2.2.3 Paikalliset määrittelyt

- Muitten ohjelmointikielten tapaan myös Haskell sallii kirjoittaa sellaisia määrittelyjä, jotka näkyvät vain tässä lausekkeessa, eivät missään muualla.

- Sitä varten on lauseke

  `let` *määrittely*$_1$
  ⋮
  määrittely$_k$
  `in` e

- Jokainen paikallinen *määrittely*$_i$ on (mahdollisesti tyypitetty)

  funktion määrittely tai

  nimentä muotoa

  hahmo = *lauseke*.

  – Silloin tätä hahmoa sovitetaan tämän lausekkeen arvoon.

  – Nimennän vasen puoli on hahmo (eikä pelkkä muuttujan nimi) jotta saadaan nimettyä arvon kentät.

- Esimerkiksi `lastBit`-esimerkkimme (☞ 2.1.4) on yhtenä funktiona:

```
1   lastBit :: Int -> (Bool,Int)
2   lastBit n =
3       let (q,r) =
4               quotRem n 2
5       in (r==1,q)
```


- Saman `let`-lausekkeen määrittelyt *määrittely*$_i$ ja *määrittely*$_j$ voivat viitata

  rekursiivisesti itseensä

  vuorottain rekursiivisesti (mutually recursive) toisiinsa

  ympäröivien `let`**-lausekkeiden** määrittelyihin

  ympäröivien hahmojen muuttujiin

  aina päätasolle saakka.

- Muista ohjelmointikielistä tutun *leksikaalisen näkyvyyden* (lexical scoping) mukaisesti käytetään *lähintä* näin löydettyä määrittelyä.

```
1   f x =
2       let g 0 =
3               0
4           g y =
5               let x =
6                       h y
7                   in x + g (y-1)
8           h y =
9               g (div y 2)
10      in g (max x 0)
```

Rivillä 6 `h` viittaa riville 8 ja `y` riville 4.

Rivillä 7 `x` viittaa riville 5, `y` riville 4 ja `g` tähän funktioon itseensä.

Rivillä 9 `g` viittaa riveille 2–7 ja `y` riville 8.

- Rekursion ja laiskuuden vuoksi `let`-lausekkeiden tarkkaa semantiikkaa ei voi aivan suoraan kuvailla hahmonsovituksen korvausidealla (☞ 2.2.2).

  Tarkkaan semantiikkaan palataan myöhemmin (☞ 7.2).

- Intuitiivisesti

  `let` x `=` d
  `in` e

  ei vielä suorita yhtään mitään

  vaan määrittelee "jos `in`-lauseketta e joskus sievennetään, niin siinä nimi x tarkoittaa lauseketta d".

- Silloin esimerkiksi seuraava funktio toimii oikein myös parametrilla $z = 0.0$:

```
1   inv z =
2       let y = 1.0 / z
3       in if z == 0.0
4          then 0.0
5          else y
```


2.2.4 Asemointisääntö

- Itse asiassa paikallisen `let`-määrittelyn (☞ 2.2.3) täydellinen syntaksi on

  `let {` *määrittely*$_1$ `;`
  ⋮
  määrittely$_k$ `}`
  `in` e

  jossa yhtä aikaa tehtävät määrittelyt on erotettu

  puolipistein toisistaan

  aaltosuluin ympäröivästä lähdekoodista.

- Haskell-kieleen on kuitenkin määritelty erillinen *lähdekoodin asemointisääntö* (layout rule) jolla nämä erottavat puolipisteet ja aaltosulut voi korvata luontevin *sisennyksin:*
  - Periaatteena on, että sisempi näkyvyysalue sisennetään ulomman oikealle puolelle.
  - Lyhentää lähdekoodia — vaan selkeyttääkö sitä?
  - Pelkkä välilyöntien lisäys tai poisto voi muuttaa lähdekoodin merkitystä!
  - Käytä siis Haskell-syntaksin hallitsevaa editoria ja anna sen sisentää puolestasi!
  - Sellainen on esimerkiksi laitoksen Linux-koneiden editorin `xemacs` moodi `Haskell-mode`.

- Näillä kalvoilla käytetään asemointia erottimien sijaan.
- Sisennä `let`-lausekkeen määrittelyt d_i alkamaan keskenään samasta kohdasta, joka on varatun sanan `let` oikealla puolella:

```
let määrittely_1
    ⋮
    määrittely_k
in e
```

  Tämä sisennys korostaa, että ne ovat keskenään samantasoisia mutta varatulle sanalle `let` alisteisia.

- Määrittelysäännön sisennys:

```
f hahmo
    | vahti_1 =
      tulos_1
    ⋮
    | vahti_m =
      tulos_m
```

  - Jokainen haara alkakoon keskenään samasta kohdasta (jolla siis on vahdin aloittava pystyviiva), joka on nimen f oikealla puolella (koska nimeen f liittyvän hahmon muuttujat näkyvät vahteihin)
  - Jokainen tulos olkoon sisennetty oman pystyviivansa oikealle puolelle (koska sen vahti määrää valitaanko tämä tulos).

2.2.5 Sääntökohtaiset määrittelyt

- Usein Haskell-ohjelman sääntöön

```
f hahmo
    | vahti_1 =
      tulos_1
    ⋮
    | vahti_m =
      tulos_m
```

  haluttaisiin tehdä paikallisia `let`-määrittelyjä (☞ 2.2.3) siten, että ne näkyisivät tämän säännön kaikkiin haaroihin mutta eivät minnekään muualle.

- Haskell-syntaksissa on tähän tilanteeseen rakenne

$$
\begin{array}{l}
f\ \mathit{hahmo} \\
\quad \mid\ \mathit{vahti}_1\ \texttt{=} \\
\quad\quad \mathit{tulos}_1 \\
\quad \vdots \\
\quad \mid\ \mathit{vahti}_m\ \texttt{=} \\
\quad\quad \mathit{tulos}_m \\
\quad \texttt{where}\ \mathit{määrittely}_1 \\
\quad\quad\quad \vdots \\
\quad\quad\quad \mathit{määrittely}_k
\end{array}
$$

- Myös `where`-määrittelyillä on asemointisääntö (☞ 2.2.4):
  - varattu sana `where` samaan kohtaan kuin haara(t)
  - paikalliset määrittelyt kuten `let`-lausekkeessa.

2.2.6 Monen parametrin funktiot

- Haskellissa on myös > 1 parametrin funktioita.
- Sellaisen funktion f tyypitys on

$$f\ \texttt{::}\ \mathit{tyyppi}_1\ \texttt{->}\ \ldots\ \texttt{->}\ \mathit{tyyppi}_p\ \texttt{->}\ \mathit{tulostyyppi}$$

jossa jokainen tyyppi_i on vastaavan i. parametrin tyyppi.

- Sellaisen funktion jokainen sääntö alkaa muodossa

$$f\ \mathit{hahmo}_1\ \ldots\ \mathit{hahmo}_p$$

jossa jokainen hahmo_i on vastaavaa tyyppiä tyyppi_i.

- Sellaisen funktion kutsu on muotoa

$$f\ \mathit{lauseke}_1\ \ldots\ \mathit{lauseke}_p$$

jossa jokainen $\mathit{lauseke}_i$ on vastaavaa tyyppiä tyyppi_i.

- Kutsun sievennyksen idea on sama kuin aikaisemminkin (☞ 2.2.2):
  - Jokaista argumenttia $\mathit{lauseke}_i$ sievennetään sen verran, että nähdään, sopiiko se vastaavaan hahmoon hahmo_i.
  - Jos kaikki sopivat, niin ryhdytään tutkimaan tämän säännön haaroja sovituksen tuottamilla muuttujien arvoilla.
  - Erityisesti samaa muuttujaa ei saa toistaa saman säännön eri hahmoissa.

- Haskell sallii myös funktion *osittaisen soveltamisen* (partial application) argumentteihinsa:
  - Funktiolle voi antaa *vähemmän* argumentteja:

    $$f\ \mathit{lauseke}_1\ \ldots\ \mathit{lauseke}_q.$$

  - Tällainen lauseke on tyyppiä

    $$\mathit{tyyppi}_{q+1}\ \texttt{->}\ \ldots\ \texttt{->}\ \mathit{tyyppi}_p\ \texttt{->}\ \mathit{tulostyyppi}$$

    eli jatkaen samaa ideaa *uusi* $p-q$ *parametrin funktio*.
  - Kun se saa puuttuvat argumentit

    $$\mathit{lauseke}'_{q+1}\ \ldots\ \mathit{lauseke}'_p$$

    niin tulos on

    $$f\ \mathit{lauseke}_1\ \ldots\ \mathit{lauseke}_q\ \mathit{lauseke}'_{q+1}\ \ldots\ \mathit{lauseke}'_p.$$

- Siis moniparametrisen funktion tyyppi onkin implisiittisesti

  $$f\ \texttt{::}\ \mathit{tyyppi}_1\ \texttt{->}\ (\mathit{tyyppi}_2\ \texttt{->}\ (\ldots\ \texttt{->}\ (\mathit{tyyppi}_p\ \texttt{->}\ \mathit{tulostyyppi})\ldots)))$$

  eli
  - sen tyyppi onkin oikeasti "yksiparametrinen funktio$_1$, jonka arvo on toinen yksiparametrinen funktio$_2$, jonka arvo on kolmas ... jonka arvo on p. yksiparametrinen funktio$_p$, jonka arvo on tyyppiä *tulostyyppi*"
  - tyyppikonstruktori -> on oikealle assosiatiivinen (right associative).

- Siten yhden monikon ottava funktio tyyppiä

  $$(\mathit{tyyppi}_1, \mathit{tyyppi}_2, \mathit{tyyppi}_3, \ldots, \mathit{tyyppi}_p)\ \texttt{->}\ \mathit{tulostyyppi}$$

  voidaan esittää myös monen parametrin funktiona tyyppiä

  $$\mathit{tyyppi}_1\ \texttt{->}\ \mathit{tyyppi}_2\ \texttt{->}\ \mathit{tyyppi}_3\ \texttt{->}\ \ldots\ \texttt{->}\ \mathit{tyyppi}_p\ \texttt{->}\ \mathit{tulostyyppi}.$$

- Tämän tyyppimuunnoksen nimi on *Currying* (suomeksi vaikkapa "kuritus")

- Esimerkiksi Haskell-vakiokirjastosta `Prelude` löydämme seuraavat korkeamman kertaluvun kuritetut listafunktiot (☞ 1.3.7):

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- Ensimmäiseksi tulee se funktiotyypin parametri `f` jolla näitä funktioita voi muokata eri tilanteisiin.
- Tyyppi `[a]` tarkoittaa listaa, jonka alkiot ovat tyyppiä `a` (☞ 3).
- Tyypit `a` ja `b` ovat *muuttujia*
  - koska ne alkavat pienellä kirjaimella.
  - Intuitiivisesti ne saavat olla mitä tahansa tyyppejä.
  - Haskell-kielessä voi siis ilmaista tietyn laista *yleiskäyttöisyyttä* (☞ 4.4).

- Näillä funktioilla operaatio "laske annetun lukulistan lukujen neliöiden summa" voidaan kirjoittaa lyhyesti

```
1   sos =
2       foldl sum 0 . map square
3       where sum x y =
4                 x + y
5             square z =
6                 z * z
```

- `map square` on funktio, joka neliöi saamansa listan jokaisen luvun
- `foldl sum 0` on funktio, joka laskee saamansa listan lukujen summan

- '`.`' on `Prelude`ssa määritelty operaattori jonka

  intuitio on "f `.` g tarkoittaa funktiota h, joka
  - ohjaa saamansa parametrin funktioon g
  - ohjaa funktion g tuloksen funktioon f
  - palauttaa tuloksenaan funktion f antaman tuloksen"

  määritelmä voisi olla korkeamman kertaluvun funktio

```
f . g = h where h x = f (g x)
```

  matemaattinen vastine on *yhdistetty funktio* $f \circ g$ — joka on *funktioiden välinen* laskutoimitus!

- Tämä esimerkki osoittaa, miten funktionaalinen ohjelmointi helpottaa *abstraktiotason nostamista:*
  - Funktion `sos` määrittely voidaan lukea operaationa *lukulistoilta lukulistoille*.
  - Määrittelyssä ei kuitenkaan tarvitse käydä listoja läpi luku kerrallaan.
  - Operaatio on siis määritelty "korkeammalla tasolla" kuin yksittäisten lukujen taso.
  - Korkeimmillaan se on määritelty jopa ei edes listojen vaan *listafunktioiden* välisenä laskutoimituksena:

    '`.`' liittää neliöijän summaajaan.

2.2.7 Hahmonsovituslauseke

- Haskell-kielessä on lauseke, joka yleistää (esimerkiksi) Java-kielen `switch`-lausetta hahmonsovituksella:

  `case` *valintalauseke*
  `of` *valintasäännöt*

- Tässä *valintasäännöt* ovat muuten kuin funktion f määrittelyn säännöt (☞ 2.2.1) paitsi että
  - hahmojen edessä ei ole funktion nimeä f
  - tulosten edessä onkin nuoli '->' eikä '='.

- Itse asiassa yksiparametrisen funktion määrittely

  f *hahmo*$_1$ *haarat*$_1$
  ⋮
  f *hahmo*$_m$ *haarat*$_m$

  voidaankin kirjoittaa muodossa

  $f\ x$ = `case` x
  `of` *hahmo*$_1$ *haarat'*$_1$
  ⋮
  hahmo$_m$ *haarat'*$_m$

  missä x on uusi muuttujasymboli ja jokainen *haarat'*$_i$ on *haarat*$_i$ jonka '=' onkin '->'.

- Samoin k-parametrisen funktion määrittely (☞ 2.2.6)

  f *hahmo*$_{1,1}$ ... *hahmo*$_{1,k}$ *haarat*$_1$
  ⋮
  f *hahmo*$_{m,1}$ ... *hahmo*$_{m,k}$ *haarat*$_m$

  voidaankin kirjoittaa monikkojen (☞ 2.1.4) avulla

  $f\ x_1 \ldots x_k$ =
  `case` $(x_1,\ldots,x_k)$
  `of` (*hahmo*$_{1,1}$,...,*hahmo*$_{1,k}$) *haarat'*$_1$
  ⋮
  (*hahmo*$_{m,1}$,...,*hahmo*$_{m,k}$) *haarat'*$_m$

- `case`-lausekkeen semantiikan voi siis lukea funktiokutsun semantiikan (☞ 2.2.2) kautta:
  - Sovitetaan *valintalausekkeen* arvoa (sievennellen tarpeen mukaan) *valintasääntöihin* ja niiden *vahteihin* kirjoitusjärjestyksessä.
  - Valitaan ensimmäinen tulos jonka hahmo sopii ja vahdit toteutuvat.
  - Jos sellaista ei ole, niin päädytään ajonaikaiseen virheeseen.
- Haskell-kielen standardissa `case` onkin se *primitiivinen hahmonsovitusoperaatio* johon kaikki muut hahmonsovitusoperaatiot palautetaan tällaisilla lähdekoodin muunnossäännöillä.

- Standardi määrittelee hahmonsovituksen toiminnan `case`-sievennyssääntöinä.

- Erityisesti siis funktion f määrittely voidaan aina muuntaa muotoon

  $$f\ x_1\ \ldots\ x_k\ \texttt{=}\ c$$

  – jossa jokainen parametri x_i on pelkkä muuttujan nimi, ja

  – kaikki eri säännöt on sulautettu yhdeksi `case`-lausekkeeksi c.

- Ohitetaan `where`-määrittelyjen (☞ 2.2.5) syntaksin aiheuttamat mutkat.

2.2.8 Nimettömät funktiot

- koska funktionaalisessa ohjelmoinnissa funktiot ovat täysivaltaisia arvoja (☞ 1.3.8) niin funktionaalisissa ohjelmointikielissä on myös sellainen *lauseke joka arvo on uusi nimetön funktio*.

- Tällaisia funktionluontilausekkeita kutsutaan λ-*lausekkeiksi* ne esitelleen λ-laskennan kunniaksi (☞ 1.1).

- Haskell-kielen notaatio on

  $$\texttt{\textbackslash}\ x\ \texttt{->}\ e$$

  jossa

  – takakenoviiva \ vastaa kreikkalaista kirjainta λ

  – x on muuttuja, joka nimeää funktion parametrin

  – e on tämän funktion määrittelevä lauseke

  – muuttujan x esiintymät lausekkeessa e viittaavat tähän parametriin.

- Lisäksi Haskell-kielessä on notaatio

  $$\texttt{\textbackslash}\ p\ \texttt{->}\ e$$

  jossa p on mutkikkaampi hahmo lyhenteenä

  $$\texttt{\textbackslash}\ x\ \texttt{->}\ \texttt{(case}\ x\ \texttt{of}\ p\ \texttt{->}\ e\texttt{)}$$

  jossa x on uusi muuttuja.

- Siis parametrina voi olla myös sellainen hahmo p — kunhan hahmonsovitus onnistuu. . .

- Samoin on myös notaatio

$$\backslash\ p_1\ p_2\ p_3\ \ldots\ p_k\ \texttt{->}\ e$$

lyhenteenä lausekkeelle

$$\backslash\ p_1\ \texttt{->}\ \backslash\ p_2\ \texttt{->}\ \backslash\ p_3\ \texttt{->}\ \ldots\ \texttt{->}\ \backslash\ p_k\ \texttt{->}\ e.$$

- Näin voi määritellä moniparametrisia funktioita (☞ 2.2.6).

- λ-lausekkeen semantiikka on Leibnizin lain (☞ 1.3) hengessä sievennyssääntö, jonka mukaan funktiokutsulauseke (☞ 2.2.2)

$$(\backslash\ x\ \texttt{->}\ e_{\text{runko}})\ e_{\text{parametri}}$$

sievenee sellaiseksi lausekkeeksi, joka saadaan korvaamalla lausekkeessa e_{runko} muuttujan x esiintymät lausekkeella $e_{\text{parametri}}$.

- Näin λ-lausekkeet ovat formaali muotoilu intuitiiviselle parametrinvälitysmekanismillemme (☞ 2.2.2).

- Tämä korvaus tehdään kuitenkin vain niihin esiintymiin, jotka liittyvät juuri tähän x
  - eli niihin, jotka eivät ole minkään muun λ-lausekkeen $\backslash\ x$ -> ... tai hahmon $\ldots x \ldots$ sisällä.
  - koska nehän peittävät tämän x näkyvistä leksikaalisten näkyvyyssääntöjen (☞ 2.2.3) nojalla.

- Erityisesti siis funktion f määrittely voidaan aina muuntaa edelleen muodosta (☞ 2.2.7)

$$f\ x_1\ \ldots\ x_k\ \texttt{=}\ c$$

muotoon

$$f\ \texttt{=}\ \backslash\ x_1\ \ldots\ x_k\ \texttt{->}\ c.$$

- Funktionaalinen ohjelmoija lyhentää esimerkiksi Haskell-funktion "annetun lukulistan neliösumma" (☞ 2.2.6) muotoon

```
sos =
    foldl (\ x y -> x + y) 0 . map (\ z -> z * z)
```

koska apufunktiot `sum` ja `square` ovat niin yksinkertaisia.

- Niiden tarkoitus on näet niin selvä, ettei niiden määrittely erikseen antaisi lisätietoa ohjelmakoodin lukijalle.

2.3 Operaattoreista

- Binääriset operaattorit kuten '+', '`&&`' ja '`.`' ovat muuten tavallisia 2-parametrisia funktioita, mutta
  - niiden nimet koostuvat muista merkeistä kuin kirjaimista ja numeroista
  - ne on ilmoitettu kirjoitettaviksi *syntaktisesti* parametriensa *väliin* sanomalla

    `infix` *presedenssi* $op_1, op_2, op_3, \ldots$

    * jokainen op_i on kyseinen operaattori
    * *presedenssi* on sen sidontavoima 0 . . . 9 (tai jos se puuttuu, niin oletuksena on 9).

- Mitä suurempi *presedenssi* on, sitä pienempi on se alue, jolta operaattori parametrinsa ottaa usean *eri* operaattorin lausekkeessa: Esimerkiksi kertolaskulla '`*`' on 7 ja yhteenlaskulla '+' on 6, joten lausekkeessa p`*`q+r parametrit ryhmitellään (p`*`q)+r eikä p`*`(q+r).

- Sanomalla `infixl` voi määritellä operaattorin *vasemmalle assosioivaksi* (left associative):
  - Silloin *saman* operaattorin lausekkeet ryhmitellään vasemmalta alkaen.
  - Esimerkiksi vähennyslasku '-' assosioi vasemmalle, joten p-q-r ryhmitellään (p-q)-r eikä p-(q-r).

- Oikealla assosiointi voidaan vastaavasti sanoa `infixr`.

- Operaattorista voi ottaa myös *viipaleen* (section):
  - Lauseke (p *operaattori*) antaa *operaattori*lle vain sen vasemman parametrin p — eli on 1-parametrinen funktio \ q -> p *operaattori* q.
  - (*operaattori* q) antaa vain sen oikean parametrin q.
  - Samoin pelkkä (*operaattori*) yksinään suluissa tarkoittaa sitä vastaavaa 2-paikkaista funktiota — eli riisuu siltä hetkeksi operaattorin syntaksin.

- Silloin "annetun lukulistan neliösumma" (☞ 2.2.8) on

```
sos = foldl (+) 0 . map (\ z -> z * z)
```

- Myös tavallinen aakkosnumeerinen 2-paikkainen funktion *nimi* voi olla operaattorina, jos se kirjoitetaan *taaksepäin nojaavien heittomerkkien väliin*

  ‘*nimi*‘.

- Tällainen on esimerkiksi kokonaislukujen jakojäännös `mod` (eli Javan $p\%q$).

- Se voidaan kirjoittaa kahdella tavalla: parametriensa

  eteen tavallisessa muodossa `mod` p q

  väliin muodossa p ‘`mod`‘ q.

  Silloin sovelletaan täyttä operaattoreiden syntaksia: presedenssiä, assosiatiivisuutta ja sektioita.

2.4 Kommentit

Lyhyet kommentit

alkavat kahdella peräkkäisellä väliviivalla '`--`'

päättyvät saman rivin loppuun.

Pitkät kommentit

alkavat avaavalla aaltosululla ja väliviivalla '`{-`'

päättyvät väliviivaan ja sulkevaan aaltosulkuun '`-}`'

sallivat sisäkkäisyyden `{-...{-...-}...-}`.

3 Listat

- Funktionaalisessa ohjelmoinnissa *listat* ovat keskeinen tietorakenne:
  - Niitä on tapana käyttää aina silloin, kun käsitellään *tietoalkioiden kokoelmia*.
  - Mutkikkaampiin tietorakenteisiin (kuten puihin) siirrytään vasta jos listat osoittautuvat liian hitaiksi.
  - Laiskoja listoja voidaan käyttää kuten *iteraattoreita*.
- Siksi funktionaalisissa ohjelmointikielissä — myös Haskellissa — on sisäänrakennettu tuki listoille ja niiden käsittelylle.

- Haskell-kielen listakäsite on *induktiivinen:* Lista on

  joko tyhjä eli täysin alkioton lista, jota merkitään `[]`

  tai epätyhjä lista

  $$h : t$$

  missä

  - h on listan ensimmäinen alkio, ja
  - t on vuorostaan yhtä alkiota lyhyempi loppulista, jonka loputkin alkiot ovat samaa tyyppiä τ kuin h

- Tässä ':' on *listanrakennusoperaattori*.

- Huomaa: epätyhjän listan kaikki *alkiot ovat keskenään samaa tyyppiä* τ!

- Tämä tyyppi "lista, jonka alkiot ovat tyyppiä τ" kirjoitetaan tyyppimäärittelyissä $[\tau]$.

- Listatyypin $[\tau]$ hahmot/lausekkeet kirjoitetaan samalla notaatiolla

$$h : t$$

  missä

  – h on hahmo/lauseke tyyppiä τ

  – t on hahmo/lauseke tyyppiä $[\tau]$.

- Listahahmoissa/-lausekkeissa on yksinkertaistavia lyhennysmerkintöjä:

  – Listanrakennusoperaattori ':' assosioi oikealle:

$$h_1{:}h_2{:}h_3{:}\ldots$$

  tarkoittaa

$$h_1{:}(h_2{:}(h_3{:}\ldots)).$$

  – Tasan m alkiosta koostuva lista

$$h_1{:}h_2{:}h_3{:}\ldots{:}h_m{:}[]$$

  voidaan kirjoittaa

$$[h_1,h_2,h_3,\ldots,h_m].$$

- Hahmonsovitussäännöt (☞ 2.2.2) laajenevat listoille:

  – Listatyyppistä *lauseke*tta sievennetään sen verran, että selviää onko sen *arvo*

    tyhjä lista [] vai

    epätyhjä lista muotoa $h{:}t$ sieventämättä vielä sen osia h ja t — laiskuusperiaate.

  – Jos *arvo* = *hahmo* = [] niin sopii.

  – Jos *arvo* = $h{:}t$ ja *hahmo* = $h'{:}t'$ niin yritetään sovittaa h ja h' sekä t ja t'.

  – Muuten toinen on tyhjä ja toinen epätyhjä — ei sovi.

3.1 Listarekursio

- Myös listojen käsittelyfunktiot ohjelmoidaan usein niiden induktiivista rakennetta noudattaen:

  – Tyhjä lista [] on rekursion pysähtyvä erikoistapaus.

    Tai yleisemmin, ne tapaukset jotka käsittelevät *kiinteän monta* alkiota sisältävät listat $[h_1,h_2,h_3,\ldots,h_m]$.

  – Rekursiivinen tapaus käsittelee listan ensimmäisen alkion h ja etenee rekursiivisesti loppulistaan t.

    Tai yleisemmin, *kiinteän monta* ensimmäistä alkiota $h_1{:}h_2{:}h_3{:}\ldots{:}h_m{:}t$.

- Perusesimerkkejä ovat `foldl` ja `map` (☞ 2.2.6).
- Tarkastellaan hieman suurempana esimerkkinä Tietorakenteet-kurssilta tuttua *pikajärjestämistä* (quicksort).
- Rivien 1 ja 2 mukaan tyhjä lista on helppo `järjestää`.
- Rivillä 3 otetaan epätyhjän listan ensimmäinen alkio `jakoalkio`ksi hahmonsovituksella, kun taas `muut` alkiot käsitellään rekursiivisesti (mutta ei suoraan).
- Riveillä 4 ja 5 tehdään `ositus`, jossa `pienet` alkiot ovat pienempiä kuin `jakoalkio` ja `suuret` ainakin yhtä suuria.


```
1   järjestä [] =
2       []
3   järjestä (jakoalkio:muut) =
4       let (pienet,suuret) =
5               ositus (< jakoalkio) muut
6       in (järjestä pienet)++jakoalkio:(järjestä suuret)
7
8   ositus _ [] =
9       ([],[])
10  ositus testi (x:xs)
11      | testi x =
12          (x:joo,ei)
13      | otherwise =
14          (joo,x:ei)
15      where (joo,ei) =
16                ositus testi xs
```


- Rivi 6
  - `järjestää pienet`
  - liittää niiden perään (apuoperaatorilla '++')
  - `järjestämänsä suuret`
  - joiden edessä on `jakoalkio`.
- Apufunktio `ositus` puolestaan palauttaa parin
  - `joo` eli ne syötelistan alkiot jotka täyttivät ehdon `testi`
  - `ei` eli ne muut.

3.2 Listainduktiosta

- Koska tyypin $[\tau]$ arvot on *määritelty* induktiolla, niin niistä voidaan myös *todistaa* väitteitä induktiolla niiden määrittelyn suhteen.
- Induktio pätee kuitenkin vain *äärellisen* kokoisiin arvoihin.
- Haskell-kielen laiska suoritusmekanismi (☞ 2.2.2) sallii myös *äärettömät* listat (☞ 3.5) (ja muut tietorakenteet).

  Niille voitaisiin soveltaa ns. *koinduktiota* (coinduction) mutta sivuutetaan se.

- Yksinkertaisin listainduktion muoto on induktio *rakenteen* (structural) suhteen:

  Perustapaus: Todistetaan, että haluttu väite pätee tyhjälle listalle `[]`.

  Induktio: Tarkastellaan epätyhjää listaa

  1. Oletetaan, että väite pätee sen loppulistalle t.
  2. Todistetaan, että oletuksen nojalla se pätee myös koko listalle $h{:}t$.

- Vertaa luonnollisten lukujen perusinduktio: Todistetaan väite arvolle 0. Oletetaan väite edellisellä arvolla n. Todistetaan väite seuraavalla arvolla $n + 1$.

- Tarkastellaan esimerkkinä `Preluden` tarjoamaa listojen katenointioperaattoria:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Osoitetaan, että se on liitännäinen, eli että

$$x \mathbin{++} (y \mathbin{++} z) = (x \mathbin{++} y) \mathbin{++} z$$

  pätee kaikilla listoilla x, y ja z.

- Edetään rakenteisella induktiolla yli listan x.

  (Eli oletetaan x äärelliseksi.)

Perustapaus $x = \texttt{[]}$**:** Väitteen vasen puoli

$$\begin{aligned} x \mathbin{++} (y \mathbin{++} z) &= \texttt{[]} \mathbin{++} (y \mathbin{++} z) \\ &= (y \mathbin{++} z) \end{aligned}$$

ja oikea puoli

$$\begin{aligned} (x \mathbin{++} y) \mathbin{++} z &= (\texttt{[]} \mathbin{++} y) \mathbin{++} z \\ &= (y) \mathbin{++} z \end{aligned}$$

antavat saman arvon

$$y \mathbin{++} z$$

katenoinnin ensimmäisen säännön perusteella.

Induktio-oletus: $x = h{:}t$ ja väite pätee loppulistasta t eli

$$t \mathbin{++} (y \mathbin{++} z) = (t \mathbin{++} y) \mathbin{++} z.$$

Induktioaskel: Väitteen vasen puoli

$$\begin{aligned} x \mathbin{++} (y \mathbin{++} z) &= (h : t) \mathbin{++} (y \mathbin{++} z) \\ &= h : \underline{(t \mathbin{++} (y \mathbin{++} z))} \end{aligned}$$

ja oikea puoli

$$\begin{aligned} (x \mathbin{++} y) \mathbin{++} z &= ((h : t) \mathbin{++} y) \mathbin{++} z \\ &= (h : (t \mathbin{++} y)) \mathbin{++} z \\ &= h : \underline{((t \mathbin{++} y) \mathbin{++} z)} \end{aligned}$$

antavat saman arvon, koska induktio-oletuksen mukaan <u>alleviivatut</u> arvot ovat samat.

- Todistuksemme oli näin yksinkertainen koska:
  - Puhtauden vuoksi ei ole väliä, missä järjestyksessä eri operaatiot '++' suoritetaan (☞ 1.3.6).
  - Saimme siis valita sellaisen järjestyksen, jolla todistuksemme oli helpoin — muuten olisimme joutuneet aina seuraamaan sitä järjestystä, jossa se oikeasti etenee.
  - Viittausten läpikuultavuuden (☞ 1.3.1) vuoksi kaksi listaa ovat samat aina kun ne syntyvät samanlaisilla lausekkeilla.
  - Saimme siis tehdä tuttuja '='-päättelyaskeleita — muuten olisimme joutuneet joka vaiheessa sanomaan "sisältävät samat alkiot" tms.

- Tämä väite sattuu pätemään myös silloin, kun x onkin ääretön: koinduktiolla voitaisiin osoittaa, että silloin

$$x \mathbin{++} \ldots = x.$$

- Induktio rakenteen suhteen ei aina riitä:
  - Esimerkiksi pikajärjestämisessä (☞ 3.1) rekursio ei käsitellytkään listan loppuosaa `muut` vaan *toisia* listoja `pienet` ja `suuret`.
  - Ne ovat kuitenkin aidosti lyhyempiä kuin `jakoalkio:muut`.
  - Voimme siis edetä induktiolla ei listan rakenteen vaan sen *pituuden* suhteen.

Perustapaus: Osoitetaan, että todistettava väite pätee 0 alkiota pitkälle eli tyhjälle listalle `[]`

Induktio-oletus: Oletetaan, että väite pätee kaikille listoille, joiden pituus on $\leq n$ alkiota — eli kaikille nykyistä listaa aidosti lyhyemmille listoille.

Induktioaskel: Osoitetaan induktio-oletusta käyttäen, että väite pätee myös $n+1$ alkiota pitkille listoille — eli nykyiselle listalle, joka on saatu lisäämällä aidosti lyhyemmän listan eteen yksi uusi alkio.

- Vertaa luonnollisten lukujen yleinen induktio, jossa väite oletetaan *kaikille pienemmille* luvuille $\leq n$.

- Osoitetaan induktiolla syötelistan pituuden n suhteen, että

$$\texttt{järjestä}\ [x_1, x_2, x_3, \ldots, x_n]$$

palauttaa syötteensä jonkin sellaisen permutaation (esimerkiksi)

$$[x_3, x_{26}, x_n, \ldots, x_7]$$

joka on järjestyksessä, eli

$$x_3 \leq x_{26} \leq x_n \leq \ldots \leq x_7.$$

Perustapaus: `järjestä [] = []`.

Induktio-oletus: Olkoon y mikä tahansa lista, jonka pituus $\leq n$ alkiota. Silloin `järjestä` y palauttaa listan y jonkin järjestetyn permutaation.

Induktioaskel alkaa: Tarkastellaan kutsua `järjestä` z missä listan z pituus on $n + 1$ alkiota

1. Rivin 3 hahmossa on $z =$`jakoalkio:muut`.

   Erityisesti loppulistan `muut` pituus on vain $= n$ alkiota.

Induktioaskel jatkuu:

2. Jätetään lukijan harjoitustehtäväksi aputulos:

   `osita` p u=(v,w) missä lista $v +\!\!+ w$ on listan u jokin sellainen permutaatio, jossa

   - kaikki listan v alkiot täyttävät ehdon p, mutta
   - mikään listan w alkioista ei täytä sitä.

3. Aputuloksen 2 nojalla rivit 4 ja 5 permutoivat loppulistan `muut` siten, että

   - permutaatio `pienet` koostuu sen niistä alkioista, jotka ovat < `jakoalkio`, ja
   - muut sen alkiot ovat permutaationa `suuret`.

Induktioaskel päättyy:

4. Listan `pienet` pituus on korkeintaan yhtä suuri kuin listan `muut` eli $\leq n$ alkiota.

   - Saamme soveltaa induktio-oletustamme rekursiokutsuun `järjestä pienet`.
   - Se siis permutoi `pienet` järjestykseen.
   - Samoin `järjestä suuret`.

5. Silloin myöskin rivillä 6 konstruoitava permutaatio on väitteen mukainen.

- Induktio listan rakenteen tai pituuden suhteen riittää monissa tilanteissa — mutta ei aina.

3.3 Merkit ja merkkijonot

- Haskell-kielessä on myös perustyyppi *merkki* eli `Char`.
- Merkkivakio kirjoitetaan Javan tapaan yksinkertaisten lainausmerkkien väliin: 'c'.
  - Merkki c kirjoitetaan Javan tyyliin
  - käyttäen Unicode-merkistöä.

- Haskell-kielessä on myös *merkkijonotyyppi* `String`.
- Se on kuitenkin vain toinen nimi tyypille `[Char]`.

  Siis merkkijonot ovat *merkkien listoja*.
- Merkkijonovakio kirjoitetaan Javan tyyliin kaksinkertaisten lainausmerkkien väliin: "$c_1c_2c_3\ldots c_n$" jossa jokainen c_i on merkkivakio.
- Merkkijonoja voi siis käsitellä kaikilla listafunktiolla.

- Lisäksi merkkijonoille on omiakin funktioita kuten `words` joka jakaa merkkijonon "sanoihin":

```
Prelude> words "Minäpä olen se pikkuinen Matti..."
["Min\228p\228","olen","se","pikkuinen","Matti..."]
```

- Tarkastellaan esimerkkinä lista- ja merkkijonokirjastofunktioiden yhteispelistä *konkordanssin* (concordance) laatimista:
  - Eli järjestetun listan pareja

    (tekstissä esiintynyt sana, esiintymien lukumäärä).
  - Yksinkertaistus 1: muut kuin aakkoset voi unohtaa.
  - Yksinkertaistus 2: isot kirjaimet voi tulkita pieniksi.


```
1   import List
2   import Char
3   concord :: String -> [(String,Int)]
4   concord =
5       map (\ g -> (head g,length g)) .
6           group .
7                 sort .
8                      words .
9                            map toLower .
10                               filter kiintoisat
11      where kiintoisat c =
12                isAlpha c || isSpace c
```

- Rivit 1 ja 2 liittyvät moduuleihin (☞ 6); intuitiivisesti ne tarkoittavat samaa kuin Javassakin.

3.4 Funktioiden samuudesta

- Edellä (☞ 3.2) pääteltiin tuloksia arvoista (listoista).
- Näitä tuloksia voi "nostaa" funktioiden tasolle *ekstensionaalisuusperiaatteella* (extensionality principle):

  Jos palautusarvot $f\ a = g\ a$ kaikilla argumenteilla a, niin myös vastaavat funktiot $f = g$
  (ja kääntäen).
- Koska ohjelmat voivat myös jäädä ikuiseen silmukkaan, niin "palautusarvoihin" täytyy ottaa myös '$\perp$'.

- Osoitetaan esimerkiksi, että funktioilla

$$\texttt{foldl} \oplus 0_l = \texttt{foldr} \oplus 0_r \qquad \text{missä:}$$

  - $\oplus$ on jokin liitännäinen 2-paikkainen laskutoimitus, eli $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ kaikilla x, y ja z.
  - 0_l on sen "vasen nolla" eli $0_l \oplus y = y$ kaikilla y. Vastaavasti 0_r on sen oikea nolla.
  - Sekä `foldl` (☞ 2.2.6) että `foldr` ovat vakiokirjastosta:

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs).
```

  - Jos $x \oplus y \neq \bot$ niin $x \oplus y \neq x \oplus z$ jollakin z (†).

`foldl` **"taittaa listan vasemmalle":**

$$\texttt{foldl} \otimes p\,[x_1, x_2, x_3, \ldots, x_n] = ((((p \otimes x_1) \otimes x_2) \otimes x_3) \ldots) \otimes x_n.$$

`foldr` **taas oikealle:**

$$\texttt{foldr} \otimes q\,[x_1, x_2, x_3, \ldots, x_n] = x_1 \otimes (x_2 \otimes (x_3 \otimes (\ldots (x_n \otimes q)))).$$

- Osoitetaankin muunnettu vahvempi väite: kaikilla b pätee

$$\texttt{foldl} \oplus b = (b \oplus) \circ (\texttt{foldr} \oplus 0_r).$$

  Alkuperäinen väite saadaan valitsemalla $b = 0_l$.

- Edetään induktiolla näiden kahden funktion saaman lista-argumentin a suhteen:

$$\texttt{foldl} \oplus b\,a = b \oplus (\texttt{foldr} \oplus 0_r\,a).$$

- Perustapaus $a = \texttt{[]}$ on

$$\begin{aligned} \texttt{foldl} \oplus b\,\texttt{[]} &= b \\ b \oplus (\texttt{foldr} \oplus 0_r\,\texttt{[]}) &= b \oplus 0_r \\ &= b. \end{aligned}$$

- Induktiivinen tapaus $a = x : xs$ on

$$\begin{aligned} \texttt{foldl} \oplus b\,(x : xs) &= \texttt{foldl} \oplus \overbrace{(b \oplus x)}^{\text{uusi } b}\,xs \\ b \oplus (\texttt{foldr} \oplus 0_r\,(x : xs)) &= b \oplus (x \oplus (\texttt{foldr} \oplus 0_r\,xs)) \\ &= \underbrace{(b \oplus x)}_{\text{uusi } b} \oplus (\texttt{foldr} \oplus 0_r\,xs) \end{aligned}$$

  ja ne ovat samat induktio-oletuksen perusteella.

- Entäpä jos $a = [x_1, x_2, x_3, \ldots]$ onkin ääretön?
  - `foldl` "palauttaa" $\perp$ — se yrittää käydä läpi koko listan a
  - mutta `foldr` voikin *lopettaa kesken* läpikäynnin, jos operaatio '$\oplus$' voidaankin sieventää unohtaen kokonaan sen jälkimmäinen argumentti.
  - mutta viimeinen oletuksistamme (†) sulkee sen pois.
- Väitteemme on siis todistettu.
- Jos suoritusmekanismi olisi ahkera, niin (†) olisi tarpeeton, koska myös `foldr` yrittäisi aina käydä läpi äärettömänkin a.