä"
tutkiOstoslista [x] = "Ostoslistalla on vain 1 alkio"
tutkiOstoslista _ = "Ostoslistalla on enemmän kuin 1 alkio"
```

”Tuohan on tosi kätevää!”, hihkuu Pate. ”Kun käytämme hahmonsovitusta, saamme siis jo hyvissä ajoin vihiä millaisia argumentteja funktioon on tulossa ja osaamme säätää funktiomme tuotoksen argumentin luonteen mukaisesti. Jos ostoslistalla on paljon asioita, voisimme ottaa kauppareissulle ison kassin mukaan. Jos ostoslista on tyhjä, emme tarvitsisi kauppakassia ollenkaan, sillä tuskin lähtisimme kauppaan laisinkaan. Mutta voisiko Selma joskus käydä niin, että menisimme kauppaan ilman kauppalistaa ja tekisimme heräteostoksia? Olisiko mahdollista että päähämme pälkähtäisi vasta kaupassa, mitä haluamme. Niinhän monet ihmiset toimivat. Voisimmeko me koiratkin ohjelmoidessamme toimia hetken mielijohteesta?”

”Nyt olet Pate asian luissa, ytimissä ja ydinluissa”, vastaa Selma. ”Meidän koiramaisissa esimerkeissämme ei ole tähän mennessä ollut hetken mielijohteita. Hetken mielijohdetta voitaisiin kuvata satunnaisuudella. Satunnaisuus on kuitenkin jotain, mitä emme vielä tässä vaiheessa opintojamme tarvitse – tehdään satunnaisia juttuja ja päähänpälkähdyksiä opiskelun myöhemmissä vaiheissa. Lupaan, että senkin aika vielä tulee ja pääsemme toimimaan kuten emäntämme ja isäntämme joskus tuntuvat toimivan, arvaamalla.”

”Ok! Tässä vaiheessa me siis vielä touhuamme funktioilla, joiden argumentit määrittelevät, mitä funktiosta tulee ulos. Ehkä on parempikin, ettemme vielä lähde laatimaan sattumanvaraista tuotoksia luovia funktioita. Onhan se ajatuksena hiukan pelottavakin, sillä onhan se mukava saada kanaherkku aina, kun menee kiltisti pyydettäessä maahan tai istumaan! Emännän tai isännän satunnainen kehu on tietysti kiva palkinto sekin, mutta kanaherkku on aina kanaherkku”, filosofoi Pate kuin Sokrates ikään.

”Yhtä asiaa haluan vielä hahmonsovituksesta sinulle Pate korostaa. Muista, että koodirivien järjestyksellä on merkitystä, kun mietit funktioosi annosteltavien argumenttien luonnetta. Jos laitat ensimmäiseksi vaihtoehdoksi _ eli ”mikä tahansa hahmo”, niin valituksi tulisi aina ” "Ostoslistalla on enemmän kuin 1 alkio". Tämä johtuu siitä, että lista, joka on tyhjä tai siinä on yksi alkio tai useampia, on aina ”mikä tahansa hahmo”. _ kuvaa siis tässä yhteydessä mitä tahansa hahmoa, joten sitä ei kannata tutkia ensimmäisenä! Saitko ajatuksestani kiinni Pate?”

”Aah. No tuo onkin tärkeä asiaa muistaa. Toimiiko logiikka samalla tavalla, jos tutkisimme hahmojen sijaan arvoja?” kysyy Pate uteliaana.

”Kyllä vaan”, vastaa Selma. ”Idea on arvoja tutkittaessa aivan sama. Ensin tutkitaan ne kaikkien yleisimmät tapaukset ja loput jätetään loppupäähän”, jatkaa Selma.

”Ai-van”, myhäilee Pate ja jatkaa ”Tästä sainkin hyvän muistisäännön kaikkeen arvojen tutkimiseen. Se pätee myös koiran elämään. Tee ensin tärkeät asiat ja jätä loput loppupäähän. Ja ne asiat, jotka tapahtuvat usein, kannattaa tutkia ensin. Se nopeuttaa ohjelman toimintaa!”

Luku 6. Filter-funktio

”Laitetaan Pate nyt listalla neljä hyvää herkkua, joista kaksi ovat samoja.”

```
herkkuLista = ["kanaherkku","kanaherkku","koiransuklaanappi",
"herkkuluu"]
```

”Meidän pitäisi Pate nyt valita listalta ensin kanaherkut ja laskea niiden lukumäärä. Sitten pitäisi valita listalta koiransuklaanapit ja laskea niiden lukumäärä. Lopuksi pitäisi vielä valita listalta herkkuluut ja laskea niiden lukumäärä. Jos Pate sinun pitäisi tehdä sellainen valintafunktio, joka osaisi valita listalta herkkuja, niin miten sinä sen tekisit?” kysyy Selma.

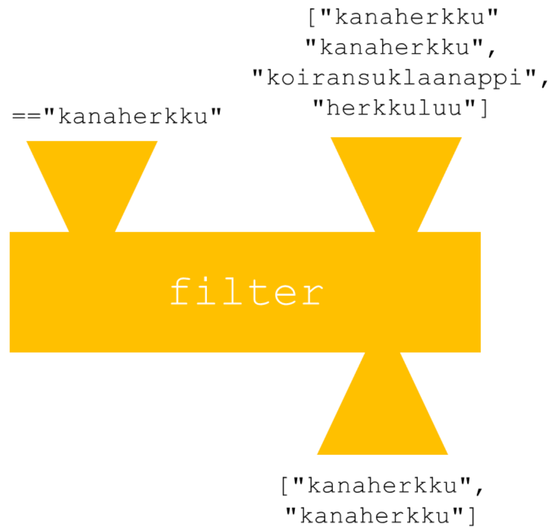
”Ensin tietysti kaikki herkut olisivat sekaisin yhdessä isossa herkkupussissa. Sitten valitsisin siitä pussista ensin tietysti parhaimmat eli kanaherkut ja laittaisin ne yhteen tyhjistä pusseista. Sitten valkkaisin koiransuklaanapit ja laittaisin ne toiseen tyhjään pussiin ja lopuksi laittaisin vielä herkkuluut kolmanteen pussiin ja alkuperäisen ison pussin laittaisin tietysti siististi eteisen kaappiin ostosreissua odottamaan. Lisäksi laittaisin jokaiseen pussiin tarran, jossa lukisi mitä herkkuja pussissa on.”

”Nyt kun tekisimme ohjelman, jossa jokaista pussia vastaisi lista ja sitten antaisimme jokaisen listan argumenttina `length`-funktiolle, niin eikös meillä sitten olisikin tieto siitä kuinka paljon eri herkkuja on? `length`-funktiohan pystyisi laskemaan eri pussien alkioiden eli herkkujen lukumäärät, eikös vaan?” kysyy Pate.

”Oikeassa olet!” vastaa Selma. ”Meidän pitää vain keksiä ensin, miten valintafunktio toimii. Keksitkö sinä Pate?”

”Voisin ainakin yrittää”, jatkaa Pate. ”Meidän pitää jotenkin kertoa valintafunktiolle, mitä halutaan valita. Pitää antaa siis jokin ehto, joka voisi olla muotoa ”ota pussista kaikki sellaiset, jotka ovat kanaherkkuja ja laita ne tyhjään pussiin”, vai mitä sanot Selma?”

”Juuri niin Pate. Sinähän olet jo tosi etevä. Katsotaanpa seuraavaksi millaiset funktiot saamme näistä ajatuksista aikaiseksi.”

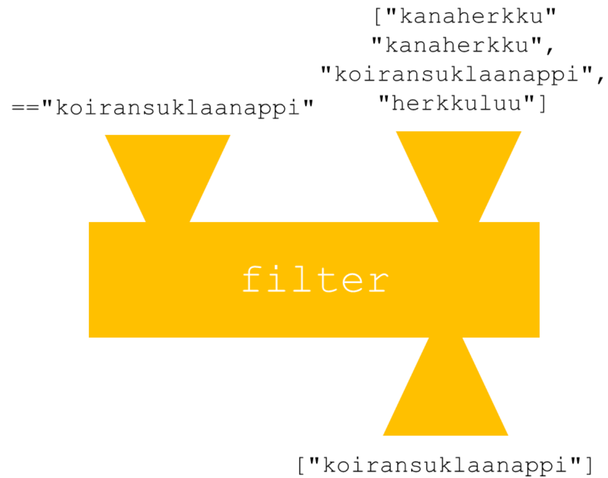


Kuva 4. Kanaherkkuja listalta poimiva funktio.

```
kanaherkkuLista = filter(=="kanaherkku") herkkuLista
```

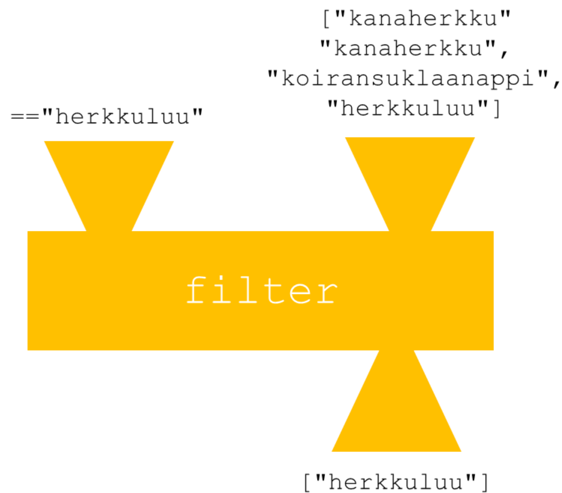
”Katsohan yllä olevaa kuvaa Pate”, jatkaa Selma. ”`Filter` on funktio, joka käy yksitellen listan alkioita läpi niin, että vertailufunktiota sovelletaan jokaiseen läpikäytävään alkioon. Joka kerta, kun yhtäsuuruudenvertailufunktio(`==”kanaherkku”`) tuottaa arvon tosi, vertailtava alkio poimitaan vanhalta listalta uudelle listalle. `Filter`-funktio saa siis tässä tapauksessa kaksi erilaista argumenttia: vertailufunktion ja listan, joiden alkioita halutaan käydä läpi.

Seuraavan kuvan alla oleva ohjelmarivi tarkoittaa, että poimitaan listalta kaikki alkiot, jotka sisältävät merkkijonon ”kanaherkku”. `Filter`-funktiolla luodaan siis uusi lista, jossa on vain ”kanaherkku” alkioita. Samaa ideaa toistetaan myös herkkuluille ja koiransuklaanapeille.”



Kuva 5. Koiransuklaanappeja listalta poimiva funktio.

```
koiransuklaanappiLista = filter(=="koiransuklaanappi") herk-
kuLista
```



Kuva 6. Herkkuluita listalta poimiva funktio.

```
herkkuluuLista = filter(=="herkkuluu") herkkuLista
```

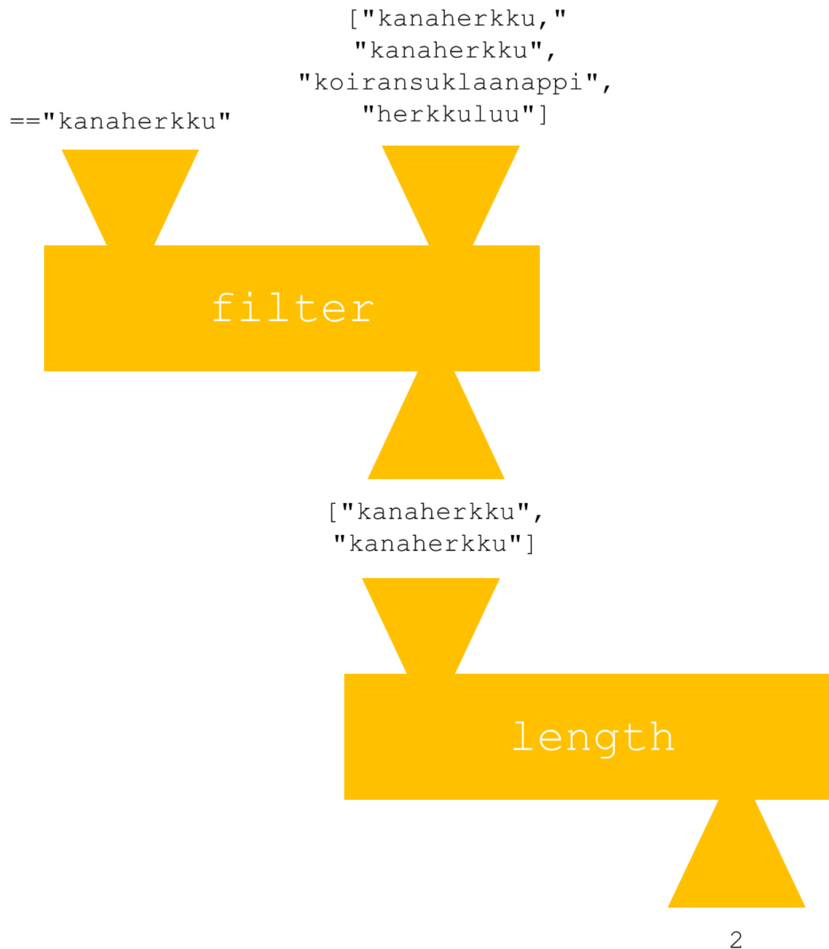
”Nyt kun Pate saimme eri herkut omille listoilleen, on helppo selvittää jokaisen listan alkioiden lukumäärät tuttuun tapaan `length`-funktiolla.”

```
> kanaherkkuLkm = length kanaherkkuLista

> herkkuluuLkm = length herkkuluuLista

> koiransuklaanappiLkm = length koiransuklaanappiLista
```

Kahden funktion yhdistäminen yhdeksi

”No eipä ollut vaikeata”, tokaisee Pate. ”Onko muuten aina tehtävä tuo homma kahdessa vaiheessa eli ensin määriteltävä tuo `kanaherkkuLista` ja vasta sitten `kanaherkkuLkm` vai voisimmeko päätyä kanaherkkujen ja muiden herkkujen lukumääriin ilman moisia välivaiheita?”

”Kyllä se vaan onnistuu!” vastaa Selma. ”Nokkelana koirana voit yhdistää herkkujen valintafunktion ja herkkujen lukumäärän laskufunktion ikään kuin yhdeksi funktioksi. Funktioon annostellaan yhtenä argumenttina iso herkkupussi eli lista, jossa voi olla kaikenlaisia herkkuja ja toisena argumenttina vertailufunktio, kuten `==”kanaherkku”`. Funktion tuotos on vertailufunktion toiseksi argumentiksi annetun herkun lukumäärä isossa pussissa. Tältä näyttäisi funktio, jolla voimme selvittää kuinka monta kanaherkkua herkkupussissa on.”



Kuva 7. Kanaherkkuja listalta poimiva funktio, joka on yhdistetty listan alkioiden lukumäärän laskevaan funktioon.

```
kanaherkkuLkm = length(filter(=="kanaherkku")herkkuLista)
```

”Ohjelma voidaan Pate ajatella myös matematiikasta tuttuna sievennyksenä. Ensin lasketaan: `filter(=="kanaherkku")herkkuLista`, joka tuottaa `["kanaherkku","kanaherkku"]`. Seuraavaksi tuotos annetaan argumentiksi `length`-funktiolle: `length(["kanaherkku","kanaherkku"])`. Sen jälkeen `length`-funktio vuorostaan tuottaa listan alkioiden lukumäärän eli luvun 2.

Toisin sanoen ensin poimitaan `herkkuLista`lta kaikki alkiot, jotka sisältävät merkkijonon ”kanaherkku” ja sitten lasketaan poimittujen alkioiden määrä.

Yksityiskohtaisemmin selitettynä se menee näin. Ensin luodaan siis uusi lista, jossa on vain ”kanaherkku” alkioita. Uusi lista saadaan aikaan soveltamalla filter-funktion arumenttia eli `=="kanaherkku"` -funktiota jokaiseen `herkkuLista`n alkioon. Ja kun ”kanaherkku” alkioita sisältävä lista on luotu, `length`-funktiota sovelletaan luodulle listalle, mikä tuottaa alkioiden lukumäärän. Ei hullumpaa vai mitä sanot Pate?”

”Ei ollenkaan”, vastaa Pate. ”Jos oikein ymmärsin, niin koko ohjelma on itse asiassa sieventämistä – mutta voiko se olla niin yksinkertaista?”

”Kyllä se Pate voi. Yksinkertainen on kaunista!” hihkuu Selma. ”Tässä on Pate vielä herkkuluiden ja koiransuklaanappien lukumäärät selvittävät ohjelmakoodit. Jos haluat harjoitella, voit piirtää niistäkin tuollaiset kuvat, niin opit asiat paremmin.”

```
herkkuluuLkm = length(filter(=="herkkuluu")herkkuLista)
```

```
koiransuklaanappiLkm = length(filter(=="koiransuklaanappi")
herkkuLista)
```

”Kuulehan Selma. Jotta `=="herkkuluu"` funktiota ei jatkuvasti tarvitsisi toistaa eri puolilla ohjelmaa, voisimmeko korvata sen antamalle tuolle vertailufunktiolle nimen?”

”Kyllä vaan Pate. Luonteva nimi vertailufunktiolle voisi olla `onkoHerkkuluu`. Nimi on hyvä siksi, että == - funktio antaa aina tuotokseksi tosi tai epätosi, joten voi olla luontevaa ikään kuin kysyä `onkoHerkkuluu` ja antaa funktion sitten tuottaa vastaukseksi tosi tai epätosi. Voidaan ajatella, että tosi merkitsee kyllä ja epätosi ei. Katsohan koodia. Voit laatia koodin joko näin

```
onkoHerkkuluu = (=="herkkuluu")
```

tai hahmonsovitusta käyttäen näin

```
onkoHerkkuluu "herkkuluu" = True
```

```
onkoHerkkuluu _ = False
```

Sitten vaan muutat ohjelmakoodiasi näin

```
herkkuluuLkm = length(filter(onkoHerkkuluu)herkkuLista)
```

Ja nyt voit käyttää `onkoHerkkuluu`-funktiota aina siellä, missä sinun tarvitsee tutkia, onko jokin merkkijono `"herkkuluu"` vai ei. Näin sinun Pate ei tarvitse aina erikseen kirjoittaa tuota `=="herkkuluu"` -funktiota joka puolelle, vaan voit korvata sen `onkoHerkkuluu` -nimellä. Jos kirjoitat moneen paikkaan `=="herkkuluu"`, niin joskus voikin tulla virhe ja kirjoitat vahingossa `=="hekkuluu"` ja tuletkin vahingossa etsineeksi herkkulistaltasi jotain, mitä siellä tuskin on. Kun käytät yhtä ja samaa funktiota, voit korjata virheellisen `"hekkuluu"` -sanan ja se tulee saman tien voimaan kaikissa niissä muiden funktioiden määritelmissä, jossa esiintyy `onkoHerkkuluu` -nimi.

”Ohjelma näyttää Selma kyllä toimivan, mutta onko tuo koodi nyt niin fiksua kuin se voisi olla”, kysyy Pate hieman epäilevästi. ”Eikö tuossa ole aika paljon samannäköisiä koodirivejä, joissa vaihtuu vain tuo herkun tai sitä vastaavan funktion (esim. `onkoHerkkuluu`) nimi. Nuo lukumääriä laskevat funktioiden nimet, kuten `herkkuluuLkm` ja `koiransuklaanappiLkm` eroavat toistaan vain `filter`-funktion argumentin eli annosteltavien argumenttien osalta. Loppujen lopuksi näytämme saavan aikaiseksi jakauman eri herkkujen määristä pussissa, mitä edustavat nuo `Lkm` -päätteiset funktiot.

Kun nuo herkkujen nimet on jo kuitenkin esitelty tuossa `herkkuLista`ssa, niin emmekö voisi käyttää nimiä suoraan tuosta listasta? Nythän nimiä käytetään vertailufunktiossa tassupelillä. Mehän tarvitsisimme nyt vain sellaisen listan, joka sisältäisi vain ne `herkkuLista`n alkiot, jotka ovat ainutlaatuisia, eli listan, jossa on vain **eri** herkkuja – listan, jossa ei ole kahta samaa alkiota. Näin saisimme tietää erilaisten herkkujen **joukon**. Ja mitä sitten tapahtuu, jos erilaisia herkkuja onkin vaikka 50, pitääkö kaikki ohjelmarivit kirjoittaa käsin. Minä osaan Selma helpostikin kuvitella 50 erilaista koiranherkkua. Lkm- ja Lista -loppuisia funktioiden nimiä tulee hurja määrä ja on hirveän helppo tehdä kirjoitusvirhe niitä kirjoitellessa ja meneehän siihen jo aikaakin paljon”, huokailee Pate.

”Oletpa Pate taas kerran oikeassa!” jatkaa Selma. ”On todellakin niin, että ohjelmasta voisi tehdä paljon fiksumman. Katsotaanpa yhdessä, miten se onnistuisi. Olet oikeassa myös siinä, että `herkkuLista`lla olevia nimiä kannattaa käyttää uudelleen. Ei ole mielekästä tassupelillä kirjoittaa uudelleen herkkujen nimiä, koska olemme ne jo kerran kirjoittaneet. Mutta osaisitko kertoa millainen olisi se ohjelmakoodi, jolla saisimme toisteisuutta vähemmälle ohjelmakoodissa?”

”Nyt pistit kyllä Selma pahan pähkinän minun koiranaivoilleni, mutta yritetäänpä. Jos siis saisimme aikaiseksi tuollaisen listan, jossa on herkkujen joukko, niin mehän voisimme käyttää sitä listaa valitsimena siinä tilanteessa, että meidän pitäisi laskea kuinka monta kertaa eri herkut esiintyvät tuossa `herkkuLista`-listalla. Otettaisiin ensin herkkujen joukon ensimmäinen alkio ja annosteltaisiin se argumenttina sellaiseen funktioon, joka osaisi käyttää joukon alkiota argumenttinaan ja laskea kuinka monta kertaa kyseinen herkku eli joukon alkio esiintyy `herkkuLista`-listalla. Eli me tarvitsemme siis kaksi listaa: herkkujen joukkoa kuvaavan listan ja alkuperäistä herkkupussin sisältöä kuvaavan listan. Ja siten vaan tuotetaan niistä argumenteista uusi lista, jonka jokainen alkio kuvaa tietyn herkun esiintymiskertoja alkuperäisellä `herkkuLista`-listalla.”

”Kuulostaa järkevältä”, vastaa Pate Selmalle. ”Miten ihmeessä saamme aikaiseksi erilaisia alkioita sisältävän joukon tuosta alkuperäisen herkkupussin sisältöä kuvaavasta `herkkuLista` -listasta? Taas jollain funktiolla?”

”Funktiollapa hyvinkin”, vastaa Selma. ”Ja sellainen funktio on jo keksitty. Sen nimi on `nub`-funktio. Annat sille argumenttina minkä tahansa listan, niin se tuottaa sinulle listan, jossa on vain erilaisia alkioita. Mitäs sellaisesta funktiosta sanot?”

”Wau ja hau!” vastaa Pate. ”Sitähän me sitten käytämme. Mutta kun sellainen lista saadaan aikaiseksi, niin miten me käytämme sitä `nub`-funktion tuottamaa listaa hyväksemme? Miten me saamme nuo alkuperäiset tassupelillä luetellut merkkijonot korvattua niillä `nub`-funktion tuottamilla arvoilla? Miten se homma saadaan oikein rullaamaan?”

”Kuulehan Pate!” jatkaa Selma. ”Homma saadaan toimimaan niin, että opettelemme uuden funktion. Sen funktion nimi on `map`. Se on funktioiden aatelia ja saa argumenttinaan toisen funktion ja listan asioita. Kun opit käyttämään `map`-funktiota, niin olet jo ohjelmoinnin opinnoissasi pitkällä. Se on yksi tärkeimpiä funktioita ja `map`-funktio kannattaa opetella hyvin. Koirakaverisi Topi saattaa kyllä mennä vihreäksi kateudesta, kun kerrot, että osaat käyttää `map`-funktiota. Sitten voit opettaa `map`-funktion käytön Topillekin, niin hän saa turkkiinsa tutun ruskean sävyn takaisin!

Luku 7. Map-funktio

”Kuulehan Pate! Nyt on aika tutustua `map`-funktioon. Se on hienostunut funktio, jota käytetään nykyään monissa ohjelmointikielissä.”

”Vai niin Selma”, vastaa Pate. ”Mitä erinomaista `map`-funktiossa sitten oikein on?”

”Oletko Pate kuullut kennelistä, jossa jokaisen koiran kanaherkkuannosta nostettiin kolmella kanaherkulla?”

”No enpä ole”, vastaa Pate uteliaana. ”Olisi tietysti mukava olla koirana sellaisessa kennelissä, missä kanaherkkuannosta voidaan nostaa vaivattomasti kaikille koirille kerralla. Se olisi kennel minun makuuni, kirjaimellisesti.”

”No varmasti olisi”, vastaa Selma. ”Ajatellaan, että meillä olisi viiden koiran kennel ja koirien nimet olisivat Selma, Topi, Repa, Motti ja Pate. Koirien tämänhetkiset kanaherkkuannokset olisivat:

Selma, 2 annosta

Topi, 3 annosta

Repa, 5 annosta

Motti, 1 annos

Pate, 3 annosta.

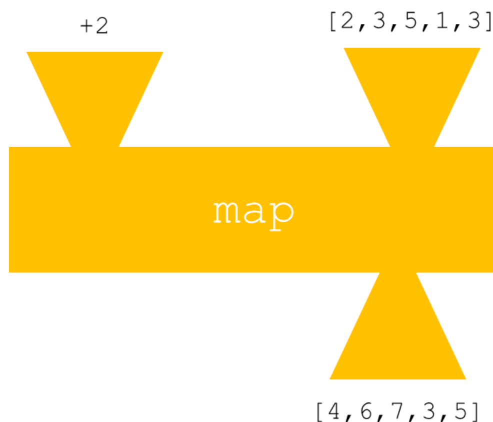
Annoksia voitaisiin kuvata myös listalla, joka sisältäisi jokaisen koiran annoksen alkaen Selmasta ja päättyen Pateen `[2,3,5,1,3]`.

Jos jokaisen koiran annoskokoa kasvatettaisiin kahdella kanaherkulla, niin uudet annokset olisivat `[4,5,7,3,5]`. Jokaiseen annokseen voitaisiin siis yksitellen lisätä luku 2. On kuitenkin melkoinen työ lisätä yksitellen jokaiseen lukuun tuo sama kakkonen. Mietipä Pate, jos kyseessä olisikin 200 koiran kennel. Siinä loppuisivat äkkiä tassut kesken kasvattaessa jokaisen koiran annoskokoa yksitellen.

Eikö olisi Pate mukavaa, jos voisimme käsitellä listalla olevien annosten kokoja ihan samalla tavalla kuin käsittelemme yhtä annosta. Jos voimme kasvattaa yhden koiran annoksen kokoa kahdella, niin miksemme kysyisi, että ehkä jollain kumman konstilla voisimme kasvattaa jokaisen koiran annoskokoa kahdella. Sellainen funktio tarvitsisi argumenteikseen listan eri koirien kanaherkkuannosmääristä ja tiedon siitä, miten kunkin koiran kanaherkkuannosta muutetaan."

"No todellakin se olisi viisasta", jatkaa Pate uteliaana. "Palan jo halusta nähdä, miten se `map`-funktio toimii. Näyttäisitkö miltä se näyttää?"

"Totta kai näytän", vastaa Selma ja nyökkää ystävällisesti. "Näin se toimii, katsohan kuvaa ja ohjelmakoodia."

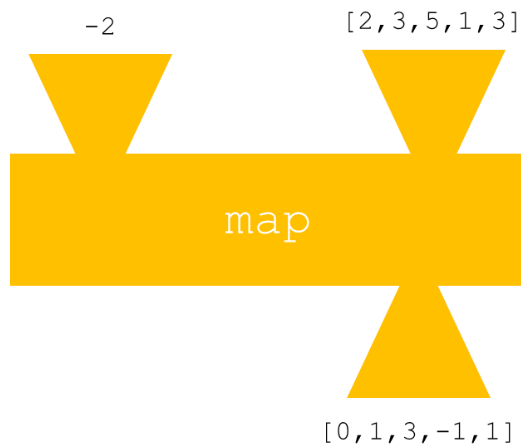


Kuva 8. Kasvatetaan kanaherkkuannosten kokoa kahdella map-funktion avulla.

```
> annosLista = [2,3,5,1,3]

> uudetAnnoksetLista = map (+2) annosLista

> uudetAnnoksetLista

=> [4,5,7,3,5]
```

”No tuo on todella edistyksellistä”, sanoo Pate. ”+2 näyttää olevan se pariaate, jolla kanaherkkuannosmäärää muutetaan. Jokaisen koiran annokseen lisätään kaksi. Samalla periaatteellahan voisimme nyt myös pienentää herkkuannoksia, jos kennelin koirat päättäisivät yhtenä laumana juosta vaikkapa jänisten perässä ilman lupaa.”

”Hyvin olet läksysi tehnyt, aivan hyvin voisimme pienentää kaikkien kennelin koirien kanaherkkuannoksia kahdella näin.



Kuva 9. Pienennetään kanaherkkuannosten kokoa kahdella map-funktion avulla.

```
> annosLista = [2,3,5,1,3]

> uudetAnnoksetLista = map (-2) annosLista

> uudetAnnoksetLista

=> [0,1,3,-1,1]
```

Huomasit myös Pate varmaan, kuinka listan neljännen koiran kanaherkkuannosta ilmaiseva luku on nyt negatiivinen -1. Tällainen tilanne ei tietenkään ole hyvä, sillä eihän kanaherkkuja voi onneksi olla nollaa vähempää. Emme kuitenkaan tässä vaiheessa puutu tähän ilmeiseen ongelmaan. On silti hyvä huomata että ohjelmassamme on parantamisen varaa.”

”Kiitos taas hyvistä opeista Selma”, vastaa Pate. ”Koska nälkä kasvaa koiran syödessä, niin haluaisin laatia sellainen funktion, jonka avulla hyvästä käytöksestä saisi enemmän kanaherkkuja ja huonosta käytöksestä vähemmän kanaherkkuja. Funktio voisi toimia vaikka niin, että jos viikossa olisi juossut useammin kuin kaksi kertaa jäniksen perään ilman lupaa, niin kanaherkkuannos pienenisi yhdellä. Olisiko sellaisen funktion rakentaminen näillä tiedoilla mahdollista ja millainen funktio siitä Selma oikein tulisi?”

”No sinähän olet oikein innoissasi näistä funktioista enkä yhtään ihmettele, sillä näillä funktioillahan voi nokkela koira tehdä mitä vaan. Ensin tarvittaisiin tietysti tieto siitä, kuinka monta kertaa kukin koira on jäniksen perään viikon aikana juossut. Listataanpa karkailukerrat koirittain:

Selma, 4 kertaa

Topi, 2 kertaa

Repa, 2 kertaa

Motti, 9 kertaa

Pate, 2 kertaa.

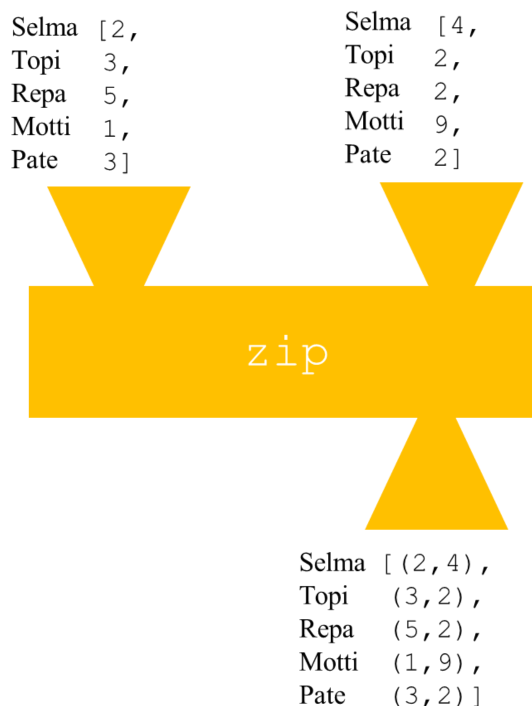
Koko koiraporukan karkailukertojen lukumääriä voitaisiin kuvata listalla `[4,2,2,9,2]`.

Nyt meillä on jäljellä yksinkertainen tehtävä. Kanaherkkuannosta pitää pienentää, jos koira on juossut jäniksen perään useammin kuin kaksi kertaa viikon aikana. Vaadittava ohjelma on nyt hieman mutkikkaampi kuin edellisessä esimerkissä, mutta kaikkea muuta kuin mahdoton tehtävä innokkaalle vehnäterrierille.

Ongelma ratkeaa esimerkiksi niin, että luomme listan pareja joissa parin ensimmäisenä alkiona (`fst` eli first) on koiran nykyinen viikkoannoskoko ja toisena alkiona (`snd` eli second) on koiran karkailujen lukumäärä. Koska jokaiselle koiralle tarvitsemme yhden tällaisen parin, tarvitsemme siis listan, jossa on 5 paria.

Parilistan voi luoda myös käsin, mutta `zip`-funktiota käyttäen saamme listan luotua helpommin. `Zip`-funktiolle annamme ensimmäiseksi argumentiksi kanaherkkuannoskokolistan ja toiseksi argumentiksi karkailumäärälistan. Lopputuloksena saamme listan, jossa on pareja, joiden ensimmäisenä alkiona on kanaherkkuannoskoko ja toisena alkiona karkailumäärä.

Tiesitkö Pate, että zip on englantia ja tarkoittaa paitsi hienoa funktiota, jonka nimi on zip, myös vetoketjun kiinni vetämistä. Ja siltähän tuo seuraava kuvakin näyttää - otetaan oikealta ja vasemmalta puolelta luvut ja yhdistetään ne yhdeksi listaksi – näin vetoketju sulkeutuu. Näin kahdesta vetoketjun puolikkaasta tulee yksi, jossa listojen alkiot paritetaan kuin vetoketjun puolet konsanaan, kun ne liittyvät toisiinsa vetoketjua suljettaessa.



Kuva 10. Yhdistetään kaksi listaa parilistaksi zip-funktion avulla.

```
> zip [2,3,5,1,3] [4,2,2,9,2]
```

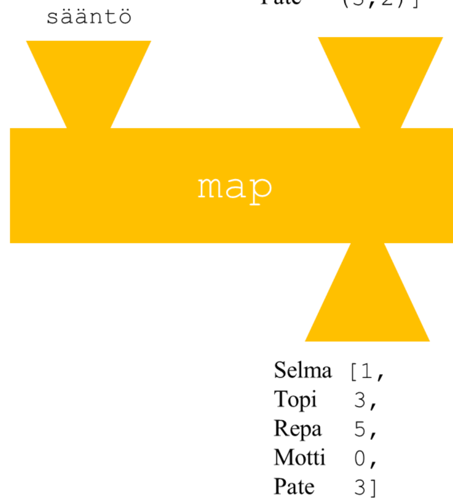
```
=> [(2,4),(3,2),(5,2),(1,9),(3,2)]
```

Mutta millainen on funktio, jonka avulla saadaan luotua uusi kanaherkkuannoslista, jossa tiettyjen koirien kanaherkkuannosta on pienennetty karkailumäärän perusteella?

Kuten muistamme `map`-funktio käy läpi listan ja tuottaa uuden listan. `Map`-funktiomme voisi nyt ottaa käsittelyyn listan ensimmäisen alkion `(2,4)`. Jos listan alkiota (tässä tapauksessa pari) kuvataan kirjaimella `x`, niin parin jälkimmäistä alkiota voidaan kuvata ilmaisulla `snd x`. `Snd` on siis funktio, jonka avulla saadaan parin toisen alkion arvo. `Fst` on funktio, jonka avulla saadaan parin ensimmäisen alkion arvo. Ensin siis tutkimme, onko parin jälkimmäinen alkio suurempi kuin 2. Kun yhdistämme tähän tietomme siitä, mitä vahtien avulla voi tehdä, saamme aikaan seuraavanlaisen funktion:

```
sääntö x          -- x on yksi pari!
  |snd x>2 = fst x-1
  |otherwise = fst x
```

`sääntö`-funktio saa argumenttina parin. Jos parin toinen alkio(karkailumäärä) on suurempi kuin 2, funktion tuotos on parin ensimmäisen alkion(herkkuannoksen) arvo vähennettynä yhdellä eli `fst x-1`. Jos karkailumäärä on kaksi tai pienempi (`otherwise`) funktion tuotoksena on `fst x`, mikä on koiran nykyinen herkkuannos. Katsotaanpa Pate millaisen kuvan ja ohjelmakoodin tästä kaikesta saamme aikaiseksi.”



Kuva 11. Funktioon toisena argumenttina annosteltava sääntö kertoo miten parilistan alkioista muodostetaan yksittäisistä luvuista muodostuva lista.

```
annokset=[2,3,5,1,3]
karkailut=[4,2,2,9,2]
annoksetkarkailut = zip annokset karkailut
sääntö x
 |snd x>2 = fst x-1
 |otherwise = fst x

uudetAnnokset = map sääntö annoksetkarkailut
```

"Nyt kun Pate käynnistämme `uudetAnnokset`-funktion saamme tietää millaiset uudet kanaherkkuannokset kullakin koiralla on. Sovellamme siis `sääntöä` `annoksetkarkailut`-listan alkioihin.

```
> uudetAnnokset

=> [1,3,5,0,3]
```

”Jos nyt palautamme mieleen koirien järjestyksen listalla saamme seuraavan taulukon.

Selma, 1 annosta

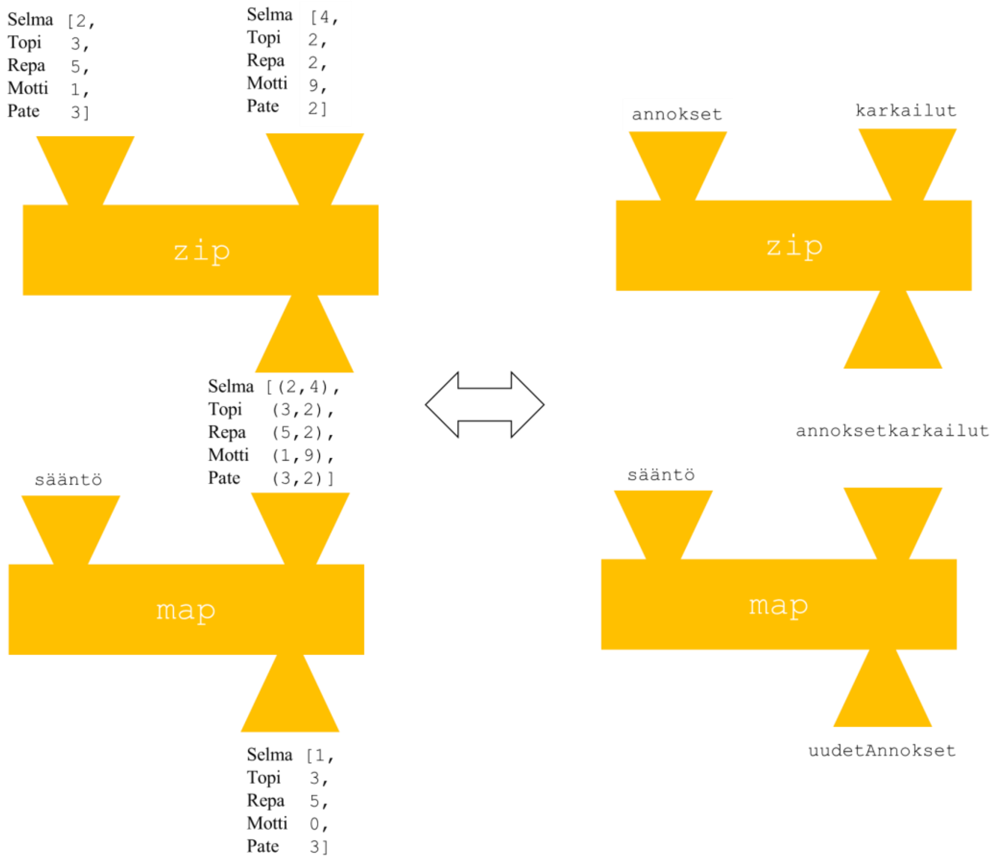
Topi, 3 annosta

Repa, 5 annosta

Motti, 0 annosta

Pate, 3 annosta.

Näyttäisi siis siltä, että Selman ja Motin annoskokoja on pienennetty, koska he ovat hölmöilleet. Tarkastelkaamme Pate vielä tekemäämme kokonaisuutta kuvan avulla. Lisäsin kuvaan selvyyden vuoksi koirien nimet, jotta näemme argumenttina funktioon syötetyn listan ja funktion tuottaman kanaherkkuannoslistan selkeämmin. Koska on niin, että `zip`-funktion tuotos annostellaan `map`-funktioon argumenttina, niin voimme hyvin esittää koko ohjelman yhdellä kuvalla.



Kuva 12. Antamalla listoille nimet niitä on helppo käyttää uudelleen eri puolilla ohjelmaa.

Vasemmalla puolella kuvaa näet `zip`-funktion jonka tuotos annostellaan `map`-funktioon. Oikealla puolella oleva kuva tarkoittaa samaa kuin vasemmalla puolella oleva kuva, mutta argumentit ja tuotokset on kuvattu **nimillä** arvojen sijaan.

`Map`-funktio toimii niin, että `Sääntö`-funktiota sovelletaan jokaiselle `map`-funktioon annostellulle parille. `Sääntö`-funktion viisaus on siinä, että se osaa argumenttina saamansa parin perusteella päätellä, mitä tietyn koiran kanaherkkuannos muuttuu, jos muuttuu ensinkään. Siinä, missä `filter`-funktio luo listan, joka on tyhjä tai siinä on enintään yhtä monta alkiota kuin `filter`-funktion argumenttina käytetyssä listassa, `map`-funktio luo listan, jossa on aina yhtä monta alkioita kuin `map`-funktion argumenttina käytetyssä listassa.”

”Onpa melkoinen funktio tuo `map`-funktio”, sanoo Pate. ”Mutta emme kuitenkaan ole vielä ratkaisseet edellisessä luvussa esittelemäämme ongelmaa. Siinähän meillä oli tarkoituksena luoda `map`-funktion avulla funktio, jonka avulla voisimme saada jakauman herkkupussin sisällöstä niin, että eri herkkujen lukumääriä laskevia funktioita ei tarvitsisi määritellä erikseen. Se olisi minun mielestäni todellista automaattista tietojenkäsittelyä sanan varsinaisessa merkityksessä. Kerrohan Selma, miten sellainen funktio tehtäisiin?”

”Kerronhan toki. Se on todella yksinkertaista. Sovellamme ensin `nub`-funktiota `herkkuLista`an, jolloin saamme listan, jossa jokainen herkku esiintyy vain kerran. Sen jälkeen sovellamme `map`-funktiota `herkkuJoukko`-listaan. `Map`-funktio saa listan lisäksi argumentiksi funktion, joka käy läpi `herkkuJoukko`-listan alkioita. Jokaisen `herkkuJoukko`-listan alkion osalta selvitetään, kuinka monta kertaa kyseinen alkio esiintyy `herkkuLista`-listalla.

Meidän on nyt vaan keksittävä, miten tuo `map`-funktiolle annosteltava funktio on laadittava, jotta ongelma ratkeaa. Funktion idea on, että se saa argumenttina herkun nimen ja selvittää nimen perusteella kuinka monta kertaa nimi esiintyy `herkkuListalla`. Funktio tuottaa nimen esiintymislukumäärän `herkkuListalla`. Funktio voitaisiin nimetä vaikkapa `laskuri`-funktioksi ja se voitaisiin toteuttaa näin. Herkun nimeä kuvaa `x`:

```
laskuri x=length(filter(==x)herkkuLista))
```

Nyt kun `laskuri`-funktio on laadittu, voimme kirjoittaa lopun ohjelmakoodin, joka käyttää `laskuri`-funktiota:

```
herkkuLista = ["kanaherkku","kanaherkku","koiransuklaanappi",
"herkkuluu"]

herkkuJoukko=nub herkkuLista

herkkujenEsiintymiskerratPussissa=map laskuri herkkuJoukko

> herkkujenEsiintymiskerratPussissa
```

```
=> [2,1,1]
```

Katsotaanpa Pate vielä vaiheittain, miten juttu etenee. HerkkuLista on `["kanaherkku","kanaherkku","koiransuklaanappi","herkkuluu"]`. Herkkujen joukko luodaan siis `nub`-funktiolla annostelemalla siihen kaikki herkut ja tuotoksena on siis `["kanaherkku","koiransuklaanappi","herkkuluu"]`.

Lopuksi annostelemme `herkkuJoukko`-listan `map`-funktioon ja annamme `laskuri`-funktion selvittää kuinka monta kertaa tietty herkku `herkkuLista`lla esiintyy. Näin saamme tietää eri herkkujen määrän pussissa niin, että ohjelmakoodimme pysyy lyhyenä ja napakkana. Jos herkkupussiin eli `herkkuLista`-listalle lisätään uusia herkkuja, ohjelmakoodissa esiintyvät funktiot osaavat ottaa uudet herkut huomioon, sillä `nub`-funktio tuottaa listan, joka sisältää kaikki `herkkuLista`n erilaiset alkiot.

On muuten Pate niin, että `map` ja `filter`-funktiot ovat kenties yleisimmin käytettyjä funktionaalisen ohjelmoinnin funktioita ja niitä käytetään monissa muissakin ohjelmointikielissä kuin Haskellissa.

`Map` ja `filter`-funktioiden toiminta voidaan kuitenkin tarvittaessa korvata `fold`-funktiolla. Funktio tunnetaan yleisesti myös nimellä reduce. `Fold`-funktio on kirjan viimeinen esimerkkifunktio. `Fold`-funktion käyttämisessä on kuitenkin sen verran pohdittavaa, että käymme asian läpi seuraavassa luvussa. Nyt on aika unohtaa funktiot hetkeksi ja syödä pari kanaherkkua palkkioksi ahkerasta opiskelusta.”

”Sopii minulle”, sanoo Pate ja syö jo ensimmäistä kanaherkkuaan.

Luku 8. Fold-funktio

”Tiesitkö Pate, että nykyaikainen tietojenkäsittely on muutakin kuin kanaherkkuannosten koon muuttamista koirien käyttäytymisen perusteella. Usein tiedosta halutaan saada aikaan myös erilaisia yhteenvetoja. Yksi esimerkki tällaisista yhteenvedoista on jakauma, joka saadaan eri herkkujen esiintymistiheyksistä `herkkuLista`lla.

Herkkulista sisältää kaiken tiedon, mikä jakauman esittämiseen tarvitaan. Meidän työkalupakistamme kuitenkin puuttuu funktio, jonka avulla on helppo luoda kaikenlaisia yhteenvetoja. Sen funktion nimi on `fold`. Tutkimme miten `map`- ja `filter`-funktion yhteistoiminta voidaan korvata kätevästi yhdellä funktiolla, `fold`illa.”

”Vai että `fold`-funktio. Mahtaa olla melkoinen funktio, jos sillä niin helposti yhteenvetojakin saa aikaiseksi ja varsinkin jos sillä voi korvata kaksi funktiota”, jatkaa Pate.

”Niin. Siinäpä meillä onkin miettimistä”, toteaa Selma. ”Ajatellaanpa, että eurasier ystäväsi Topi on juuri saanut seuraavanlaisen todistuksen koirakoulun ensimmäiseltä luokalta. Arvioidut taidot jaetaan todistuksessa kolmeen ryhmään: metsästys, pihatyöt ja muut taidot.

Metsästys

Pupun jäljestys	9
Hirven jäljestys	8
Linnun noutaminen	8

Pihatyöt

Lumen pöllyytys	10
Kukkapenkkien kaivaminen	10

Muut

Parvekkeen vahtiminen	9
Piilotetun luun löytäminen	7
Oman hännän jahtaaminen	9
Kuun ulvonta	8

Jotta taitojen yhteispistemääriä voitaisiin verrata ryhmittäin koirien välillä, ne pitää tietenkin laskea ensin yhteen. Mutta mitä tietoa meillä pitäisi olla, jotta tämä olisi mahdollista? Mitä Pate tuumaat?” jatkaa Selma.

”No ainakin meidän pitäisi tietää, mitä taitoja eri ryhmiin kuuluu, eli meillä pitäisi olla jonkinlainen ryhmittely. Tässä voisimme varmaan käyttää pareja. Parin ensimmäinen alkio voisi tarkoittaa taitoa ja toinen alkio ryhmää, johon alkio kuuluu. Parityyppisestä tietorakenteesta käytetään myös nimitystä monikko (tuple), koska se sisältää aina 2 tai useampia arvoja. Voisiko ryhmittelyn tehdä näin?” kysyy Pate ja näyttää ohjelmakoodia.

```
ryhmittely=[("Pupun jäljestys","Metsästys"),
          ("Hirven jäljestys","Metsästys"),
          ("Linnun noutaminen","Metsästys"),
          ("Lumen pöllyytys","Pihatyöt"),
          ("Kukkapenkkien kaivaminen","Pihatyöt"),
          ("Parvekkeen vahtiminen", "Muut"),
          ("Piilotetun luun löytäminen","Muut"),
          ("Oman hännän jahtaaminen","Muut"),
          ("Kuun ulvonta","Muut")]
```

”Kyllä vaan voi”, vastaa Selma. ”Siinähän ne ovat! Nyt meidän vielä pitäisi esittää koirakoulun todistus niin, että siinä ovat vain arvosanat eri taidoista. Osaisitko Pate kirjoittaa sellaisen ohjelman?”

”No aina voi yrittää, ja näin minä sen todistuksen muotoilin ohjelmaksi”, vastaa Pate.

```
topinTodistus'=   [("Pupun jäljestys",9),
                    ("Hirven jäljestys",8),
                    ("Linnun noutaminen",8),
                    ("Lumen pöllyytys",10),
                    ("Kukkapenkkien kaivaminen",10),
                    ("Parvekkeen vahtiminen",9),
                    ("Piilotetun luun löytäminen",7),
                    ("Oman hännän jahtaaminen",9),
                    ("Kuun ulvonta",8)]
```

”Tuohan näyttää tosi hyvältä”, sanoo Selma. ”Nyt meidän pitää vain luoda ohjelma, jonka avulla saamme laskettua arvosanojen summat ryhmittäin. Siihen voimme avuksi luoda pienen oman tietotyyppimme. Me emme tässä yhteydessä käsittele tietotyyppejä sen seikkaperäisemmin, mutta kerrottakoon, että niiden avulla voimme tehdä kaikenlaisia hauskoja temppuja. Myös sellaisia temppuja, joita ei koirien agility radoilla ole kuunaan tullut vastaan. Lisätäänpä Pate soppaan tällainen ohjelmarivi.

```
data RyhmienSummat = RyhmienSummat {metsästys::Int, piha-
työt::Int, muut::Int}
```

Kaikki oleellinen on nyt määritelty Pate, nyt meidän tarvitsee vain miettiä, millaisia funktioita tarvitsisimme, jotta saisimme arvosanat ryhmiteltyä ja laskettua eri ryhmiin kuuluvien arvosanojen summat. Mutta koska olet Pate ensimmäistä kertaa tällaista ongelmaa ratkaisemassa, niin kerron sinulla, miten ongelmaa voisi lähestyä", jatkaa Selma.

"No jos vielä tämän tehtävän jaksan kuunnella ja pohtia, niin lupaatko Selma, että syödään sen jälkeen kunnon kanafileet?" kysyy Pate.

"Kyllä vaan syödään", vastaa Selma topakasti. "Näin minä sen ongelman ratkaisisin", sanoo Selma.

```
summatAiheittain' = foldr (\x acc->case (ryhmittele (fst x)
of

"Metsästys" -> acc {metsästys=(metsästys acc)+snd x}

"Pihatyöt" -> acc {pihatyöt=(pihatyöt acc)+snd x}

"Muut" -> acc {muut=(muut acc)+snd x})

RyhmienSummat {metsästys=0, pihatyöt=0, muut=0} topinTodis-
tus'
```

"Tässä ohjelmakoodissa on nyt joitain uusia asioita, kuten omia tietotyyppejä ja hiukan muutakin, mutta me koirat emme pikkuseikkoihin takerru, vaan selvitämme isot linjat. `SummatAiheittain'`-funktion idea on, että se käyttää `foldr`-funktiota. `Foldr`-on funktio, jolle annostellaan kolme asiaa.

Ensinnäkin annostelemme funktiolle Topin todistuksen ilman ryhmittelyä, se on viimeinen annosteltavista asioista ja sitä kuvaa nimi `topinTodistus'`. Sitä ennen annostelemme `foldr`-funktioon oman tietotyyppimme arvon `{metsästys=0, pihatyöt=0, muut=0}`. Tuossa rimpsussa annamme jokaiselle ryhmälle alkuarvoksi nollan. `RyhmienSummat` nimi kertoo, millaisen tietotyypin arvon luomme – ja meidän oman tietotyyppimme nimi on `RyhmienSummat`.

Mutta mitä ihmettä tuo funktio sitten oikein tekee, kun annosteltuja arvoja aletaan käsitellä? Funktio alkaa käydä läpi topinTodistus'-listaa ja ottaa käsittelyyn ensimmäisen alkion. Funktio tutkii ensimmäisen parin taitoa koskevaa alkiota. Muistathan, että arvosanat määriteltiin listalla olevina (taito, arvosana) pareina. Jos taito on metsästys, lisäämme oman tietotyyppimme RyhmienSummat metsästys-tietokenttään todistuksessa olevan arvon. TopinTodistus'-listan alkion arvosanan saamme ottamalla kulloinkin käsittelyssä olevan parin x toisen alkion snd-funktiolla, johon viittaa ilmaisu snd x.

```
acc {metsästys=(metsästys acc)+snd x}
```

Tämä onnistuu, sillä acc on nimi arvolle, joka on RyhmienSummat-tyyppiä ja sisältää kolme kokonaislukutyyppistä tietokenttää. Nämä kolme kokonaislukutyyppistä tietokenttää metsästys, pihatyöt ja, muut kuvaavat niitä ryhmiä eli arvosanojen summia, joita kerrytetään Topin todistuksesta saatavilla arvosanoilla fold–funktion evaluoinnin aikana.

Acc tarkoittaa siis funktion toiminnan aikana kerrytettävää tietoa (sisältää oman tietotyyppimme tyyppisen arvon – tietotyypin, jonka kenttien arvot alustimme nolliksi). Tästä kerrytettävästä tiedosta luodaan uusi arvo joka kerta, kun topinTodistus'-listalta löytyy ryhmittelemätön arvosana. Uusi arvo luodaan lisäämällä tietyn ryhmän pistemäärään todistuksesta löytyvä pistemäärä. Funktio käy läpi topinTodistus'-listan ensimmäisestä alkiosta viimeiseen ja lopulta funktio tuottaa RyhmienSummat-tyyppisen arvon, joka sisältää arvosanojen summat ryhmittäin.

Mutta jotta funktio toimisi oikein, sen tulee tavalla tai toisella löytää topinTodistus'-listan monikkoalkion taitoa vastaavaa ryhmä, jotta ryhmittely olisi mahdollista. Tätä varten loin pienen apufunktion ryhmittele, jolle annostellaan argumenttina taito, ja funktio tuottaa sitä vastaavana ryhmän. Kuten muistat, näin ryhmittele-funktiota käytetään summatAiheittain'-funktiossamme.

```
(\x acc->case (ryhmittele (fst x)…
```

Koska `x` on pari ja kuvaa käsittelyssä olevaa `topinTodistus'`-listan alkiota, taito saadaan ottamalla `x`:stä ensimmäinen alkio käyttämällä `fst`-funktiota. Sen jälkeen taito annostellaan `ryhmittele`-funktioon, joka näyttää tältä:

```
ryhmittele taito = snd(head(filter(\z-> (fst z)==taito) ryh-
mitttely))
```

Se on siis yksinkertainen funktio, jolle annostellaan taito ja funktio tuottaa ryhmän. Funktioon annostellaan ryhmittely, joka sisältää tiedon, mihin ryhmään mikin taito kuuluu. Lisäksi funktioon annostellaan funktio, jonka avulla poimitaan juuri tiettyä taitoa koskeva ryhmä `(filter(\z-> (fst z)==taito)`. Koska `filter`-funktio tuottaa aina listan, on meidän otettava tuotetusta listasta ensimmäinen alkio `head`-funktiolla. Tämän jälkeen nappaamme vielä parin toisen alkion `snd`-funktiolla ja näin meillä on siis tuotoksena ryhmä, mihin funktioon annosteltu taito kuuluu. Kun kirjoitamme komentoriville

```
> summatAiheittain'
```

saamme vastaukseksi

```
=> RyhmienSummat {metsästys = 25, pihatyöt = 20, muut = 33}
```

Näimme Pate useassa kohtaa merkintätavan `(\z->` , jossa z:n paikalla saattoi olla muutakin. Kysymys on siitä, että määrittelemme funktion ilman nimeä, siinä paikassa, missä funktiota tarvitaan. Ohjelmoinnin käsittein funktio määriteltiin lambda-lausekkeena.

Tämä viimeinen `fold`-funktiota koskeva luku oli ilman muuta luvuista haastavin ja sinähän Pate voit palata aiheeseen ja vaikkapa käyttää hakukoneita ja alkaa etsiä itse tietoa asioista, nyt kun tunnistat peruskäsitteitä.

Lambdojen ja monien muiden funktionaalisen ohjelmointikielen käsitteiden ymmärtämisestä ja soveltamisesta on paljon iloa paitsi Haskell-kielellä ohjelmoidessa myös monilla muilla ohjelmointikielillä työskennellessä. Funktionaalisia piirteitä sisältävät monet nykyaikaiset ohjelmointikielet, kuten JavaScript (ECMAScript), TypeScript, Python, Java, Kotlin, Scala, Ruby, PHP, ja C++ ja monet muut.

Lienee Pate sanomattakin selvää, että funktionaalisen ohjelmoinnin perusjuttujen opetteleminen tukee hyvin myös muiden funktionaalisten kielten opiskelua. Sellaisia kieliä ovat esimerkiksi Elm, Racket, Clojure, ML ja Idris.

Nyt on Pate kanaherkkujen aika. Eiköhän mennä pyytämään emännältä yhdet sellaiset oikein isot ja yliherkulliset kanaherkut, koska emmehän me koirat pelkästä ohjelmoinnista elä, emmehän Pate?”

”No emme totisesti elä”, vastaa Pate. ”Ei kun kanaherkuille, mars”, säestää Pate.

Niin painui koirakaverusten päivä illaksi ja molemmat nukahtivat kanaherkun makuisiin uniinsa pohtien voisiko jonkin ohjelmointikielen kanaherkkuja sisältävä lista olla äärettömän pitkä.

Lähteet

Haskell-kieltä käsittelevä verkkosivusto

https://www.haskell.org/

Lukusuosituksia

Haskell-kieleen tutustuvalle

Lipovača, Miran 2011: Learn You a Haskell for Great Good!, https://www.learnyouahaskell.com

Haskell-kielen teoreettisesta viitekehyksestä kiinnostuneille

Milewski, Bartosz 2014: Category theory for programmers, https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface

”Lista sisään, lista ulos, siitä syntyy lopputulos”, riimittelee Pate.

2010-luvun lopulla ohjelmoinnin alkeisymmärrys katsottiin Suomessa niin tärkeäksi, että aihe sulautettiin peruskoulun opetussuunnitelmiin. Ohjelmointia voidaan harrastaa monella tavalla ja toisille ohjelmointi on nykyinen tai tuleva ammatti. Funktionaalinen ohjelmointityyli on eräs hauskimmista, varsinkin jos ratkaistavana on tiedon käsittelyyn liittyviä ongelmia.

Kirjassa seikkailevat koirakaverukset Pate ja Selma pohtivat yhdessä tuumin funktionaalisen ohjelmoinnin peruskäsitteitä niin kuin vain vehnäterrieri ja bichon frisén ja villakoiran risteytys osaavat.

Kirja sopii jokaiselle uteliaalle, joka joskus on pohtinut mitä ohjelmointi on. Kirja voi tarjota mahtavia ahaa-elämyksiä avoimelle, sitkeälle ja rohkeasti funktioita kokeilevalle mielelle. Lukija voi hämmästyttää itsensä ja huomata kuinka yksinkertaista ohjelmointi onkaan.