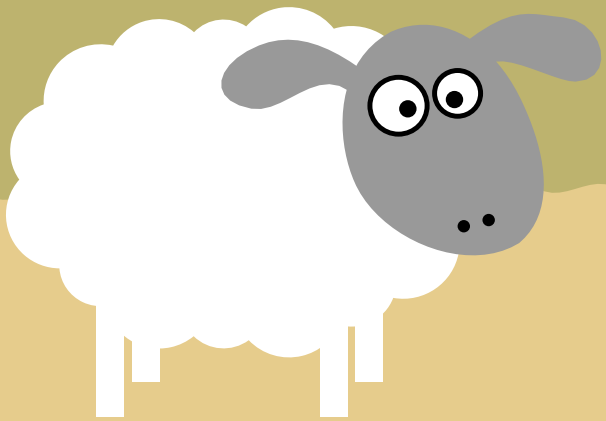
Haskell

ohjelmointikieli

• 13.6.2019 • Jarmo Hietala •



Luku 1

Haskell-ohjelmointikieli

Haskell-kieli on nykyaikainen funktionaalinen ohjelmointikieli. Kielen ensimmäinen versio julkaistiin vuonna 1990. Haskell-ohjelmat voidaan joko kääntää konekielelle tai ajaa tulkattuina. Käännetyt ohjelmat toimivat itsenäisesti ja ovat hyvin tehokkaita. Tulkkaus puolestaan mahdollistaa vuorovaikutteisuuden kielen kanssa. Tavallisesti modernit tietokoneohjelmat sisältävät sekä käännettyjä että tulkattavia ominaisuuksia.

1.1 Vuorovaikutteinen tulkki `ghci` ja kääntäjä `ghc`

Aloitamme tutustumisen Haskell-kieleen käynnistämällä Haskell-kääntäjän (*Glasgow Haskell Compiler*) interaktiivisen tulkin komennolla

```
ghci
```

Kun myöhemmin käännämme ohjelmia lähdetiedostoista, käytämme Haskell-kääntäjää komennolla `ghc`. Esimerkiksi ohjelman lähdetiedostosta `example.hs` kääntäisimme komennolla

```
ghc example.hs
```

Kokeiluvaiheessa voimme käynnistää ohjelman (esimerkiksi `example.hs`) lähdetiedostosta Haskell-tulkin suoritettavaksi ilman käännösvaihetta komennolla

```
runhaskell example.hs
```

Käynnistettäessä Haskell-kääntäjän tulkki kertoo ohjelman version ja jää odottamaan käyttäjän komentoja.

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
>
```

Voimme käyttää Haskell-tulkkia vaikkapa laskimena.

```
> 64 * 16
1024
```

Haskell-kielen standardikirjasto `Prelude` tarjoaa oletuksena käytettäväksemme joukon funktioita ja tietorakenteita, joihin tutustumme tässä ensimmäisessä luvussa.

1.2 Prefix- ja infix-funktiomuodot

Haskell-kielessä funktiot esiintyvät *prefix-* ja *infix*-muodoissa.

Nimitämme prefix-muotoiseksi funktiota, jossa funktionimi edeltää parametreja. Funktiota, jossa funktionimi on parametrien välissä, nimitämme infix-muotoiseksi.

Haskell-kielessä kirjoitamme aakkosnumeeriset funktionimet prefix-muodossa sellaisenaan, infix-muodossa ympäröimme ne niin sanotuilla yksinkertaisilla takalainausmerkeillä `` `f` `` (englanniksi *backtick*, *grave accent*, ASCII-koodi 96).

Operaattorit (eli muut kuin aakkosnumeeriset funktionimet) ympäröimme prefix-muodossa kaarisulkeilla `()`, infix-muodossa kirjoitamme ne sellaisenaan.

Esimerkiksi Haskell-kielen infix-muotoisen lausekkeen `2 + 3` voimme esittää prefix-muodossa seuraavasti:

```
> (+) 2 3
5
```

Tietueiden `(x,y)` ja listojen `[x,y]` esitystapaa Haskell-kielessä nimitämme *mixfix*-muodoksi.

1.3 Funktiot `min` ja `max`

Funktio `min` palauttaa kahdesta annetusta argumentista pienemmän ja funktio `max` suuremman. Kutsumme näitä funktioita seuraavassa prefix-muodossa.

```
> min 2 5
2
> max 5 7
7
```

Voimme kirjoittaa saman myös infix-muodossa

```
> 2 `min` 5
2
> 5 `max` 7
7
```

1.4 Funktiot `abs`, `even` ja `odd`

Funktiokutsu `abs x` palauttaa luvun x itseisarvon $|x|$.

```
> abs (-3)
3
```

Funktio `even` palauttaa totuusarvon `True` annetun argumentin ollessa parillinen, muutoin se palauttaa arvon `False`. Funktio `odd` palauttaa totuusarvon `True` annetun argumentin ollessa pariton, muutoin se palauttaa arvon `False`.

```
> odd 2
False
> even 4
True
```

1.5 Operaattorit `(+)`, `(-)`, `(*)` ja `(/)`

Peruslaskutoimitukset on Haskell-kielessä toteutettu funktioina `(+)`, `(-)`, `(*)` ja `(/)`. Käytämme näitä operaattoreita yleisemmin infix-muodossa.

```
> 3 - 4
-1
> 4 * 5
20
> 1 / 2
0.5
```

Desimaalilukujen desimaalierottimena toimii piste ja merkitsemme edeltävän nollan näkyviin matemaattisia merkintätapoja seuraten. Negatiiviset luvut ympäröimme kaarisulkeilla `()`, jolloin kääntäjä erottaa ne vähennyslaskuoperaatiosta.

```
> 5 * (-0.02)
-0.1
> (-1) + (-2)
-3
```

1.6 Funktiot `div`, `mod` ja `divMod`

Funktio `div` palauttaa kokonaislukujen jakolaskun kokonaisosan ja funktio `mod` jakojäännöksen.

```
> 7 `div` 3
2
> 7 `mod` 3
1
```

Funktio `divMod` palauttaa kahden alkion tietueena jakolaskun kokonaisosan ja jakojäännöksen.

```
> 7 `divMod` 3
(2,1)
```

1.7 Operaattorit `(==)`, `(/=)`, `(<)`, `(>)`, `(<=)` ja `(>=)`

Funktiot `(==)`, `(/=)`, `(<)`, `(>)`, `(<=)` ja `(>=)` määrittävät matemaattiset vertailuoperaatiot $=$, $\neq$, $<$, $>$, $\leq$ ja $\geq$. Ne kukin palauttavat totuusarvon `True` tai `False`.

```
> 5 == 5
True
> 4 /= 5
True
> 5 < 2
False
```

Muun muassa palautusarvosta ja määrittämättömästä assosiatiivisuudesta johtuen emme voi kirjoittaa useampia loogisia vertailuoperaatioita ketjuun.

```
> 1 < 2 < 5
"ERROR: Precedence parsing error."
```

Sen sijaan jaamme lausekkeen operaatiot pienempiin osiin.

```
> (1 < 2) && (2 < 5)
True
```

1.8 Funktiot `not`, `(&&)` ja `(||)`

Funktio `not` on yhden parametrin totuusarvofunktio. Se vastaa logiikan operaatiota **ei** ($\neg$) eli se palauttaa negaation annetusta argumentista.

Funktiot `(&&)` ja `(||)` ovat kahden parametrin funktioita. Ne vastaavat logiikan operaatioita **ja** ($\wedge$) ja **tai** ($\vee$). Käytämme funktioita `(&&)` ja `(||)` yleensä infix-muodossa.

Funktio `(&&)` palauttaa totuusarvon `True`, mikäli molemmat argumentit ovat arvoltaan `True`, muutoin se palauttaa arvon `False`. Funktio `(||)` palauttaa totuusarvon `True`, mikäli vähintään toinen argumentti on arvoltaan `True`, muutoin se palauttaa arvon `False`.

```
> not False
```

```
True
> not (even 3)
True
> True && False
False
> True || False
True
```

1.9 Muuttujien määrittely

Määrittelemme seuraavassa kokonaislukumuuttujat `a = 2`, `b = 3` ja `c = a + b`. Nyt muuttujan `c` arvo on `c = a + b` $= 2 + 3 = 5$.

```
> a = 2
> b = 3
> c = a + b
> c
5
```

Vastaavasti määrittelemme totuusarvomuuttujat `p = True` ja `r = False`. Nyt lausekkeen `p || r` arvo on `True || False` = `True`.

```
> p = True
> r = False
> p || r
True
```

Seuraavassa määrittelemme merkkimuuttujat `c1 = 'e'` ja `c2 = 'o'`. Lausekkeen `c1 < c2` arvo on nyt `'e' < 'o'` = `True`. (Aakkosjärjestys määrää merkkien suuruusjärjestyksen.)

```
> c1 = 'e'
> c2 = 'o'
> c1 < c2
True
```

Kun määrittelemme funktion `plus = (+)`, saamme aiemmilla muuttujien `a = 2` ja `b = 3` arvoilla lausekkeen `a `plus` b` arvoksi `a + b` $= 2 + 3 = 5$.

```
> plus = (+)
> a `plus` b
5
```

Määrittelemällä funktion `(×) = (*)` saamme lausekkeen `3 × 4` arvoksi 12.

```
> (×) = (*)
> 3 × 4
12
```

Seuraavassa merkkijonomuuttuja `s` saa arvon `s = "Mare Australe"` ja funktio `size` arvon `size = length`. Standardikirjastossa määritelty funktio `length` palauttaa merkkijonon (listan) pituuden, joten myös funktio `size` palauttaa merkkijonon (listan) pituuden. Lausekkeen `size s` arvo on siten 13.

```
> s = "Mare Australe"
> size = length
> size s
13
```

1.10 Tiedon esittäminen listoina

Nimitämme *listaksi* samaa tyyppiä olevien alkioiden jonoa. Listan merkintätapa Haskell-kielessä on ympäröivät hakasulkeet `[]`.

Esimerkkinä määrittelemme listan `ts`, jonka alkioita ovat luvut 2, 3, 5, 7 ja 11.

```
> ts = [2,3,5,7,11]
> ts
[2,3,5,7,11]
```

Nimeämme listat usein englannin kielen monikkomuodon mukaisesti päätteellä `-s` (`ts`, `xs`, …).

Lista `[]` on tyhjä lista, lista `[5]` yhden alkion lista, lista `[2,3]` kahden alkion lista, ja niin edelleen.

1.11 Funktiot `head` ja `tail`

Funktio `head` palauttaa listan ensimmäisen alkion, jota nimitämme listan *pääksi*. Funktio `tail` palauttaa ensimmäistä alkiota lukuun ottamatta loput listan alkiot, eli listan *hännän*.

```
> head ts
2
> tail ts
[3,5,7,11]
```

Havaitsemme, että funktiot `head` ja `tail` eivät sovellu tyhjien listojen käsittelyyn. Pyrimme siksi välttämään näiden funktioiden käyttöä kaikissa sellaisissa tapauksissa, joissa argumentti saattaa olla tyhjä lista.

```
> head []
"ERROR: Empty list."
> tail []
"ERROR: Empty list."
```

1.12 Funktiot `and` ja `or`

Funktiot `and` ja `or` ovat kahden parametrin funktioiden `(&&)` ja `(||)` vastineet listojen käsittelyyn.

Funktio `and` palauttaa totuusarvon `True`, mikäli listan kaikki alkiot ovat arvoltaan `True`, muutoin se palauttaa arvon `False`.

Funktio `or` palauttaa totuusarvon `True`, mikäli yksikin listan alkioista on arvoltaan `True`, muutoin se palauttaa arvon `False`.

```
> and [True,False,True]
False
> or [True,False,True]
True
```

1.13 Funktiot `any` ja `all`

Funktiokutsu `any p` palauttaa totuusarvon `True`, mikäli jokin listan alkioista täyttää ehdon `p`, muutoin se palauttaa arvon `False`. Funktiokutsu `all p` palauttaa arvon `True`, mikäli kaikki listan alkiot täyttävät ehdon `p`, muutoin se palauttaa arvon `False`.

```
> ts = [2,3,5,7,11]
> any (even) ts
True
> all (odd) ts
False
> any (> 10) ts
True
> all (< 10) ts
False
```

1.14 Funktiot `sum` ja `product`

Funktiot `sum` ja `product` palauttavat numeerisen listan alkioiden summan ja tulon.

```
> sum ts
28
> product ts
2310
```

1.15 Funktiot `minimum` ja `maximum`

Funktiot `minimum` ja `maximum` palauttavat järjestyvän listan pienimmän ja suurimman alkion.

```
> minimum ts
2
> maximum ts
```

```
11
```

1.16 Funktiot `take`, `drop` ja `(!!)`

Funktiokutsu `take n` palauttaa `n` ensimmäistä alkiota listasta. Funktiokutsu `drop n` pudottaa pois `n` ensimmäistä alkiota.

```
> ts = [2,3,5,7,11]
> take 3 ts
[2,3,5]
> drop 2 ts
[5,7,11]
```

Funktio `(!!)` palauttaa listasta alkion, jolla on argumentin mukainen järjestysluku. Listan alkioiden numerointi alkaa luvusta 0.

```
> ts !! 3
7
```

1.17 Funktiot `elem` ja `NotElem`

Funktio `elem` palauttaa totuusarvon `True` mikäli annettu alkio on listan jäsen, muutoin se palauttaa arvon `False`. Funktio `notElem` palauttaa totuusarvon `True` mikäli annettu alkio ei ole listan jäsen, muutoin se palauttaa arvon `False`.

```
> 7 `elem` ts
True
> 1 `elem` ts
False
> 1 `notElem` ts
True
```

1.18 Funktiot `last` ja `init`

Funktio `last` palauttaa listan viimeisen alkion ja funktio `init` listan alun aina toiseksi viimeiseen alkioon saakka.

```
> ts = [2,3,5,7,11]
> last ts
11
> init ts
[2,3,5,7]
```

1.19 Funktio `reverse`

Funktio `reverse` palauttaa käänteisen listan.

```
> reverse ts
[11,7,5,3,2]
```

1.20 Funktiot `length` ja `null`

Funktio `length` palauttaa listan alkioiden lukumäärän. Funktio `null` palauttaa totuusarvon `True`, mikäli lista on tyhjä, muutoin se palauttaa arvon `False`.

```
> ts = [2,3,5,7,11]
> length ts
5
> null ts
False
> null []
True
```

1.21 Kuvausfunktio `map`

Funktiokutsu `map f` kuvaa funktion `f` listalle.

```
> ts = [2,3,5,7,11]
> map (* 2) ts
[4,6,10,14,22]
> map (+ 11) ts
[13,14,16,18,22]
> map negate ts
[-2,-3,-5,-7,-11]
```

1.22 Funktio `filter`

Funktiokutsu `filter p` suodattaa listan ja jättää jäljelle ehdon `p` täyttävät alkiot.

```
> filter (< 10) ts
[2,3,5,7]
> filter even ts
[2]
> filter odd ts
[3,5,7,11]
```

1.23 Funktio `splitAt`

Funktio `splitAt` jakaa listan kahden alkion tietueeksi annetusta katkaisukohdasta. Alkioiden numerointi alkaa luvusta 0.

```
> ts = [2,3,5,7,11]
> splitAt 2 ts
([2,3],[5,7,11])
```

1.24 Tietueet

Nimitämme *tietueeksi* alkioiden jonoa, jossa alkiot eivät välttämättä ole toistensa kanssa samaa tyyppiä. Tietueen merkintätapa Haskell-kielessä on ympäröivät kaarisulkeet `()`. Esimerkkinä määrittelemme kahden alkion lukuparin `p = (2,3)`.

```
> p = (2,3)
> p
(2,3)
```

Lukupari on esimerkki vektorimuotoisesta tiedon esittämisestä. Merkintämuoto on laajennettavissa useampiulotteisiin vektoreihin, esimerkiksi `q = (2,7,5)` tai `r = (2,3,6,7)`.

```
> q = (2,7,5)
> q
(2,7,5)
> r = (2,3,6,7)
> r
(2,3,6,7)
```

Tietueen voimme määritellä myös prefix-muodossa.

```
> (,) 1 2
(1,2)
> (,,) 3 4 7
(3,4,7)
```

Ilmaisemme yksiulotteiset vektorit skalaareina. Kääntäjä tulkitsee mahdolliset sulkumerkit tällöin lausekkeen arvoon ja laskuoperaatioihin kuuluviksi.

```
> (5)
5
```

Tietueen alkiot voivat edustaa eri tyyppejä. Esimerkiksi kokonaisluvun ja merkin muodostamat parit ovat sallittuja. Samaan tapaan tietueet voivat edelleen koostua alitietueista.

```
s = (1,'a')
t = (1,2,'b')
```

```
u = ((1,'a'),(1,2,'b'),1)
```

1.25 Funktiot `fst` ja `snd`

Funktio `fst` palauttaa kahden alkion tietueesta ensimmäisen alkion. Funktio `snd` palauttaa tietueen toisen alkion.

```
> p = (2,3)
> fst p
2
> snd p
3
```

Funktioiden `fst` ja `snd` määrittelyt ovat seuraavan kaltaiset:

```
fst (x,y) = x
snd (x,y) = y
```

1.26 Lukujonot listoina

Haskell-kielessä voimme määritellä listat aritmeettisina lukujonoina antamalla jonon ensimmäisen ja viimeisen alkion. Oletuksena lista kasvaa yhden yksikön verran jokaista alkiota kohden. Ykkösestä eroavan kasvun ilmaisemme antamalla kaksi ensimmäistä alkiota listan alusta.

```
> [3..7]
[3,4,5,6,7]
> [6,5..1]
[6,5,4,3,2,1]
> [1,3..11]
[1,3,5,7,9,11]
```

Vastaavalla tavalla voimme muodostaa myös muiden järjestyvien joukkojen, kuten reaalilukujen tai merkkien, muodostamia listoja.

```
> [1.5..5.5]
[1.5,2.5,3.5,4.5,5.5]
```

```
> ['a'..'g']
"abcdefg"
```

1.27 Päättymättömät listat

Edellä kuvattua menetelmää käyttäen voimme muodostaa päättymättömiä listoja.

```
xs = [1..]
zs = [1,3..]
```

Tässä lista `xs` sisältää kaikki kokonaisluvut alkaen luvusta 1. Lista `zs` sisältää kaikki parittomat luvut alkaen luvusta 1.

1.28 Funktiot `takeWhile` ja `dropWhile`

Funktiokutsu `takeWhile p` palauttaa listan alkioita alusta niin pitkälle kuin ehto `p` on voimassa. Funktiokutsu `dropWhile p` pudottaa listan alkioita pois alusta niin pitkälle kuin ehto `p` on voimassa.

```
> takeWhile (< 10) [1,3..]
[1,3,5,7,9]
> dropWhile (< 3) [1,2,3,4,5,1,2,3]
[3,4,5,1,2,3]
```

Päättymättömiä listoja emme voi suodattaa funktiolla `filter`. Esimerkiksi funktiokutsu `filter (< 10) [1..]` palauttaa yhdeksän ensimmäistä kokonaislukua ja jatkaa lukujen seulontaa tuloksetta kunnes käyttäjä sen keskeyttää.

```
> filter (< 10) [1..]
[1,2,3,4,5,6,7,8,9 CTRL-C Interrupted.
```

Funktion `filter` sijasta käytämmekin funktiota `takeWhile`. Funktiokutsun `filter (< 10) [1..]` esitämme siksi muodossa `takeWhile (< 10) [1..]`.

```
> takeWhile (< 10) [1..]
[1,2,3,4,5,6,7,8,9]
```

1.29 Funktiot `repeat` ja `cycle`

Funktio `repeat` tuottaa päättymättömän listan, jossa annettu alkio toistuu. Funktio `cycle` tuottaa päättymättömän listan, jossa annetun listan alkiot toistuvat.

```
> repeat 1
[1,1,1,...]
> repeat [1,2]
[[1,2],[1,2],[1,2],...]
> cycle [1,2]
[1,2,1,2,1,2,...]
```

Funktiot `take` ja `head` toimivat päättymättömille listoille samaan tapaan kuin äärellisille listoille. Ne palauttavat äärellisen määrän alkioita ja jättävät päättymättömän listan muut alkiot huomiotta.

```
> take 5 [1..]
[1,2,3,4,5]
> head [1..]
1
> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]
> take 5 (repeat [1,2])
[[1,2],[1,2],[1,2],[1,2],[1,2]]
> take 10 (cycle [1,2])
[1,2,1,2,1,2,1,2,1,2]
```

1.30 Funktio `zip`

Funktio `zip` muodostaa pareja yhdistämällä alkioita kahdesta listasta alkio kerrallaan. Funktio palauttaa uuden listan, kun lyhyempi alkuperäisistä listoista on läpikäyty.

```
> ts = [2,3,5,7,11]
> vs = [1,5,10,14]
> zip ts vs
```

```
[(2,1),(3,5),(5,10),(7,14)]
```

Voimme luonnollisesti yhdistää listan itsensä tai oman häntänsä kanssa. Esimerkiksi lista `tail ts` on listan `ts` häntä. Nyt lauseke `tail ts` saa arvon `tail [2,3,5,7,11] = [3,5,7,11]` ja lauseke `zip ts (tail ts)` arvon `[(2,3),(3,5),(5,7),(7,11)]`.

```
> zip ts ts
[(2,2),(3,3),(5,5),(7,7),(11,11)]
> zip ts (tail ts)
[(2,3),(3,5),(5,7),(7,11)]
```

Kuten funktiot `take` ja `head`, myös funktio `zip` soveltuu päättymättömien listojen käsittelyyn.

```
> zip [0..] ts
[(0,2),(1,3),(2,5),(3,7),(4,11)]
> zip [1..] ['a'..'g']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f'),(7,'g')]
```

1.31 Funktio `zipWith`

Funktio `zipWith` on funktion `zip` yleistetty versio. Funktiokutsun `zipWith f` parametri `f` on funktio, joka kertoo säännön kuinka tulosalkio muodostetaan, kun lähtöalkiot saadaan annetuista listoista.

```
> zipWith (+) [1..7] [1..]
[2,4,6,8,10,12,14]
> zipWith (*) [1..7] [1..]
[1,4,9,16,25,36,49]
> zipWith (,) [1..] ['a'..'g']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f'),(7,'g')]
```

Kun funktion `zipWith` ensimmäinen argumentti on tietuekonstruktorifunktio `(,)`, saamme yhtäpitävän määrittelyn edellä esittelemällemme funktiolle `zip`.

```
> zip' = zipWith (,)
> zip' [1..] ['a'..'g']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f'),(7,'g')]
```

1.32 Tyyppijärjestelmä

Haskell-kieli on vahvasti tyypitetty kieli. Tämä tarkoittaa, että kaikilla arvoilla on tyyppi. Mikäli emme erikseen määritä arvon tyyppiä, kääntäjä päättelee tyypin. Kun arvojen tyypit ovat käännöshetkellä tiedossa, kääntyy ohjelma konekielelle tehokkaammin.

Tavanomaisia tietotyyppejä ovat kokonaislukutyyppi `Int`, liukulukutyypit `Float` ja `Double`, merkkityyppi `Char` sekä totuusarvotyyppi `Bool`.

```
i :: Int
i = 5
f :: Float
f = 1.5707964
d :: Double
d = 6.283185307179586
c :: Char
c = 'e'
b :: Bool
b = True
```

Tyyppimäärittelyn `x :: y` luemme: "x on tyyppiä y". Esimerkiksi edellä määrittelemämme muuttuja `c` on tyyppiä `Char`.

1.33 Funktiotyyppi

Matematiikasta meille ovat tuttuja kokonaislukujen joukon symboli $\mathbb{Z}$, ja funktion määrittelyjoukon merkintä, missä esimerkiksi funktio f kuvaa kokonaisluvun kokonaisluvuksi

$$f : \mathbb{Z} \to \mathbb{Z}$$

Edellä luetelluista Haskell-kielen tyypeistä vastaa matematiikan kokonaislukuja lähinnä tyyppi `Int`. Funktion kuvausnuolena toimii merkkipari `->`. Funktion, joka kuvaa tyypin `Int` luvun tyypin `Int` luvuksi, tyyppiallekirjoitus on siten

```
f :: Int -> Int
```

1.34 Abstraktit tietotyypit

Vahvasta tyypityksestä huolimatta Haskell-kielessä funktiot voivat operoida alkioilla, joiden tyyppiä emme ole erikseen määränneet. Näin on esimerkiksi silloin, kun alkioiden tyypillä ei ole funktion tuloksen kannalta merkitystä. Toisaalta ohjelman suunnittelun tasolla voimme toisinaan säilyttää tyypin käsitteen abstraktiona, joka saa käytännön toteutuksensa vasta ohjelman yksityiskohtien tarkentuessa. Sanomme tällaista tyyppiä *abstraktiksi* tyypiksi. Kuvaamme abstraktia tietotyyppiä tyyppimuuttujalla, jonka kirjoitamme pienellä alkukirjaimella, esimerkiksi `t`. Tyyppi `t` voi olla mikä tahansa tyyppi.

Abstraktit tyypit voivat esiintyä esimerkiksi funktioiden tyyppiallekirjoituksissa. Tällöin yhden parametrin funktio `f x`, joka palauttaa muuttujan `x` kanssa samaa tyyppiä olevan arvon, on tyyppiä `a -> a`.

```
f :: a -> a
```

Yhden parametrin funktio `g x`, jonka palautusarvo ei ole riippuvainen muuttujan `x` tyypistä, on tyyppiä `a -> b`.

```
g :: a -> b
```

Kahden parametrin funktio `h x y` on tyyppiä `a -> b -> c` ja kolmen parametrin funktio `k x y z` tyyppiä `a -> b -> c -> d`.

```
h :: a -> b -> c
k :: a -> b -> c -> d
```

1.35 Tyyppiluokat

Voimme asettaa abstrakteille tyypeille lisävaatimuksen kuulua tiettyihin tyyppiluokkiin. Vaatimuksena tyypille voi olla esimerkiksi (suluissa tyyppiluokka) mahdollisuus verrata alkioiden välistä yhtäsuuruutta (`Eq`), järjestyvyysominaisuus (`Ord`), lueteltavuus (`Enum`), numeerisuus (`Num`),

desimaalimuotoisuus (`Floating`), osamäärämuotoisuus (`Fractional`), mahdollisuus esittää arvo merkkijonona (`Show`) tai mahdollisuus lukea arvo merkkijonosta (`Read`).

Esitämme tyypille asetettavan lisävaatimuksen seuraavasti:

```
(+) :: Num a => a -> a -> a
min :: Ord a => a -> a -> a
(/) :: Fractional a => a -> a -> a
sin :: Floating a => a -> a
show :: Show a => a -> String
elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

Tässä esimerkiksi funktion `(+)` parametrit voivat olla mitä tahansa tyyppiä `a` sillä ehdolla, että tyyppi `a` on numeerinen eli tyyppiluokan `Num` jäsen.

1.36 Luetellut tyypit

Luetellut eli algebralliset tyypit määrittelemme avainsanalla `data`. Annamme tyypin nimen yhtäsuuruusmerkin vasemmalla puolella ja tyypin alkioiden mahdolliset arvot (*konstruktorit*) oikealla puolella. Esimerkiksi seuraavassa tietotyyppi `Bool` voi saada toisen arvoista `False` (*epätosi*) tai `True` (*tosi*), ja tietotyyppi `Direction` (*ilmansuunta*) jonkin arvoista `N`, `S`, `E`, `W`, `NE`, `NW`, `SE` tai `SW`.

```
data Bool = False | True
data Direction = N | S | E | W | NE | NW | SE | SW
```

1.37 Korkeamman asteen tyypit

Korkeamman asteen tyypeillä on konstruktori ja parametreja.

```
data Point = Point Double Double
data RGB = RGB Double Double Double
```

Tässä tyypin `Point` arvoilla on konstruktori `Point` ja kaksi liukulukutyypin `Double` argumenttia. Tyypin `RGB` arvoilla on konstruktori `RGB` ja kolme tyypin

`Double` argumenttia. Tyypillä on usein sama nimi kuin sen ensimmäisellä konstruktorilla.

1.38 Konstruktorit funktioina

Kohtelemme konstruktoreita Haskell-kielessä funktioina, joilla on tyyppi. Konstruktorien ainoa tehtävä on muodostaa alkio, joka on annettua tyyppiä. Konstruktoreiden toteutus on siten niiden määrittelyssä, joten emme Haskell-kielessä luo taikka hävitä konstruktoreita erikseen. Seuraavassa esitämme konstruktorien `False` ja `Point` tyypit.

```
False :: Bool
Point :: Double -> Double -> Point
```

1.39 Kommentit lähdekoodissa

Haskell-kielessä voimme kirjoittaa ohjelmakoodiin kommentteja tavuviivaparin `--` jälkeen. Kääntäjä jättää tällöin kommenttimerkit ja sitä seuraavan tekstin huomiotta rivin loppuun saakka. Useamman rivin kommentit voimme ympäröidä merkkipareilla `{-` ja `-}`. Kommenttimerkintöjen avulla voimme myös piilottaa varsinaista ohjelmakoodia kääntäjältä näkymättömiin.

Haskell-kielen kääntäjäympäristöön kuuluu dokumentointiohjelma Haddock. Sen avulla voimme luoda automaattisesti ohjesivun kirjoittamallemme kirjastolle. Funktion ja tietotyypin määrittelyn yläpuolella oleva kommenttimerkintä

```
-- | Explanation what a function does
```

on ohje dokumentointiohjelmalle, jonka perusteella se yhdistää kommentin sitä vastaavaan määritelmään.

1.40 Merkitsevä sisennys

Kuten perustellusti monessa muussa nykyaikaisessa ohjelmointikielessä, myös Haskell-kielessä ohjelmakoodin sisennys on merkityksellinen ja siten pakollinen. Sisennys jäsentää muuttujamäärittelyjen näkyvyyden eri tasoille. Esimerkiksi `where`-rakenteen sisällä määritellyt muuttujat eivät näy päätasolla.

```
s = a + b
  where
    a = 5
    b = 7
```

Hyvä sisennysmäärä on esimerkiksi kaksi välilyöntiä. Sisennykseen ei pidä käyttää tabulaattoria.

Emme käy sisennyssääntöjä tässä tarkemmin lävitse, sillä ne noudattavat useimmin samaa päättelylogiikkaa ihmisen kanssa.

Toisinaan saatamme ohjeista huolimatta kirjoittaa määritelmiä esimerkiksi vuorovaikutteisessa tulkissa yhdelle riville käyttäen aaltosulkeita `{}` ja puolipistettä `;`.

```
s = a + b where { a = 5; b = 7 }
```

1.41 Vuorovaikutteisen tulkin komentoja

Tässä luvussa esittelimme joitakin standardikirjaston funktioita, joiden toimintaa kokeilimme Haskell-kielen vuorovaikutteisessa tulkissa. Vuorovaikutteinen tulkki `ghci` tarjoaa myös joukon omia komentoja, jotka eivät ole Haskell-kielen funktioita. Vuorovaikutteisen tulkin omat komennot alkavat kaksoispisteellä `:`.

Vuorovaikutteisen tulkin komentoluettelon saamme komennolla `:help`. Kuten useimmat komennot, se lyhenee alkukirjaimen mukaan muotoon `:h` tai `:?`.

Jos esimerkkiohjelmamme on tiedostossa `example.hs`, lataamme sen vuorovaikutteiseen tulkkiin komennolla `:load example`.

Tehtyämme muutoksia tiedostoon, lataamme tiedoston uudelleen komennolla `:reload`.

Komento `:browse` tulostaa kirjaston määrittelemät funktiot ja tietorakenteet. Esimerkiksi kattavan luettelon standardikirjaston `Prelude` funktioista saamme komennolla `:browse Prelude`.

Komennolla `:type` näemme annetun lausekkeen tyypin.

```
> :type 'c'
'c' :: Char
> :type (+)
(+) :: Num a => a -> a -> a
```

Komento `:info` tulostaa tietoa annetusta funktiosta tai tietorakenteesta.

```
> :info max
class Eq a => Ord a where
  max :: a -> a -> a
...
> :info Int
data Int = GHC.Types.I# GHC.Prim.Int#
                                    -- Defined in ‘GHC.Types’
instance Bounded Int                 -- Defined in ‘GHC.Enum’
instance Enum Int                    -- Defined in ‘GHC.Enum’
...
```

Muuttuja `it` palauttaa kulloinkin viimeksi lasketun lausekkeen arvon.

```
> 3 * 4
12
> it + 1
13
```

Komennolla `:!` suoritamme komentotulkin komennon.

```
> :! date
pe 4.1.2019 21.16.43 +0200
```

Useammalle riville jaetut määrittelyt voimme antaa komentoparin `:{` ja `:}` sisällä.

```
> :{
| s = a + b
|  where
|   a = 5
|   b = 7
| :}
> s
12
```

Olemme tämän kirjan tulosteessa käyttäneet komentokehotteena merkkiparia `"> "` ja useammalle riville jaettujen määrittelyiden kehotteena merkkiparia `"| "`. Käytetyt komentokehotteet voimme asettaa komennoilla `:set prompt` ja `:set prompt-cont`. Kun lisäämme komennot tiedostoon `~/.ghc/ghci.conf`, tulkki suorittaa ne automaattisesti käynnistyksen yhteydessä. Voimme asettaa komentokehotteille myös värimäärityksiä.

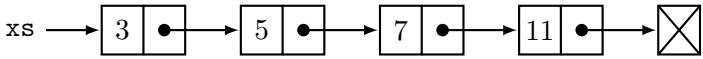
```
:set prompt  "\ESC[34m\STX> \ESC[m\STX"
:set prompt-cont "\ESC[34m\STX| \ESC[m\STX"
```

Vuorovaikutteisesta tulkista poistumme komennolla `:quit`.

Luku 2

Listat

Haskell-kielen listan käsite abstrahoi yksisuuntaisen linkitetyn listan toteutuksen alkioineen, linkkeineen ja loppumerkkeineen (kuva 1).



Kuva 1. Lista `xs = [3,5,7,11]` yksisuuntaisena linkitettynä listana.

Haskell-kielessä lista on yksinkertainen mutta monikäyttöinen tietorakenne.

Kuten kuvasta 1 havaitsemme, tapahtuu listan käsittely alkaen ensimmäisestä alkiosta, ja listan viimeisen alkion saavutamme vasta käytyämme listan kaikki muut alkiot lävitse.

Kun käsittelemme listoja, emme muuta yksittäisten alkioiden sisältöä, vaan kopioimme olemassaolevia listoja tai listan osia, joita vaihdamme, liitämme ja poistamme haluamamme tuloksen saavuttamiseksi.

2.1 Listan määrittely

Määrittelemme listat muodollisesti monimuotoisena rekursiivisena tietotyyppinä. Lista on joko tyhjä lista `[]` tai minkä tahansa tyypin `a` alkio liitettynä

listakonstruktorilla vastaavan tyypin listaan.

```
data [a] = [] | a : [a]
```

Äärelliset listat rakentuvat siten alkaen tyhjästä listasta `[]`. Listat ovat oikealle assosioivia eli ne kasvavat oikealta vasemmalle. Listan pää sijaitsee vasemmalla ja listan häntä oikealla.

Listan häntä on lista, kun taas listan pää on alkio.

```
> xs = [3,5,7,11]
> head xs
3
> tail xs
[5,7,11]
> head (tail xs)
5
> tail (tail xs)
[7,11]
```

Lista `[]` on tyhjä lista, lista `[5]` yhden alkion muodostama lista ja luku `5` listan alkio. Listakonstruktorifunktiolla `(:)` emme voi liittää toisiinsa kahta listaa, vaan ensimmäisen argumentin tulee olla listan alkio.

```
> [5] : [11]
"ERROR: Non type-variable argument in the constraint."
```

Listat voivat olla myös listojen muodostamia listoja. Listojen muodostamien listojen alkiot ovat listoja.

```
> [5] : [[11]]
[[5],[11]]
```

2.2 Listakonstruktorifunktio `(:)`

Operaattori `(:)` on listakonstruktorifunktio ja kutsumme sitä englanninkielisellä nimellä *cons*. Listakonstruktorifunktion avulla liitämme alkion listan alkuun.

```
> 3:(5:(7:(11:[])))
```

```
[3,5,7,11]
> 3:5:7:11:[]
[3,5,7,11]
> :info (:)
data [] a = ... | a : [a]   -- Defined in ‘GHC.Types’
infixr 5 :
```

Listakonstruktorifunktio `(:)` vaatii ensimmäisenä argumenttinaan alkion ja toisena argumenttinaan listan.

```
> 3 : [5,7]
[3,5,7]
> 5 : [11]
[5,11]
```

Listakonstruktorifunktion `(:)` tyyppi on `a -> [a] -> [a]`.

```
> :type (:)
(:) :: a -> [a] -> [a]
```

2.3 Funktio `(++)`

Funktio `(++)` saa argumentteinaan kaksi listaa. Se palauttaa listan, jossa argumentteina annetut listat on yhdistetty. Funktion `(++)` tyyppi on `[a] -> [a] -> [a]`.

```
> [2,3] ++ [5,7,11]
[2,3,5,7,11]
> [5] ++ [11]
[5,11]
> :type (++)
(++) :: [a] -> [a] -> [a]
```

Funktiot `(:)` ja `(++)` toimivat myös prefix-muodossa.

```
> (:) 1 []
[1]
> (:) 2 [3,4]
[2,3,4]
```

```
> (++) [3,4] [5,6]
[3,4,5,6]
```

2.4 Merkkijonot listoina

Haskell-kielessä merkkijonot ovat merkkien muodostamia listoja.

```
type String = [Char]
```

Yksittäisen merkin (tyyppiä `Char`) kirjoitamme yksinkertaisiin lainausmerkkeihin (`'F'`) ja merkkijonon (tyyppiä `String`) tuplalainausmerkkeihin (`"Frigoris"`). Kaksi merkkijonoa liitämme yhteen listat yhdistävällä operaattorilla `(++)`.

```
> :t 'F'
'F' :: Char
> :t "Frigoris"
"Frigoris" :: [Char]
> ['M','e','d','i','i']
"Medii"
> 'S':'m':'y':"thii"
"Smythii"
> "Imbr" ++ "ium"
"Imbrium"
```

Kuten muidenkin listojen kohdalla, funktio `(++)` vaatii argumentteinaan kaksi listaa ja funktio `(:)` alkion ja listan.

```
> 'M' ++ "ortis"
"ERROR: Couldn't match type ‘[Char]’ with ‘Char’."
> 'O':'d':'i':'i'
"ERROR: Couldn't match type ‘[Char]’ with ‘Char’."
```

2.5 Sovittaminen parametrimuotoon

Esimakua Haskell-kielelle ominaisesta ilmiöstä, argumentin sovittamisesta parametrimuotoon, saamme, kun listakonstruktorioperaattorin `(:)` avulla pu-

ramme listan kahdeksi muuttujaksi, pääksi ja hännäksi.

```
> x:xs = [1,2,3,4]
> x
1
> xs
[2,3,4]
```

2.6 Listamuodostimet

Nimitämme *listamuodostimeksi* listarakennetta, joka sisältää kuvauksen listan alkioiden ulkomuodosta, alkioiden tuottamiseen käytettävän algoritmin ja mahdollisesti säännöt tiettyjen alkioiden suodattamiseen saadusta tuloslistasta.

Alkioiden `x` muodostaman listan, jossa `x` on joukosta `[1..5]`, saamme listamuodostimella

```
> [x | x <- [1..5]]
[1,2,3,4,5]
```

Nimitämme lausetta `x <- [1..5]` listamuodostimen *generaattoriksi*.

Sellaisten alkioiden muodostaman listan, jossa alkiot ovat muotoa `[x]`, saamme listamuodostimella

```
> [[x] | x <- [1..5]]
[[1],[2],[3],[4],[5]]
```

Peräkkäisten lukujen muodostamien lukuparien muodostaman listan saamme listamuodostimella

```
> [(x,x+1) | x <- [1..5]]
[(1,2),(2,3),(3,4),(4,5),(5,6)]
```

Muotoa `Just x` olevien alkioden listan saamme listamuodostimella

```
> [Just x | x <- [1..5]]
[Just 1,Just 2,Just 3,Just 4,Just 5]
```

Listamuodostin voi myös sisältää *suotimia*, jotka rajoittavat tulosjoukkoa. Seuraavassa funktio `odd` toimii listamuodostimen suotimena rajoittaen tulosjoukon parittomiin lukuihin.

```
> [x | x <- [1..10], odd x]
[1,3,5,7,9]
```

Sekä generaattoreita että suotimia voi olla useita. Generaattoreista jäljempi toimii silmukkana edeltävän sisällä.

```
> [(x,y) | x <- [1..3], y <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```

Kaikki `(x,y)`-parit, joissa `x` on pariton ja joukosta `[1..5]` sekä `y` parillinen ja joukosta `[4..8]` saamme listamuodostimella

```
> [(x,y) | x <- [1..5], y <- [4..8], odd x, even y]
[(1,4),(1,6),(1,8),(3,4),(3,6),(3,8),(5,4),(5,6),(5,8)]
```

2.7 Funktio `concat`

Standardikirjaston funktio `concat` muuntaa listojen listan listaksi.

```
> concat [[1,2],[3,4],[5]]
[1,2,3,4,5]
> concat ["Ma","re Ma","rgin","is"]
"Mare Marginis"
```

Listojen lisäksi funktio `concat` toimii myös muille tyyppiluokan `Foldable` jäsenille. Listojen tapauksessa tyyppimuuttuja `t` saisi arvon `t = []` ja funktio `concat` tyyppiallekirjoituksen `concat :: [[a]] -> [a]`.

```
> :t concat
concat :: Foldable t => t [a] -> [a]
```

2.8 Funktio `concatMap`

Funktiokutsu `concatMap f xs` kuvaa funktion `f` listalle `xs` muuntaen tuloksena saadun alilistojen listan peräkkäiseksi listaksi. Funktio `concatMap` palauttaa näin ollen yhdellä kertaa sen mitä funktiot `map` ja `concat` peräjälkeen kutsuttuina palauttaisivat.

```
> replicate 3 [1,2]
[[1,2],[1,2],[1,2]]
> map (replicate 3) [1,2]
[[1,1,1],[2,2,2]]
> concatMap (replicate 3) [1,2]
[1,1,1,2,2,2]
```

Voimme esimerkiksi määritellä funktiokutsun `head1 xs`, joka palauttaa listan `xs` pään yhden alkion listana silloin, kun lista `xs` ei ole tyhjä, ja tyhjän listan silloin, kun `xs` on tyhjä lista. Nyt funktiokutsu `concatMap head1 xs` kuvaa funktion `head1` listalle `xs` ja yhdistää syntyneet alilistat.

```
> head1 xs = if (null xs) then [] else [head xs]
> map head1 ["Lacus","","Odii"]
["L","","O"]
> concatMap head1 ["Lacus","","Odii"]
"LO"
```

Funktion `concatMap` tyyppi on `(a -> [b]) -> t a -> [b]`, sillä ehdolla, että `t` kuuluu tyyppiluokkaan `Foldable`. Esimerkiksi listat ovat tyyppiluokan `Foldable` jäsen.

```
> :t concatMap
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

Luku 3

Funktion määrittely ja kutsumuodot

3.1 Funktion määrittely

Haskell-ohjelma koostuu funktiomäärittelyistä, joita kutsumme funktion *kutsumuodon* avulla.

Haskell-kielessä määrittelemme funktion yhtälönä, jossa yhtäsuuruusmerkin vasemmalla puolella ovat funktionimi ja parametrit sekä oikealla puolella funktion palautusarvo. Aakkosnumeeriset funktionimet ja parametrit kirjoitamme pienellä alkukirjaimella.

Esimerkiksi funktiomäärittelyssä

```
double x = 2 * x
```

`double` on funktionimi, `x` on funktion parametri ja `2 * x` on lauseke funktion palautusarvon laskemiseksi.

Kun kutsumme funktiota `double` argumentilla 3, saamme funktion palautusarvoksi 6.

```
> double 3
6
```

Yksinkertaisin muoto funktiomäärittelystä on vakiofunktio, jossa funktiolla ei ole lainkaan parametreja. Määrittelemme seuraavassa vakiofunktiot `i`, `s` ja `ls`.

```
i = 2
s = "Serenitatis"
ls = [1,2,3,4]
```

Funktion kutsumuoto vakiofunktion tapauksessa on yksinkertaisesti funktion nimi ilman argumentteja.

```
> i = 2
> i
2
```

Parametreja sisältävistä funktioista määrittelemme esimerkkinä yhden parametrin funktion `f` ja kahden parametrin funktion `g` standardikirjaston kerto- ja yhteenlaskufunktioiden `(*)` ja `(+)` avulla.

```
f x = x * x
g x y = x * x + y
```

Voimme jakaa funktiomäärittelyn pienempiin osiin esimerkiksi avainsanalla `where` (*missä*).

```
f = a + b
  where
    a = c + 2
    b = c + 4
    c = 3
```

Toinen vaihtoehto funktiomäärittelyn jäsentelyyn on `let` … `in` (*olkoon … -ssa*) -rakenne.

```
f = let
    a = c + 2
    b = c + 4
    c = 3
  in a + b
```

3.2 Parametrien näkyvyysalueet

Määrittelemme muuttujat `x`, `f`, `g`, `h`, `k`, `p`, `i` ja `j`. Niiden näkyvyysalue on tiedoston päätaso. Ne näkyvät kaikkialla tiedostossa, emmekä voi päätasolla määritellä niitä uudelleen.

```
x = 3
f = x
g x = x + 1
h = x + g x
k = let
    x = 5
    g = 10
  in x + g
p x = x + x
  where
    x = 5
i = g 1
j = p 2
```

Tässä vakiofunktiolla `f` ei ole parametreja. Se saa arvonsa päätason määrittelystä `x = 3`. Muuttujan `f` arvoksi tulee siten 3.

Funktion `g` parametri `x` näkyy ainoastaan funktiomäärittelyn oikealla puolella. Siellä se peittää päätason määrittelyn `x = 3`. Funktio `g` saa siten arvon `x + 1` riippuen siitä mikä on funktion `g` parametri `x`.

Funktiolla `h` ei ole parametreja. Funktion määrittelyssä esiintyvä muuttuja `x` on siksi päätason määrittely `x = 3`. Funktio kutsuu funktiota `g` päätason arvolla `x = 3` eli kutsumuodossa `g 3`, joka saa arvon $3 + 1 = 4$. Muuttuja `h` saa siten arvon $3 + 4 = 7$.

Funktio `k` sisältää `let … in` -rakenteen. Sen sisällä määritellyt muuttujat näkyvät funktiomäärittelyn oikealla puolella, mutta eivät näy päätasolla. Nämä määrittelyt peittävät päätason muuttujat. Muuttuja `k` saa siten arvon $5 + 10 = 15$.

Funktio `p` sisältää `where`-rakenteen. Siinä määrittely `x = 5` peittää sekä parametrin `x` että päätason määrittelyn `x = 3`. Muuttuja `p` saa siten arvon

$5 + 5 = 10$.

Funktio `i` kutsuu funktiota `g` arvolla 1, ja saa siten arvon 2.

Funktio `j` kutsuu funktiota `p` arvolla 2, mutta koska funktion `p` sisäinen määrittely peittää parametrin, ei funktion `p` argumentilla ole vaikutusta tulokseen. Funktion `i` arvo on siten $5 + 5 = 10$.

Muuttujien `f`, `h`, `i`, `j`, `k` ja `x` arvot ovat nyt tietueeksi koottuna

```
> (f,h,i,j,k,x)
(3,7,2,10,15,3)
```

3.3 Argumenttien sovitus parametreihin

Haskell-ohjelma koostuu päätasolla funktiomäärittelyistä, joissa sidomme joukon muuttujanimiä määrittelyihinsä. Kutsuimme edellä vakiofunktioita niiden nimillä ilman argumentteja. Haskell-kääntäjä etsii tällöin kutsumuotoa eli funktion nimeä vastaavan määrittelyn ja sitoo funktiokutsun tähän määrittelyyn. Funktion määrittely voi sijaita missä tahansa tunnetussa nimiavaruudessa. Määrittelyn tulee kuitenkin olla yksikäsitteinen. Tämän perusteella ymmärrämme, että Haskell-kielessä sama muuttuja- ja funktionimi voi olla määritelty vain yhteen kertaan. Haskell-kielen muuttujille on siten tyypillistä, että ne eivät voi muuttua.

Vakiofunktiota tavanomaisempia ovat funktiokutsut, joissa annamme funktiolle argumentteja. Esimerkiksi funktiokutsut `f 3` ja `g 4 5` ovat tällaisia.

Tarkkaan ottaen jokaisella Haskell-funktiolla voi olla korkeintaan yksi parametri. Ymmärrämme tämän myöhemmin parametrien ketjutuksen yhteydessä, jolloin toteamme muun muassa, että lauseke `g x y` suluttuu laskujärjestysmäärittelyiden perusteella muotoon `((g x) y)`. Tarkastelemme tässä kuitenkin lauseketta `g x y` kahden parametrin funktiona.

Haskell-kielelle tyypillinen ominaisuus on, että määrittelemme funktiot sellaisissa parametrimuodoissa, joissa parametrimuoto määrää funktion käyttäytymisen. Parametrimuoto toimii tällöin muottina, johon Haskell-ohjelma sovittaa funktiokutsun argumentteja. Ymmärrämme tässä *muuttujanimiin sovittamisena* ilmiön, jossa funktion kutsumuodosta sovitamme sekä funktion

nimen, argumenttien muodon että argumenttien arvon vastaaviin parametreihin määrittelyssä.

Toisin kuin funktionimiin, joiden tulee olla yksikäsitteisiä, parametreihin sovittaminen tapahtuu lähdekoodissa esitellyssä järjestyksessä, eli ainoastaan lähdekoodissa ylinnä sijaitseva sopiva parametrimuoto tulee hyväksytyksi. Mikäli mikään parametrimuodoista ei tule hyväksytyksi, seurauksena on virhetilanne.

Määrittelemme seuraavassa funktion `head` parametrimuodolla `x:xs`.

```
> head (x:xs) = x
```

Kun nyt kutsumme funktiota argumentilla `[]`, ei argumentti sovi parametrimuotoon `x:xs`.

```
> head []
"ERROR: Non-exhaustive patterns in function `head`."
```

Yksinkertainen tapaus funktion argumentin sovittamisesta on argumentin sovittaminen vakioarvoon.

```
f 1 = "one"
f 2 = "two"
f 3 = "three"
f x = "other: " ++ show x
```

Tässä parametri `x` on vapaa muuttuja, johon kaikki arvot sopivat. Jos funktion `f` argumentti ei siis ole mikään luvuista 1, 2 tai 3, se sopii vapaaseen muuttujaan `x`.

Kuvaamalla nyt funktion `f` listan `[1..6]` alkioille saamme

```
> map f [1..6]
["one","two","three","other: 4","other: 5","other: 6"]
```

3.4 Luetellut tyypit

Määrittelemme *algebralliset tyypit* avainsanalla `data`. Luettelemme määrittelyssä tyypin mahdolliset arvot, josta johtuen kutsumme tällaisia tyyppejä

usein *luetelluiksi tyypeiksi*. Esimerkiksi tyyppi `Bool` on lueteltu tyyppi, joka voi saada toisen arvoista `False` tai `True`.

```
data Bool = False | True
```

Argumentin sovituksen avulla voimme määritellä tietotyypille `Bool` funktion `show` parametrien arvoilla `True` ja `False`.

```
show True  = "True"
show False = "False"
```

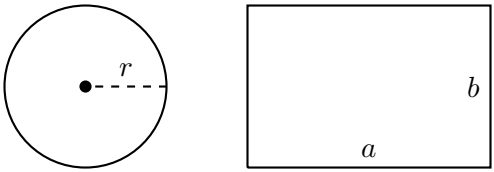
Saamme

```
> map show [True,False]
["True","False"]
```

3.5 Sovitus konstruktoriin ja purku muuttujiin

Olkoon tyyppi `Shape` lueteltu tyyppi, jonka alkioita ovat ympyrät `Circle r` ja suorakulmiot `Rectangle a b` (kuva 2).

```
data Shape = Circle Float | Rectangle Float Float
```



Kuva 2. Ympyrä `Circle r` ja suorakulmio `Rectangle a b`.

Konstruktori `Circle` on yhden parametrin funktio, konstruktori `Rectangle` kahden. Molemmat palauttavat arvon, jonka tyyppi on `Shape`.

```
> :t Circle
Circle :: Float -> Shape
> :t Rectangle
Rectangle :: Float -> Float -> Shape
```

Määrittelemme funktion `area`, joka laskee ympyrän ja suorakulmion pinta-alan

```
area (Circle r) = pi * r * r
area (Rectangle a b) = a * b
```

Funktio saa parametrinaan arvon tyyppiä `Shape` ja palauttaa liukuluvun tyyppiä `Float`.

```
> :t area
area :: Shape -> Float
```

Voimme määritellä tyyppiä `Shape` olevan ympyrän `c` ja suorakulmion `d`.

```
c = Circle 10
d = Rectangle 2 5
```

Funktio `area` purkaa nyt tyypin `Shape` arvon parametrimuotoon `Circle r` tai `Rectangle a b` riippuen siitä kumpaan muotoon muuttujat `c` ja `d` sopivat. Muuttuja `c = Circle 10` sopii parametrimuotoon `Circle r`, joten vapaa muuttuja `r` saa arvon 10, ja saamme funktion arvon lausekkeella `pi * r * r`. Vastaavasti muuttuja `d = Rectangle 2 5` sopii parametrimuotoon `Rectangle a b`, jolloin saamme funktion arvon lausekkeella `a * b`. Tällöin vapaa muuttuja `a` saa arvon 2, vapaa muuttuja `b` arvon 5, ja lauseke `a * b` arvon $2 \times 5 = 10$.

```
> area c
314.15927
> area d
10.0
```

3.6 Listan purku listakonstruktorin avulla

Määrittelemme listoille funktion `f`, joka purkaa listakonstruktorifunktion `(:)` avulla listan muuttujiin `x` ja `y` samaan tapaan kuin edellä funktio `area` purki tyypin `Shape` arvon parametrimuotoihin `Circle r` ja `Rectangle a b`.

```
> f ((:) x y) =
  "head = " ++ show x ++ ", " ++
```

```
  "tail = " ++ show y
> putStrLn (f "Marginis")
head = 'M', tail = "arginis"
```

Listakonstruktori `(:)` esiintyy hyvin yleisesti Haskell-ohjelmissa parametrimuotoon sovituksen yhteydessä. Esitämme seuraavassa määrittelemämme funktion infix-muodossa.

```
> f (x:y) =
  "head = " ++ show x ++  ", " ++
  "tail = " ++ show y
> putStrLn (f "Humorum")
head = 'H', tail = "umorum"
```

Määrittelemme aiemmin esittelemämme listafunktiot `head` ja `tail` parametrimuotoon sovituksen avulla.

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


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

Halutessamme voimme merkitä käyttämätöntä vapaata muuttujaa alaviivalla `_`, joka on muuttujamäärittelyn erikoismerkki huomiotta jätettävälle muuttujalle.

```
head (x:_)  = x
tail (_:xs) = xs
```

Kaikki epätyhjät listat ovat muotoa `(x:xs)`. Funktiokutsussa `f "Humorum"` on `f` funktionimi ja argumentti `"Humorum"` merkkijono, eli muotoa `(x:xs)` oleva lista.

```
> (:) 'H' "umorum"
"Humorum"
```

Näin ollen argumentti sopii parametrimuodon arvokonstruktoriin `(:)`, jolloin lista `"Humorum"` puretaan vapaisiin muuttujiin `x` ja `xs`.

Laskujärjestyksessä funktiototeutus on etusijalla verrattuna operaattoriin `(:)`. Kutsumuodon määrittelyssä `f (x:xs)` tarvitsemme sulkumerkkejä

laskujärjestyksen osoittamiseen, sillä muutoin funktio `f` sitoisi ensimmäisen muuttujan `x` ja koko lauseke saisi muodon `((f x):xs)`.

3.7 Tietueen purku tietuekonstruktorin avulla

Jaoimme aiemmin listan `ts` kahtia funktion `splitAt` avulla.

```
> ts = [2,3,5,7,11]
> splitAt 2 ts
([2,3],[5,7,11])
```

Tietuekonstruktorin `(,)` avulla voimme purkaa syntyneen listaparin `p` muuttujiin `a` ja `b`.

```
> p = splitAt 2 ts
> (a,b) = p
> a
[2,3]
> b
[5,7,11]
```

Sama toimii myös prefix-muodossa.

```
> (,) a b = splitAt 2 ts
> a
[2,3]
> b
[5,7,11]
```

3.8 Ehtolauseet parametrin sovituksessa

Sen lisäksi, että voimme pyrkiä sovittamaan argumenttia vakioarvoon tai arvokonstruktoriin, voimme sovittaa sitä ehtolauseeseen.

```
planet name d
  | d < 3.0   = name ++ ": small planet"
  | otherwise = name ++ ": giant planet"
```

Esimerkki tulostaa

```
> planet "Earth" 1.0
Earth: small planet
> planet "Saturn" 9.4
Saturn: giant planet
```

3.9 Funktiot `putStr` ja `putStrLn`

Funktiokutsu `putStr s` tulostaa merkkijonon `s` standarditulosvirtaan `stdout`. Funktio `putStrLn` tulostaa merkkijonon sekä rivinvaihdon. Päätteeltä käynnistettäessä standarditulosvirtaan `stdout` tulostaminen tarkoittaa tekstin tulostamista pääteikkunaan.

Syöte- ja tulostusfunktioiden tyypiksi on Haskell-kielessä määritelty `IO a`. Tyyppi `IO a` voisi ideaalimaailmassa olla määritelty seuraavasti:

```
data IO a = IO (Realworld -> (Realworld, a))
```

Funktiot `putStr` ja `putStrLn` saavat parametrinaan merkkijonon tyyppiä `String`. Ne palauttavat arvon tyyppiä `IO ()`.

```
> :t putStr
putStr :: String -> IO ()
```

Tyypin `()` ainoa alkio on tyhjä arvo `()`.

```
data () = ()
```

Jos funktioiden `putStr` ja `putStrLn` palautusarvon tyyppiä `IO ()` haluaisi sanallisesti kuvailla, se voisi olla: ”tyhjä arvo samalla standarditulosvirtaan tulostaen”.

3.10 Funktio `show`

Funktio `show` palauttaa annetun argumentin tekstuaalisen esityksen, mikäli sellainen on tyyppiluokassa määritelty eli mikäli tyypillä on tyyppiluokan `Show` instanssi.

```
> s = "g 2 3 = " ++ show (g 2 3)
> s
"g 2 3 = 7"
> putStrLn s
g 2 3 = 7
> show 7
"7"
```

Kun tyyppi `a` on tyyppiluokan `Show` jäsen, on funktion `show` tyyppi

```
> :t show
show :: Show a => a -> String
```

Funktiolla `g` ei ole tyyppiluokan `Show` instanssia.

```
> :t g
g :: Num a => a -> a -> a
> show g
"ERROR: No instance for (Show (a0 -> a0 -> a0))."
```

3.11 Funktio `print`

Standardikirjastossa on määriteltynä myös funktio `print`, joka vastaa likimain funktiota `putStrLn`, mutta tulostaa automaattisen merkkijonomuunnoksen avulla jotakuinkin kaiken tulostettavissa olevan. Samalla se kuitenkin lisää esimerkiksi lainausmerkit merkkijonomuuttujiin. Funktio `print` soveltuu siksi välitulosteisiin ohjelman kehitysvaiheessa, kun taas varsinainen tulostus käy paremmin funktiolla `putStrLn`. Havaitsemme Haskell-kielen vuorovaikutteisen tulkin tulosteen vastaavan funktion `print` paluuarvoja.

```
> (g 2 3)
7
> print (g 2 3)
7
> print "Luxuriae"
"Luxuriae"
> putStrLn "Luxuriae"
```

Luxuriae

Funktion `print` parametri on minkä tahansa tyypin `a` arvo, sillä ehdolla että tyyppi `a` on tyyppiluokan `Show` jäsen, eli sille on määritelty tyyppiluokan `Show` instanssi. Funktion `putStrLn` parametri puolestaan on merkkijono tyyppiä `String`.

```
> :t print
print :: Show a => a -> IO ()
> :t putStrLn
putStrLn :: String -> IO ()
```

3.12 Pääohjelman `main`-funktio

Funktio `main` on erityisasemassa oleva funktio, jota järjestelmä kutsuu kun käynnistämme ohjelman. Tallennamme oheisen ohjelman tiedostoon `main.hs`.

```
f x = x * x
g x y = x * x + y

main = do
  putStrLn ("f 3 = " ++ show (f 3))
  putStrLn ("g 2 3 = " ++ show (g 2 3))
```

Käynnistämme ohjelman komentoriviltä komennolla `runhaskell main.hs`.

```
$ runhaskell main.hs
f 3 = 9
g 2 3 = 7
```

Vaihtoehtoisesti voimme lukea ohjelmakoodin vuorovaikutteisen tulkin komennolla `:load` ja kutsua funktiota `main`. Vuorovaikutteinen tulkki antaa kirjastollemme automaattisesti nimen `Main`.

```
> :l main.hs
[1 of 1] Compiling Main         ( main.hs, interpreted )
Ok, modules loaded: Main.
> main
```

```
f 3 = 3
g 2 3 = 7
```

Vuorovaikutteisen tulkin komennolla `:browse` näemme määrittelemiemme funktioiden tyyppiallekirjoitukset.

```
> :browse Main
f :: Num a => a -> a
g :: Num a => a -> a -> a
main :: IO ()
```

3.13 Rekursion periaate

Tutustumme seuraavaksi algoritmin muodossa *rekursioon*, eli ongelmanratkaisumenetelmään, jossa funktion lopullinen arvo saadaan toistuvasti funktiota itseään kutsumalla. Funktionaalisissa kielissä rekursion merkitys on suuri ja kääntäjä osaa usein muuntaa paljon resursseja vaativan rekursiivisen tehtävän vähän resursseja vaativaksi iteratiiviseksi vastaavaksi.

Listakonstruktorin ja argumenttien sovituksen avulla voimme kirjoittaa seuraavan algoritmin:

```
pairs (x:y:zs) = (x,y) : pairs (y:zs)
pairs (y:zs) = []
```

Algoritmi tuottaa peräkkäiset lukuparit annetusta listasta.

```
> print (pairs [1..5])
[(1,2),(2,3),(3,4),(4,5)]
```

Muistamme, että lista `[1..5]` on muotoa `1:2:3:4:5:[]`.

Tässä parametrimuoto `x:y:zs` vastaa ainoastaan sellaisia listoja, joiden alussa on kaksi listan alkiota `x` ja `y` ja näiden perässä lista `zs`. Kutsuttaessa funktiota argumentilla `[1..5]` tämä ehto täyttyy, sillä listan `1:2:3:4:5:[]` kohdalla on voimassa `x:y:zs = 1:2:(3:4:5:[])`. Muuttujien arvot ovat siten `x = 1`, `y = 2` ja `zs = 3:4:5:[]`.

Funktion palautusarvo näillä muuttujan arvoilla on pari `(1,2)` jota seuraa listakonstruktori ja tämän jälkeen uusi funktiokutsu `pairs`, mutta nyt argu-

mentilla `y:zs` eli `2:(3:4:5:[])` joka vastaa listaa `[2..5]`. Tämä funktiokutsu luonnollisesti palauttaa parin `(2,3)`, listakonstruktorin ja jälleen uuden funktiokutsun.

Jatkamme näin kunnes argumentin muoto ei enää vastaa kutsumuotoa `x:y:zs`. Tällöin algoritmi siirtyy tarkastelemaan seuraavaa kutsumuotoa `y:zs`. Listan `[1..5]` kohdalla näin käy ainoastaan listan lopussa, kun jäljellä ovat enää alkiot `5` ja `[]`. Saamme tällöin funktion toisen kutsumuodon mukaisesti palautusarvona tyhjän listan `[]`, ja voimme lähteä kokoamaan koko funktion palautusarvoa listakonstruktorien avulla. Päädymme muotoon `(1,2):(2,3):(3,4):(4,5):[]` joka vastaa tuloslistaa `[(1,2),(2,3),(3,4),(4,5)]`.

Muistamme, että Haskell-kielessä parametrimuotojen esittelyjärjestyksellä on merkitystä.

Edellisen funktion kutsumuotojen määritteleminen päinvastaisessa järjestyksessä johtaa hyvin erilaiseen tulokseen, sillä kaikki toisen kutsumuodon mukaiset argumentit toteuttavat myös ensimmäisen kutsumuodon, eikä rekursio näin ollen pääse edes alkuun ennen kuin se on jo ohitse.

```
> :{
| pairs' (y:zs) = []
| pairs' (x:y:zs) = (x,y) : pairs' (y:zs)
| :}
> print (pairs' [1..5])
[]
```

3.14 Rekursiivisia funktioita

Määrittelemme nyt itse standardikirjaston funktiot `length`, `sum` ja `product` rekursion avulla.

```
length [] = 0
length (x:xs) = 1 + length xs

sum [] = 0
sum (x:xs) = x + sum xs
```

```
product [] = 1
product (x:xs) = x * product xs
```

Funktiot `factorial` ja `(++)` määrittelisimme rekursion avulla seuraavasti:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)


[] ++ ys     = ys
(x:xs) ++ ys = x : xs ++ ys
```

Rekursiiviset funktiot voivat myös olla päättymättömiä.

```
ones = 1 : ones
numsFrom n = n : numsFrom (n + 1)
squares = map (^ 2) (numsFrom 0)
```

Saamme nyt esimerkiksi

```
> take 6 ones
[1,1,1,1,1,1]
> take 5 (numsFrom 5)
[5,6,7,8,9]
> take 7 squares
[0,1,4,9,16,25,36]
```

Luku 4

Tyypillisiä funktiorakenteita

4.1 Yhdistetty funktio

Voimme kirjoittaa funktion `f (g (x))` muodossa `(f . g) x`. Nimitämme funktiota `f . g` *yhdistetyksi funktioksi*. Määrittelemme yhdistetyn funktion operaation `(.)` seuraavasti:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

Yhtäpitävän vaihtoehtoisen määrittelyn saamme myös nimettömänä funktiona

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

4.2 Funktio `map`

Funktion `map` tehtävä on soveltaa annettua *kuvausfunktiota* listan alkioihin. Esittelemme funktion `map` toteutuksen listamuodostimilla ja vaihtoehtoisesti rekursion avulla.

```
map :: (a -> b) -> [a] -> [b]
```

```
map f xs = [f x | x <- xs]

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

4.3 Funktio `filter`

Suodatusfunktio `filter` kokoaa listasta annetun ehdon täyttävät alkiot. Myös tämän funktion voimme toteuttaa listamuodostimilla tai rekursiolla.

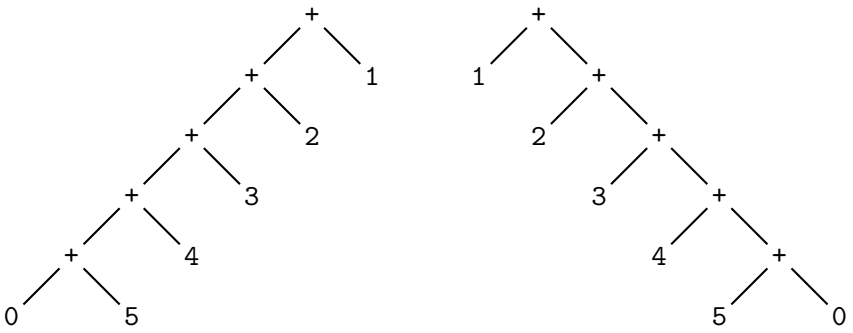
```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

4.4 Funktiot `foldr` ja `foldl`

Määrittelemme seuraavaksi niin sanotut *taittelufunktiot* `foldl` ja `foldr`. Nimet `foldl` (*fold-left*, ”taittele vasemmalle”) ja `foldr` (*fold-right*, ”taittele oikealle”) noudattavat kuvan 3 periaatetta. Sanomme *taiteltavaksi* (tyyppiluokka `Foldable`) tietorakennetta, joka on mahdolllista järjestää läpikäyntiä varten kuvan esittämään muotoon. Käytännössä taittelu on rekursiivinen operaatio, joka kohdistuu kerrallaan kahteen alkioon.

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

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

Tässä funktio `f` on taitteluoperaatio ja alkio `a` operaation alkuarvo eli nolla-alkio. Kuvassa 3 olemme esittäneet graafisesti lausekkeet `foldl (+) 0 [1..5]` ja `foldr (+) 0 [1..5]`.

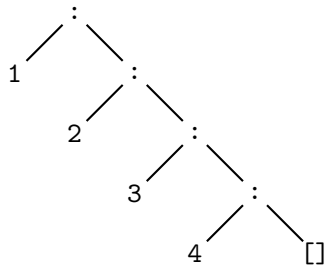


Kuva 3. Lausekkeet `foldl (+) 0 [1..5]` ja `foldr (+) 0 [1..5]`.

Molempien lausekkeiden arvoksi saamme 15.

```
> foldl (+) 0 [1..5]
15
> foldr (+) 0 [1..5]
15
```

Funktiot `foldr` ja `foldl` eivät aina toimi symmetrisesti. Esimerkiksi listakonstruktorin toinen parametri on alkio ja toinen lista, joten voimme taitella listan listakonstruktorin suhteen ainoastaan oikealle (kuva 4).



Kuva 4. Lauseke `foldr (:) [] [1..4]`.

Saamme esimerkiksi

```
> foldr (:) [] [1..4]
[1,2,3,4]
```

Voisimme määritellä listojen yhteydessä käyttämämme funktion `concat` funktion `foldr` avulla parametrittomassa muodossa varsin ytimekkäästi.

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

4.5 Funktiot `foldr1` ja `foldl1`

Voimme määritellä funktioista `foldl` ja `foldr` helppokäyttöisemmät muodot `foldr1` ja `foldl1`, jotka käyttävät listan ensimmäistä alkiota alkuarvona.

```
foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []      =  error "ERROR: empty list."

foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
foldr1 _ []      =  error "ERROR: empty list."
```

Saamme nyt

```
> foldl1 (+) [1,2,3,4]
10
> foldr1 (+) [1,2,3,4]
10
```

Havaitsemme, että funktion `foldr1` parametri `f` on funktio tyyppiä `a -> a -> a` ja lista `x:xs` tyyppiä `[a]`, joten emme voi ajatella käyttävämme listakonstruktorifunktiota `(:) :: a -> [a] -> [a]` funktiona `f`.

4.6 Ketjutus

Olemme aiemmin puhuneet yhden, kahden ja kolmenkin parametrin funktioista. Täsmällisesti ottaen Haskell-kielessä ei ole kahden saati kolmen parametrin funktioita. On vain yhden parametrin funktio, joka muodostaa uuden funktion, joka voi saada uuden argumentin, joka edelleen voi saada uuden argumentin, ja niin edelleen, muodostaen näin ihmisen silmissä useamman parametrin funktion. Tästä prosessista, jota nimitämme *ketjutukseksi*, kertoo tyyppimäärittelyn nuoli `->`, joka symbolisoi matematiikan funktion kuvauksen merkintätapaa.

```
f :: Int -> Int -> Int
f x y = x + y

g :: Int -> (Int -> Int)
(g x) y = x + y
```

Tässä funktiot `f` ja `g` ovat samoja. Vuorovaikutteinen tulkki kertoo nyt funktioiden `f`, `f 1` ja `f 1 2` tyypit.

```
> :t f
f :: Int -> Int -> Int
> :t f 1
f 1 :: Int -> Int
> :t f 1 2
f 1 2 :: Int
```

Jos nyt määrittelemme funktion `f 1` uudella nimellä `plusOne`, saamme funktion, joka odotuksiemme mukaisesti palauttaa luvun, joka on yhden argumenttiaan suurempi.

```
> plusOne = f 1
> plusOne 2
3
```

Funktio voi saada parametrinaan vektorityyppisen muuttujan, esimerkiksi parin `(x,y)` tai kolmikon `(x,y,z)`. Mikäli kuitenkin mahdollista, pyrimme välttämään muuttujien tarpeetonta pakkaamista pareiksi ja kolmikoiksi.

```
f :: (Int,Int) -> Int
```

```
f (x,y) = x + y
```

4.7 Nimettömät funktiot

Nimetön funktio on muotoa `\x -> f x`. Luemme tämän muodossa "*lambda* x palauttaa arvon $f\ x$". Lambda, λ, on kreikkalaisten aakkosten yhdestoista kirjain, jota usein käytetään funktionaalisten kielten symbolina.

Funktio `\x -> 2 * x` on yhden parametrin nimetön funktio, kun taas funktio `\x y -> x + y` on kahden parametrin nimetön funktio.

Käytämme nimettömiä funktioita usein funktioiden `filter` ja `map` yhteydessä tai parametrien järjestystä vaihdettaessa.

```
> filter (\x -> 3 <= x && x <= 5) [1..9]
[3,4,5]
> filter (\x -> snd x `mod` 2 == 1 && fst x <= 'e')
         (zip ['a'..'z'] [1..])
[('a',1),('c',3),('e',5)]
> map (\x -> x `max` 0) [-1..2]
[0,0,1,2]
> map (\x -> x `mod` 2) [1..4]
[1,0,1,0]
> map (\x -> (x,x))  [1..3]
[(1,1),(2,2),(3,3)]
```

Samaan tapaan voimme siirtää funktiomäärittelyn parametrit yhtälön toiselle puolelle.

```
add x y = x + y
add x = \y -> x + y
add = \x y -> x + y
```

4.8 Funktion osittainen toteutus

Nimettömien funktioiden avulla voimme määritellä infix-muotoisille funktioille *osittaisen toteutuksen*.

```
(x +) = \y -> x + y
(+ y) = \x -> x + y
(+) = \x y -> x + y
```

Käytimme funktioiden osittaistoteutuksia aikaisemmin `map`-funktion yhteydessä.

```
> map (2 +) [1,2,3]
[3,4,5]
> map (3 *) [1,2,3]
[3,6,9]
> map (/ 2) [2,4,8]
[1.0,2.0,4.0]
> map ('O' :) ["blivionis","dii","rientale"]
["Oblivionis","Odii","Orientale"]
> map (++ "um") ["Humor","Undar","Vapor"]
["Humorum","Undarum","Vaporum"]
```

4.9 Tietueet ja ketjutus

Määrittelemme listan `t`, jonka alkioita ovat kokonaisluvun ja kirjaimen parit.

```
> t = zip [1..5] ['a'..]
> t
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
```

Kirjain on tyyppiä `Char`, ja voimme muuttaa sen merkkijonoksi tyyppiä `String` yksinkertaisesti listakonstruktorilla `[]`.

```
> ['a']
"a"
```

Määrittelemme funktion `g` palauttamaan parin alkiot, eli luvun ja kirjaimen. Kuvaamme koko listan `t` funktion `g` määrittelemään tulostusmuotoon.

```
> t = zip [1..5] ['a'..]
> g (a,b) = show a ++ " " ++ [b]
> g (1,'a')
"1 a"
```

```
> map g t
["1 a","2 b","3 c","4 d","5 e"]
```

Nimettömän funktion `\(a,b) -> f a b` avulla voimme kutsua muodossa `f a b` olevaa funktiota parametrilla `(a,b)`.

```
> f a b = show a ++ " " ++ [b]
> f 1 'a'
"1 a"
> map (\(a,b) -> f a b) t
["1 a","2 b","3 c","4 d","5 e"]
```

Samaan tarkoitukseen löydämme standardikirjastosta `Prelude` funktion `uncurry`.

```
> t = zip [1..5] ['a'..]
> f a b = show a ++ " " ++ [b]
> map (uncurry f) t
["1 a","2 b","3 c","4 d","5 e"]
```

Vastakkaissuuntaisen muunnoksen suorittaisimme funktiolla `curry`. Funktioiden `curry` ja `uncurry` tyyppimäärittelyt ovat

```
> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Funktio `curry` muuntaa ketjuttamattoman funktion ketjutetuksi funktioksi ja funktio `uncurry` ketjutetun funktion ketjuttamattomaksi. Ketjutettu funktio operoi kahdella parametrilla, ketjuttamaton yhdellä tietuetyypin parametrilla, jolla on kaksi alkiota.

Vastaavasti voimme muodostaa haluamamme parit suoraan funktiolla `zipWith`.

```
> zipWith f [1..5] ['a'..]
["1 a","2 b","3 c","4 d","5 e"]
```

4.10 Assosiaatio ja laskujärjestys

Voimme määritellä laskutoimitusten assosiaation ja suoritusjärjestyksen. Mikäli määrittelyt ovat annetut ja ne poikkeavat oletuksesta, saamme tiedot vuorovaikutteisen tulkin komennolla `:info`. Esimerkiksi yhteenlaskuoperaatiolle saamme

```
> :info (+)
class Num a where
  (+) :: a -> a -> a
infixl 6 +
```

Standardikirjasto määrittelee muun muassa seuraavat assosiaatio- ja laskujärjestyksen prioriteettiluokat:

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  :
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
```

Määrittelemme vasemmalle assosioivat laskutoimitukset komennolla `infixl` ja oikealle assosioivat komennolla `infixr`. Laskutoimitukset, joilta assosiaatio puuttuu, määrittelemme komennolla `infix`. Funktiokutsun sitovuus suhteessa parametriin kuuluu prioriteettiluokkaan 10. Tähän luokkaan emme voi määritellä omia funktioitamme. Tavallisesti emme määrittele omille funktioillemme erikseen prioriteettiluokkaa, jolloin ne oletuksena kuuluvat luokkaan `infixl 9`.

Laskujärjestysmäärittelyiden perusteella voimme suluttaa lausekkeet seuraavasti:

```
> 2 + 3 * 4 == 2 + (3 * 4)
True
> 3 - 2 - 1 == (3 - 2) - 1
True
```

```
> 2 ^ 2 ^ 3 == 2 ^ (2 ^ 3)
True
```

4.11 Funktiot `(**)`, `(^)` ja `(^^)`

Funktiot `(**)`, `(^)` ja `(^^)` ovat kaikki potenssiinkorotusoperaatioita.

Funktio `(**)` on määritelty yleisesti liukuluvuille.

```
> :type (**)
(**) :: Floating a => a -> a -> a
```

Funktio `(^)` on määritelty ei-negatiivisille kokonaispotensseille.

```
> :type (^)
(^) :: (Num a, Integral b) => a -> b -> a
```

Funktio `(^^)` on määritelty funktioiden `(^)`, `recip` ja `negate` avulla yleisesti kokonaispotensseille.

```
x ^^ n =  if n >= 0 then x^n else recip (x^(negate n))
> :type (^^)
(^^)  :: (Fractional a, Integral b) => a -> b -> a
```

Negatiivisten potenssien laskusääntö on matematiikasta tuttu

$$a^{-n} = \frac{1}{n}$$

4.12 Funktio `($)`

Määrittelemme funktion `($)`

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Suodatamme nyt listasta `[1..10]` parilliset alkiot funktiolla `filter even`, käännämme saadun listan ympäri ja palautamme kolme ensimmäistä alkiota.

```
> take 3 (reverse (filter even [1..10]))
[10,8,6]
```

Määrittelemme funktiot `f`, `g` ja `h`

```
f = take 3
g = reverse
h = filter even
k = f . g . h
```

Muistamme aiempaa kuinka funktiot `take`, `reverse` ja `filter` toimivat.

```
> f [1..10]
[1,2,3]
> g [1..10]
[10,9,8,7,6,5,4,3,2,1]
> h [1..10]
[2,4,6,8,10]
```

Näin ollen saamme

```
> take 3 (reverse (filter even [1..10]))
[10,8,6]
> (take 3 . reverse . filter even) [1..10]
[10,8,6]
> (f . g . h) [1..10]
[10,8,6]
> k [1..10]
[10,8,6]
```

Funktion `($)` avulla voimme nyt kirjoittaa

```
> k $ [1..10]
[10,8,6]
> f . g . h $ [1..10]
[10,8,6]
> take 3 . reverse . filter even $ [1..10]
[10,8,6]
```

Vaikka funktiomuoto `(f . g . h) [1..10]` on ulkoasultaan selvästi kauniimpi, on muoto `f . g . h $ [1..10]` haskellistien suosima.

Tarkastelemme vielä miten laskujärjestys ja assosiatiivisuus vaikuttaa funktion `g . h $ [1..10]` suoritukseen. Oikealta assosioiva funktio `($)` toimii kaikkein alhaisimmalla prioriteetilla 0 odotetusti. Sen sijaan funktion `(.)` kanssa samalla prioriteetilla määritettynä seurauksena on funktion tyyppivirhe.

```
> infixr 0 $; f $ x = f x
> g . h $ [1..10]
[10,8,6,4,2]
> infixr 9 $; f $ x = f x
> g . h $ [1..10]
"ERROR: Couldn't match type ‘[a0]’ with ‘a -> [a1]’."
```

Vasemmalta assosioivana funktio toimii samoin, mutta virheilmoitus on erilainen. Se kertoo laskujärjestyksen olevan mahdoton. Lausekkeen sulutus olisi tässä tapauksessa limittäinen.

```
> infixl 0 $; f $ x = f x
> g . h $ [1..10]
[10,8,6,4,2]
> infixl 9 $; f $ x = f x
> g . h $ [1..10]
"ERROR: Precedence parsing error."
```

Missään tapauksessa laskujärjestyksen ja assosiatiivisuuden määrittely ei kumoa erikseen merkittyä sulutusta. Samoin seuraavassa yhdistetyn funktion `g . h` sitominen funktion `m` määrittelyyn nostaa sen prioriteetin automaattisesti muiden määrittelyjen ohitse.

```
> m = g . h
> g (h $ [1..10])
[10,8,6,4,2]
> m $ [1..10]
[10,8,6,4,2]
```

Muistamme, kuinka käytimme funktion `(++)` osittaistoteutusta merkkijonoille.

```
> (++ "um") "Vapor"
"Vaporum"
```

Koska määrittelimme `f $ x = f x`, voimme nyt käyttää funktion `($)` osittaistoteutusta vastaavalla tavalla.

```
> ($ 3) (4 +)
7
```

Funktion `map` avulla laajennamme toimenpiteen listoille.

```
> map (++ "um") ["Vapor","Undar","Humor"]
["Vaporum","Undarum","Humorum"]
> map ($ 3) [(4 +),(3 *),(2 ^)]
[7,9,8]
```

Luku 5

Tietotyypit

5.1 Binäärijärjestelmä tietotyyppien perustana

Kutsumme *bitiksi* pienintä tiedon esittämisen yksikköä, joka voi sisältää yhden kahdesta mahdollisesta arvostaan. Voimme tulkita nämä arvot luvuiksi 0 ja 1, totuusarvoiksi `False` ja `True`, tai jollakin muulla haluamallamme tavalla. Tässä kappaleessa tulkitsemme ne merkeiksi `'0'` ja `'1'`.

Yhden bitin avulla voimme esittää kaksi mahdollista arvoa, kahden bitin avulla neljä, kolmen bitin avulla kahdeksan, neljän bitin avulla kuusitoista, ja niin edelleen. Bittien määrän ollessa `n`, saamme niiden avulla esitettävissä olevien arvojen määrän funktiolla `f n = 2 ^ n`.

```
> f n = 2 ^ n
> map f [0..8]
[1,2,4,8,16,32,64,128,256]
```

Ennen kuin lähdemme tarkastelemaan teoreettisella tasolla varsinaisia tietotyyppejä, syvennämme hieman Haskell-kielen osaamistamme tyypillisten funktioiden parissa ja tutustumme merkkijonoaritmetiikan avulla bittijonojen muodostumiseen.

Kuten kymmenjärjestelmä, myös binäärijärjestelmä soveltuu *laskemiseen*. Voimme pyytää Haskell-kieltä laskemaan määrittelemiemme binäärijär-

jestelmän merkkien `'0'` ja `'1'` avulla. Määrittelemme tätä varten rekursion avulla funktion `toBinary`.

```
toBinary 0 = "0"
toBinary 1 = "1"
toBinary r = toBinary (r `div` 2) ++
  if (r `mod` 2 == 1) then "1" else "0"
```

Nyt saamme

```
> map toBinary [0..15]
["0","1","10","11","100","101","110","111","1000","1001",
 "1010","1011","1100","1101","1110","1111"]
```

Vaihtoehtoisesti voimme käyttää kirjaston `Text.Printf` funktiota `printf`.

```
> import Text.Printf
> inBinary i = printf
  "The value of %d in binary is: %08b\n" i i
> inBinary 5
The value of 5 in binary is: 00000101
```

5.2 Totuusarvotyyppi `Bool`

Yhden bitin avulla toteutettavista tietotyypeistä mainitsimme edellä totuusarvotyypin `Bool`. Se voi saada toisen arvoista `False` tai `True`.

```
> :t False
False :: Bool
> :t True
True :: Bool
> not False
True
```

Totuusarvotyypin `Bool` konstruktorit `False` ja `True` ovat Haskell-kielen esitystapa arvoille. Kääntäjä muuntaa ne tietokoneen sisäiseen esitystapaan, biteiksi, ohjelman käännöksen yhteydessä. Voimme määritellä totuusarvotyypin `Bool` halutessamme itse seuraavasti:

```
data Bool = False | True
```

5.3 Merkkityyppi `Char`

Jo tietokoneiden alkuajoista lähtien bitit koottiin bittijonoiksi, jotka johdettiin rinnakkain niin sanottua *väylää* pitkin prosessoriin. Ensimmäisissä kaupallisissa tietokoneissa väylän leveys oli kahdeksan bittiä. Nykyaikaiset prosessorit ovat pääosin 64-bittisiä.

Tietokoneiden historia on vaikuttanut merkittävästi ohjelmointikielten tietotyyppien koon määräytymiseen. Kahdeksan bitin järjestelmä säilyi hallitsevana vuosikymmenet. Ehkä suurin vaikutus tällä on ollut tietokoneen käyttämään merkistöön. Tänä päivänä Haskell-kieli käyttää kohtuu sujuvasti laajempaa Unicode-merkistöä, mutta 8-bittisen (aluksi 7 bittiä ja tarkistusbitti) ASCII-merkistön vaikutus on yhä nähtävissä esimerkiksi kielen syntaksissa.

Kahdeksalla (tai seitsemällä) bitillä saamme esitettyä latinan suur- ja pienaakkoset.

```
> (['A'..'Z'],['a'..'z'])
("ABCDEFGHIJKLMNOPQRSTUVWXYZ",
 "abcdefghijklmnopqrstuvwxyz")
```

Näistä `'A'` on saanut ASCII-merkistössä järjestysluvun 65 ja `'a'` järjestysluvun 97. Bittitasolla muunnos latinan suuraakkosten ja pienaakkosten välillä onnistuu yhtä bittiä muuttamalla.

```
> ord 'A'
65
> ord 'a'
97
> inBinary (ord 'A')
The value of 65 in binary is: 01000001
> inBinary (ord 'a')
The value of 97 in binary is: 01100001
```

Yleinen merkin tietotyyppi Haskell-kielessä on `Char`. Tämä tietotyyppi pitää sisällään kaikki Unicode-merkistön merkit, myös sellaiset, jotka eivät kuulu

alkuperäiseen 7-bittiseen ASCII-merkistöön.

Kirjastossa `Data.Char` on määritelty funktio `isAscii`, joka palauttaa totuusarvon `TRUE`, mikäli merkki kuuluu ensimmäiseen 127 merkin joukkoon. Vastaavasti funktio `isLatin1` palauttaa totuusarvon `TRUE`, mikäli merkki kuuluu laajennettuun ASCII-merkistöön eli ensimmäiseen 255 merkin joukkoon.

```
isAscii :: Char -> Bool
isAscii c = c < '\128'

isLatin1 :: Char -> Bool
isLatin1 c = c <= '\255'
```

Esimerkiksi pohjoismaisten kielten kirjain `'æ'` on järjestysluvultaan 230, eikä siten kuulu 7-bittiseen ASCII-merkistöön, mutta kuuluu laajennettuun Latin1-merkistöön.

```
> import Data.Char
> ord 'æ'
230
> isAscii 'æ'
False
> isLatin1 'æ'
True
```

5.4 Funktiot `minBound` ja `maxBound`

Funktioiden `minBound` ja `maxBound` avulla saamme selville tyyppiluokkaan `Bounded` kuuluvan tyypin ala- ja ylärajan. Nämä funktiot eivät saa parametreja, mutta käyttäytyvät eri tavalla tyyppiluokan sisällä tyypistä riippuen. Koska funktioiden on tiedettävä tyyppinsä, ilmoitamme tyypin tyyppimäärittelymerkinnän `::` avulla

```
> minBound :: Bool
False
> maxBound :: Bool
True
> minBound :: Char
```

```
'\NUL'
> maxBound :: Char
'\1114111'
```

5.5 Kokonaislukutyyppi `Word`

Kahdeksalla bitillä voimme esittää positiiviset kokonaisluvut `0..255`. Vastaavasti kuudellatoista bitillä voimme esittää luvut `0..65535`. Löydämme tähän tarvittavat tietotyypit kirjastosta `Data.Word`.

Perustyyppi `Word` kuuluu standardikirjastoon `Prelude`. Haskell-kääntäjä valitsee sen koon automaattisesti prosessorin väylänleveyden mukaisesti.

```
> import Data.Word
> minBound :: (Word8,Word16,Word32,Word64)
(0,0,0,0)
> maxBound :: (Word8,Word16,Word32,Word64)
(255,65535,4294967295,18446744073709551615)
> maxBound :: Word
18446744073709551615
```

5.6 Kokonaislukutyyppi `Int`

Mikäli haluamme mukaan negatiiviset luvut, joudumme varaamaan yhden bitin etumerkille. Löydämme vastaavat tietotyypit kirjastosta `Data.Int`.

Perustyypin `Int` kääntäjä valitsee automaattisesti.

```
> import Data.Int
> minBound :: (Int8,Int16,Int32,Int64)
(-128,-32768,-2147483648,-9223372036854775808)
> maxBound :: (Int8,Int16,Int32,Int64)
(127,32767,2147483647,9223372036854775807)
> minBound :: Int
-9223372036854775808
> maxBound :: Int
```

```
9223372036854775807
```

5.7 Rajoittamaton kokonaislukutyyppi `Integer`

Koska listojen koko ei Haskell-kielessä ole rajoitettu, voimme (käytettävissä olevan muistin rajoissa) esittää niiden avulla miten suuria kokonaislukuja tahansa. Luonnollisesti lukuaritmetiikka on tällöin huomattavasti hitaampaa. Haskell-kielessä voimme tällaisia rajoittamattomia kokonaislukuja esittää tietotyypin `Integer` avulla.

```
> m = 9223372036854775807
> zipWith ($) [(+0),(+1),(+2)] (repeat (m :: Integer))
[ 9223372036854775807,
  9223372036854775808,
  9223372036854775809 ]
> zipWith ($) [(+0),(+1),(+2)] (repeat (m :: Int))
[ 9223372036854775807,
 -9223372036854775808,
 -9223372036854775807 ]
```

Esimerkissä näemme tietotyypin `Int` *ylivuodon*. Luonnollisesti tietotyypin yläraja on niin suuri, että käytännön sovelluksissa tietotyyppi `Int` on täysin riittävä.

5.8 Desimaalityypit `Float` ja `Double`

Desimaalilukujen esittäminen binäärijärjestelmässä ei ole täysin yksikäsitteistä. Käytännössä usein esiintyviä desimaalityyppejä ovat Haskell-kielessä ”yksinkertaisen” tarkkuuden liukulukutyyppi `Float` ja ”kaksinkertaisen” tarkkuuden liukulukutyyppi `Double`.

```
> tau = 6.28318530717958647692
> tau :: Float
6.2831855
> tau :: Double
6.283185307179586
```

5.9 Luetellut tyypit

Luetelluista tyypeistä olemme tähän mennessä jo tutustuneet tyyppiin `Bool`.

```
data Bool = False | True
```

Vastaavalla tavalla voimme muodostaa luetellun tyypin viikonpäivistä, shakkipelin nappuloista tai korttipelin korteista.

```
data Weekday =
  Monday | Tuesday | Wednesday | Thursday | Friday |
  Saturday | Sunday
data Piece =
  King | Queen | Bishop | Knight | Rook | Pawn
data CardValue =
  Two | Three | Four | Five | Six | Seven | Eight |
  Nine | Ten | Jack | Queen | King | Ace
```

Kuten muistamme, korttipakassa on neljä maata: risti, ruutu, hertta ja pata.

```
data Suit = Club | Diamond | Heart | Spade
```

Voimme nyt esittää korttipakan kortit tietotyypin `Card` avulla

```
data Card = Card Suit CardValue
```

Tässä vasemman puolen `Card` on tyypin nimi eli tyyppikonstuktori, oikean puolen `Card` kahden parametrin arvokonstruktori. Tyypit `Suit` ja `CardValue` ovat arvokonstruktorin `Card` parametreja. Arvokonstruktorin `Card` avulla saamme herttakasista, patakympistä ja ruutuseiskasta tyypin `Card` arvoja.

```
> :t Card
Card :: Suit -> CardValue -> Card
> :t Card Heart
Card Heart :: CardValue -> Card
> :t Card Heart Eight
Card Heart Eight :: Card
> :t Card Spade Ten
Card Spade Ten :: Card
> :t Card Diamond Seven
Card Diamond Seven :: Card
```

Erilaisia kortin arvoja on 13. Saamme ne laskemalla yhteen tyypin `CardValue` konstruktorien määrän. Tämän vuoksi nimitämme tyyppiä `CardValue` *summatyypiksi*.

Korttipakassa kortteja on 52. Saamme ne kertomalla keskenään tyyppien `Suit` ja `CardValue` konstruktorien lukumäärän. Nimitämme tyyppiä `Card` *tulotyypiksi*.

5.10 Abstraktit tyypit

Nimitämme *abstraktiksi tyypiksi* tyyppiä, joka sisältää tyyppimuuttujia. Haskell-kielessä kirjoitamme tyyppimuuttujat pienellä alkukirjaimella. Abstrakti tyyppi voi toteutuksessa olla mikä tahansa tyyppi.

Yksinkertainen esimerkki tyyppimuuttujasta on identiteettifunktio `id`. Funktio `id` palauttaa saman arvon, jonka se saa argumenttinaan.

```
> id x = x
> id 4
4
```

Funktiokutsun `id x` parametri `x` on abstraktia tyyppiä `a`, eli se voi olla mitä tahansa tyyppiä `a`. Funktion palautusarvo on samaa tyyppiä `a`.

```
> :type id
id :: a -> a
```

Käyttötarkoituksesta riippuen tyyppimuuttujalla voi olla rajoitteita. Esimerkiksi funktio `max` vaatii parametreiltaan järjestysominaisuutta (*Orderable*) ja funktio `(+)` vaatii, että molemmat argumentit ovat numeerisia (*Numeric*).

```
> :t max
max :: Ord a => a -> a -> a
> :t (+)
(+) :: Num a => a -> a -> a
```

Funktion `length` parametri on lista tyyppiä `[t1]`. Tyyppi `t1` voi olla mikä tahansa tyyppi. Funktion palautusarvo on numeerinen.

```
> :t length
```

```
length :: Num t => [t1] -> t
```

Funktion `length` palautusarvot ovat kokonaislukuja. Palautusarvon tyypin määritteleminen tyyppimuuttujan `t` avulla tarjoaa kuitenkin mahdollisuuden käyttää funktion palautusarvoa edelleen lausekkeissa, joiden arvo ei välttämättä ole kokonaisluku. Esimerkiksi jakolasku tiukentaa numeerisuuden vaatimusta osamäärämuotoiseksi (*Fractional*).

```
> length [1..9] / 2
4.5
> :t length [1..9] / 2
length [1..9] / 2 :: Fractional a => a
```

5.11 Rekursiiviset tyypit

Olemme aiemmin käyttäneet rekursiivisista tyypeistä muun muassa listoja. Määrittelemme listat nyt eri konstruktorinimiä käyttäen.

```
data List a = Nil | Cons a (List a)
```

Tässä vasemman puolen `List` on tyypin nimi eli tyyppikonstruktori. Yhdessä tyyppimuuttujan `a` kanssa se muodostaa tyypin. Tyyppimuuttuja `a` voi olla mikä tahansa tyyppi, kuten `Int`, `Float`, `Bool`, `Char` tai `String`. Esimerkiksi lausekkeen `Cons 'a' (Cons 'b' Nil)` tyyppi on `List Char`.

Tyyppi `List a` on rekursiivinen, sillä konstruktori `Cons` saa ensimmäisenä parametrina alkion tyyppiä `a` ja toisena parametrina tyyppiä `List a` olevan listan.

```
> :t Nil
Nil :: List a
> :t Cons 3 (Cons 2 Nil)
Cons 3 (Cons 2 Nil) :: Num a => List a
> :t Cons 'a' (Cons 'b' Nil)
Cons 'a' (Cons 'b' Nil) :: List Char
> :t Cons "A." (Cons "B." Nil)
Cons "A." (Cons "B." Nil) :: List [Char]
> :t Cons True (Cons False Nil)
```

```
Cons True (Cons False Nil) :: List Bool
```

5.12 Tyyppikonstruktori `Maybe`

Tyypin `Maybe a` mahdollisia arvoja ovat `Nothing` ja `Just a`. Jälleen tyyppimuuttuja `a` voi olla mikä tahansa tyyppi. Yhdessä konstruktorin `Just` kanssa se muodostaa alkion tyyppiä `Maybe a`.

```
data Maybe a = Nothing | Just a
```

Tyyppimuuttuja `a` voi olla esimerkiksi rajoitettu kokonaislukutyyppi `Int`.

```
> b = 2 :: Int
> Just b
Just 2
> :t Just b
Just b :: Maybe Int
```

Tyypin `Maybe a` yleinen käyttötarkoitus on valintatilanne, jossa arvo joko on olemassa (`Just x`) tai sitä ei ole olemassa (`Nothing`). Esimerkiksi etsittäessä alkiota listasta, standardikirjaston funktio `lookup` palauttaa arvon `Just x`, kun avain löytyy listasta ja arvon `Nothing`, kun avainta ei löydy listasta.

```
> lookup 'c' [('a',0),('b',1),('c',2)]
Just 2
> lookup 'd' [('a',0),('b',1),('c',2)]
Nothing
> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Funktion `lookup` tyyppiallekirjoitus kertoo, että avaimen täytyy kuulua tyyppiluokkaan `Eq`. Tämä on ymmärrettävä vaatimus, sillä vain olemassaoleva yhtäsuuruusoperaatio mahdollistaa avainten vertailun ja tuloksellisen etsintäoperaation.

5.13 Tyyppikonstruktori `Either`

Tyyppi `Either a b` on samankaltainen tyypin `Maybe a` kanssa.

```
data Either a b = Left a | Right b
```

Tyypillisessä käyttötarkoituksessa operaatio onnistuu palauttaen arvon tyyppiä `Right b` tai epäonnistuu palauttaen virheilmoituksen tyyppiä `Left a`. Vastaavasti lausekkeen jäsentäminen voi sieventää lauseketta `x` arvoon `Right y` tai jättää sen sieventämättä arvolla `Left x`.

Rakennelman kielellinen nerous piilee siinä, että, kuten suomen kielessä, myös englannin kielessä sana *right* tarkoittaa *oikeaa* vastakohtana sekä vasemmalle että väärälle.

Voimme esimerkiksi määritellä funktion `upper`, joka palauttaa tyypin `Maybe Char` arvon riippuen siitä kuuluuko argumentti kirjainjoukkoon `['A'..'Z']` vai ei. Konstruktoriin sovittamalla voimme purkaa argumentin takaisin muuttujaan, kuten esimerkin funktiossa `isIt`.

```
upper x
  | x `elem` ['A'..'Z']  =  Right x
  | otherwise  =  Left x

isIt (Right c) = "Yes: " ++ [c]
isIt (Left c)  = "No: "  ++ [c]

> upper 'T'
Right 'T'
> upper 't'
Left 't'
> isIt (upper 'T')
"Yes: T"
> isIt (upper 't')
"No: t"
```

5.14 Kielioppipuut

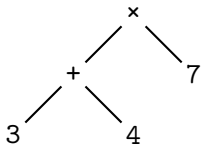
Kielioppipuut ovat esimerkki rekursiivisista tyypeistä. Kielioppipuun avulla voimme esimerkiksi jäsentää aritmeettisen lausekkeen. Seuraavassa kielioppipuu koostuu yhteenlaskuista (`Add`), kertolaskuista (`Mul`) ja kokonaislukuliteraaleista (`Lit`).

```
data Expr = Lit Int | Add Expr Expr | Mul Expr Expr
```

Määrittelemme tyypin `Expr` kielioppipuun lausekkeelle `(3 + 4) × 7`.

```
tree = Mul (Add (Lit 3) (Lit 4)) (Lit 7)
```

Esitämme saadun kielioppipuun graafisesti kuvassa 5.



Kuva 5. Lausekkeen `(3 + 4) × 7` kielioppipuu.

Kielioppipuun arvon laskemme funktiolla `eval`, jonka määrittelemme seuraavassa:

```
eval (Lit x) = x
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Saamme

```
> eval tree
49
```

Oksista (*branch*) ja lehdistä (*leaf*) koostuva puu ei suuresti eroa edellisestä.

```
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
```

Luku 6

Tyyppiluokat

Kun aiemmin määrittelimme tietotyypin `Bool`, ohitimme määrittelyn avulla tietotyypin määrittelyn standardikirjastossa `Prelude`. Määrittelemämme tietotyyppi ei täysin vastaa standardikirjaston määritelmää. Esimerkiksi oman tietotyyppimme vertailuoperaatio `(==)` johtaa virheilmoitukseen tietotyypille tunnistamattomasta operaatiosta.

```
> data Bool = False | True
> True == True
"ERROR: No instance for (Eq Bool) arising \
       \from a use of ‘==’."
```

Sen sijaan standardikirjaston tietotyypille vertailu onnistuu.

```
> Prelude.True == Prelude.True
True
```

Havaitsemme omalta tietotyypiltämme puuttuvan muitakin Haskell-kielen perustyypeille tyypillisiä ominaisuuksia. Tilanne on onneksi helppo korjata. Voimme antaa tietotyyppimme *periä* ominaisuuksia tyyppiluokilta. Yhtäsuuruusoperaatio on määritelty tyyppiluokassa `Eq`. Mikäli operaation oletustoteutus vastaa tarkoituksiamme, tyyppiluokan periminen on yksinkertainen toimenpide. Se tapahtuu tietotyypin määrittelyn yhteydessä avainsanalla `deriving`.

Automaattisen periytymisen seurauksena tyypistä `Bool` tulee tyyppiluokan

`Eq` jäsen, eli sille on määritelty tyyppiluokan ilmentymä eli *instanssi*. Funktio `(==)` tulee tällöin *ylikuormitetuksi* periaatteella: alkio on yhtäsuuri itsensä kanssa ja erisuuri muiden alkioiden suhteen.

```
> data Bool = False | True  deriving (Eq)
> True == True
True
```

Edellä kuvattu ilmiö toistuu, kun yritämme tulostaa arvon `True`.

```
> True
"ERROR: No instance for (Show Bool) arising \
       \from a use of ‘print’."
```

Voimme korjata tilanteen antamalla tietotyyppimme periä myös tyyppiluokan `Show`. Tietotyypistä `Bool` tulee näin ollen tyyppiluokan `Show` jäsen. Funktion `show` ylikuormitetuksi toteutukseksi tulee alkion nimi merkkijonoksi muutettuna.

```
> data Bool = False | True deriving (Eq,Show)
> True
True
> show True
"True"
> show False
"False"
```

Tyypillisimmin käyttämämme automaattisesti periytyvien tyyppiluokkien luettelo on `(Eq,Ord,Show,Read)`. Kertaamme näistä tutut ja esittelemme uudet.

6.1 Tyyppiluokka `Eq`

Standardikirjasto `Prelude` määrittelee tyyppiluokalle `Eq` yhtäsuuruus- ja erisuuruusoperaatiot (`==` ja `/=`). Näistä riittää toisen määrittely, toinen määritelmä seuraa ensimmäisen negaationa.

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
```

```
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

Määrittelemme seuraavassa tyypin `Boolean`, joka voi saada toisen arvoista `T` tai `F`. Kirjoitamme tyypille myös tyyppiluokan `Eq` instanssin.

```
data Boolean = F | T

instance Eq Boolean where
  T == T  =  True
  F == F  =  True
  _ == _  =  False
```

Saamme

```
> T == F
False
> T == T
True
> T /= F
True
```

6.2 Tyyppiluokka `Ord`

Tyyppiluokka `Ord` määrittää matemaattiset vertailuoperaatiot `<`, `<=`, `>` ja `>=`. Samoin se määrittää funktiot `min`, `max` ja `compare`.

Luokan pienin vaadittava määrittely on joko operaatio `<=` tai funktio `compare`. Luokan jäsenen tulee kuulua myös luokkaan `Eq`.

```
class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

  compare x y | x == y    = EQ
              | x <= y    = LT
              | otherwise = GT
```

```
    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    -- Note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

Tyyppiluokan `Ord` avulla voimme kirjoittaa kivi-paperi-sakset -leikin toteutuksen. Leikin säännöt ovat yksinkertaiset: sakset voittaa paperin, kivi voittaa sakset ja paperi voittaa kiven. Huomaamme, että tyyppiluokassa `Ord` alkiot voivat muodostaa järjestysominaisuuden suhteen silmukan, eli kahdesta alkiosta voimme valita suuremman (`max`), mutta suurinta (`maximum`) ja pienintä (`minimum`) alkiota ei ole.

```
data Roshambo = Rock | Paper | Scissors
  deriving (Eq,Show)
instance Ord Roshambo where
  Paper <= Scissors  =  True
  Scissors <= Rock  =  True
  Rock <= Paper  =  True
  _ <= _  =  False
```

Nyt saamme

```
> Scissors > Paper
True
> Paper > Rock
True
> Rock > Scissors
True
> max Paper Scissors
Scissors
> max Rock Paper
Paper
```

6.3 Tyyppiluokka `Enum`

Tyyppiluokka `Enum` tekee tyypin arvoista lueteltavia (*enumerable*).

Voimme luetella esimerkiksi haluamamme värit haluamassamme järjestyksessä.

```
data Color = Black | White | Green | Blue | Violet |
  Red | Orange | Yellow | Pink | Brown
  deriving (Enum,Show)
```

Nyt tyyppiluokan `Enum` periminen mahdollistaa arvoilla rajatun luettelon tulostamisen.

```
> [Green .. Brown]
[Green,Blue,Violet,Red,Orange,Yellow,Pink,Brown]
```

Tyyppiluokka `Enum` määrittelee funktiot `succ`, `pred`, `fromEnum`, `enumFrom`, `enumFromThen`, `enumFromTo` ja `enumFromThenTo`. Näistä `succ` palauttaa edellä olevan arvon ja `pred` jäljessä tulevan. Funktiot `enumFrom`, `enumFromThen`, `enumFromTo` ja `enumFromThenTo` määrittelevät Haskell-kielen lyhennysmerkinnän `[n,m..t]` muuttujat `n` (*from*), `m` (*then*) ja `t` (*to*).

Funktio `fromEnum` palauttaa alkion järjestysluvun.

```
> fromEnum Blue
3
> enumFrom Yellow
[Yellow,Pink,Brown]
> enumFromThen Violet Orange
[Violet,Orange,Pink]
> enumFromThenTo 1 3 11
[1,3,5,7,9,11]
> pred White
Black
> succ White
Green
> succ 4
5
```

6.4 Tyyppiluokka `Bounded`

Tyyppiluokka `Bounded` määrittelee funktiot `minBound` ja `maxBound`.

```
data Color = Black | White | Green | Blue | Violet |
  Red | Orange | Yellow | Pink | Brown
  deriving (Bounded,Show)
```

Funktio `minBound` palauttaa tyypin alarajan ja funktio `maxBound` ylärajan.

```
> minBound :: Color
Black
> maxBound :: Color
Brown
```

6.5 Tyyppiluokka `Show`

Tyyppiluokka `Show` määrittelee muun muassa funktion `show`. Funktion tehtävä on tekstuaalisen esityksen tuottaminen tyypin arvoille.

```
> show Green
"Green"
> show 12
"12"
```

6.6 Tyyppiluokka `Read`

Tyyppiluokka `Read` mahdollistaa tekstuaalisen esityksen lukemisen tiettyä tyyppiä olevaksi arvoksi.

```
data Color = Black | White | Green | Blue | Violet |
  Red | Orange | Yellow | Pink | Brown
  deriving (Read,Show)
```

Esimerkiksi merkkijonosta `"Pink"` saamme tyypin `Color` arvon `Pink` ja merkkijonosta `"5"` tyypin `Int` arvon 5.

```
> read "Pink" :: Color
Pink
> read "5" :: Int
5
```

6.7 Tyyppiluokka `Functor`

Tyyppiluokka `Functor` määrittelee kuvausfunktion `fmap`.

Haskell-kielen listat `[a]` kuuluvat tyyppiluokkaan `Functor`. Niiden tyyppiluokan `Functor` instanssi määrittelee kuvausfunktion `map` funktion `fmap` toteutuksena.

```
instance Functor [] where
  fmap = map
```

Käytimme aiemmin funktiota `map` numeerisen listan alkioiden laskutoimituksiin.

```
> map (^ 2) [1..7]
[1,4,9,16,25,36,49]
```

Funktion `map` tyyppiallekirjoitus on `(a -> b) -> [a] -> [b]` eli prefix-muodossa `(a -> b) -> [] a -> [] b`. Tämä on sama kuin funktion `fmap` tyyppiallekirjoitus `(a -> b) -> f a -> f b`, kun `f` = `[]`.

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

Tyyppiluokan `Functor` merkitys käy selville esimerkiksi tyypin `Maybe` instanssin määrittelystä.

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Tässä olennainen asia on, että funktio `fmap` ei käytä funktiota `f` arvolle `Nothing`. Tämä on mielekästä, sillä arvon `Nothing` merkitys usein on ilmaista funk-

tiokutsun epäonnistumisesta. Epäonnistuneen funktiokutsun jälkeen emme halua lisää funktiokutsuja.

Tyypille `Either` on määritelty tyyppiluokan `Functor` instanssi samaan tapaan.

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

6.8 Tyyppiluokka `Monad`

Tyyppiluokka `Monad` määrittelee funktiot `(>>=)` (*bind*) ja `return`.

```
class  Monad m  where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
    fail    :: String -> m a

    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

Luokkaan kuuluva funktio `(>>)` (*then*) on funktion `(>>=)` erikoistapaus. Funktio `fail` toimii operaation virhevaihtoehtona, kun puramme monadiarvon muuttujaan sijoitusnuolen `<-` avulla.

Haskell kielen `do`-lauseke on lyhentävä ja ohjelman luettavuutta parantava merkintätapa monadifunktioille `(>>)` ja `(>>=)`.

Lauseke

```
do
  e1
  e2
```

vastaa lauseketta

```
e1 >> e2
```

ja lauseke

```
do
  p <- e1
  e2
```

lauseketta

```
e1 >>= \p -> e2
```

Virhemahdollisuus huomioiden se on sama kuin lauseke

```
e1 >>= (\v -> case v of
  p -> e2
  _ -> fail "s")
```

Saamme esimerkiksi

```
> e1 = print "e1"
> e2 = print "e2"
> do e1; e2
"e1"
"e2"
> e1 >> e2
"e1"
"e2"
```

Lauseke

```
do
  p <- getLine;
  print p
```

puolestaan merkitsee samaa kuin

```
getLine >>= (\p -> print p)
```

Monadi `Maybe` määrittelee funktiot `return`, `(>>=)` ja `fail` seuraavasti:

```
instance Monad Maybe where
  return :: a -> Maybe a
  return = Just
```

```
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (Just x) >>= g = g x
  Nothing  >>= _ = Nothing

  fail _ = Nothing
```

Yritämme seuraavassa sovittaa monadiarvon sijoitusnuolen `<-` avulla arvoja `[2,1]`, `[1]` ja `[]` parametrimuotoon `(x:xs)`.

```
just2 = do (x:xs) <- Just [2,1]; return x
just1 = do (x:xs) <- Just [1]; return x
fail1 = do (x:xs) <- Just []; return x
```

Nyt funktiot `just2`, `just1` ja `fail1` palauttavat arvot `Just 2`, `Just 1` ja `Nothing`.

```
> just2
Just 2
> just1
Just 1
> fail1
Nothing
```

Listamonadi (`[]`) määrittelee funktiot `return`, `(>>=)` ja `fail` seuraavasti:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail _ = []
```

Funktioiden `map` ja `concat` valintaa bind-funktion `(>>=)` toteutukseen sekä funktioiden `return` ja `fail` paluuarvojen valintaa voisimme perustella esimerkiksi niiden käyttökelpoisuudella listamuodostimien rakennusosina.

```
> map (\x -> if even x then [x] else []) [1..6]
[[],[2],[],[4],[],[6]]
> concat [[],[2],[],[4],[],[6]]
[2,4,6]
```

Tämä vastaa jotakuinkin listamuodostinta

```
> [x | x <- [1..6], even x]
```

```
[2,4,6]
```

Vuorovaikutteisen tulkin komennolla `:info` saamme selville, että listat `([])` ovat muun muassa tyyppiluokkien `Monad`, `Functor`, `Applicative` ja `Monoid` jäseniä.

```
> :info []
data [] a = [] | a : [a]
...
instance Monad []
instance Functor []
instance Applicative []
instance Monoid [a]
...
```

Koska listat ovat monadi, voimme monadiarvon sijoitusnuolen `<-` avulla purkaa muuttujan monadista (ja lisätä sen vaikkapa takaisin funktion `return` avulla).

Lauseke

```
do
  x <- [1..4]
  return x
```

palauttaa siten arvon `[1,2,3,4]`.

```
> do x <- [1..4]; return x
[1,2,3,4]
```

6.9 Tyyppiluokka `Num`

Numeeristen tyyppiluokkien laajin luokka on tyyppiluokka `Num`, johon kuuluvat kaikki numeeriset tyypit.

Esimerkiksi literaali `5` on oletuksena tyyppiluokan `Num` arvon esitystapa.

```
> :t 5
5 :: Num t => t
```

Tyyppiluokassa `Num` on määritelty funktiot `(+)`, `(-)`, `(*)`, `negate`, `abs`, `signum` ja `fromInteger`.

6.10 Tyyppiluokka `Real`

Tyyppiluokassa `Real` on määritelty funktio `toRational`. Luokkaan kuuluvat lukutyypit `Word`, `Integer`, `Int`, `Float` ja `Double`.

6.11 Tyyppiluokka `Integral`

Tyyppiluokka `Integral` on kokonaislukutyyppien `Int`, `Word` ja `Integer` luokka. Luokka määrittelee kokonaisjakofunktiot `quot`, `rem`, `div`, `mod`, `quotRem` ja `divMod` sekä funktion `toInteger`.

6.12 Tyyppiluokka `RealFrac`

Tyyppiluokkaan `RealFrac` kuuluvat liukulukutyypit `Float` ja `Double`. Luokka määrittelee muun muassa funktiot `truncate`, `round`, `ceiling` ja `floor`.

6.13 Tyyppiluokka `Floating`

Tyyppiluokkaan `Floating` kuuluvat liukulukutyypit `Float` ja `Double`. Luokka määrittelee matemaattiset funktiot `pi`, `exp`, `log`, `sqrt`, `(**)`, `logBase`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, ja `atanh`.

6.14 Tyyppiluokka `Fractional`

Tyyppiluokka `Fractional` määrittelee funktiot `(/)`, `recip` ja `fromRational`. Luokkaan kuuluvat liukulukutyypit `Float` ja `Double`.

Luku 7

Geometrisia kuvioita

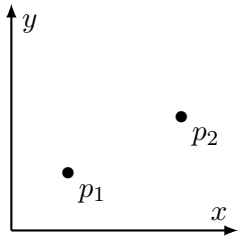
7.1 Piste `Point`

Määrittelemme tyypin `Point`, joka kuvaa pistettä (x, y) kaksiulotteisessa avaruudessa.

```
data Point = Point Double Double
  deriving Show
```

Pisteet `p1` ja `p2` sijaitsevat koordinaateissa $p_1 = (1, 1)$ ja $p_2 = (3, 2)$ (kuva 6).

```
p1 = Point 1 1
p2 = Point 3 2
```



Kuva 6. Pisteet $p_1 = (1, 1)$ ja $p_2 = (3, 2)$.

Perimällä tyyppiluokan `Show` saamme tekstuaalisen esityksen pisteille `p1` ja

p2. Pisteet p1 ja p2 ovat tyyppiä Point ja niiden muodostama lista [p1,p2] tyyppiä [Point].

```
> p1
Point 1.0 1.0
> p2
Point 3.0 2.0
> :type [p1,p2]
[p1,p2] :: [Point]
```

7.2 Tyyppi Shape

Määrittelemme tyypin Shape, joka voi olla ympyrä Circle, viiva Line, viivajono PolyLine tai ympyränkaari Arc.

```
data Shape = Circle Double Point
  | Line Point Point
  | PolyLine [Point]
  | Arc Double Point Angle Angle
  deriving Show
```

Ympyrän Circle parametrit ovat säde r ja keskipisteen koordinaatit (x, y).

Viivan Line parametrit ovat päätepisteiden koordinaatit (x_1, y_1) ja (x_2, y_2).

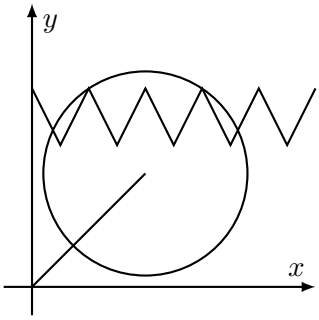
Viivajonon PolyLine parametri on koordinaattipisteiden lista tyyppiä [Point].

Kaari Arc saa parametreinaan kaaren säteen, keskipisteen, alkukulman ja loppukulman.

Piirrämme esimerkkinä viivan line1, jonka lähtöpiste on $(0, 0)$ ja päätepiste $(2, 2)$, ympyrän circle1, jonka säde on $r = 1.8$ ja jonka keskipiste sijaitsee pisteessä $(2, 2)$ sekä viivajonon polyline1, joka muodostaa siksak-kuvion kooordinaattimuuttujien xs = [0,0.5..5] ja ys = cycle [3.5,2.5] määrittämänä (kuva 7).

```
line1 = Line (Point 0 0) (Point 2 2)
circle1 = Circle 1.8 (Point 2 2)
```

```
ys = cycle [3.5,2.5]
xs = [0,0.5..5]
polyline1 = PolyLine [Point x y | (x,y) <- zip xs ys]
```



Kuva 7. Viiva `line1`, ympyrä `circle1` ja viivajono `polyline1`.

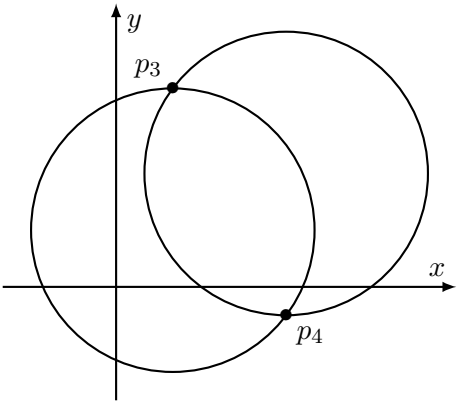
Haskell-kielessä listan alkioiden tulee olla samaa tyyppiä. Määrittelimme ympyrän, viivan ja viivajonon tyypin `Shape` alkioiksi. Näin ollen ne täyttävät listan alkioille asetetun vaatimuksen, ja voimme koota viivan `line1`, ympyrän `circle1` ja viivajonon `polyline1` listaksi, jonka tyyppi on `[Shape]`. Annamme listalle nimen `shapes1`.

```
> shapes1 = [line1,circle1,polyline1]
> :t shapes1
shapes1 :: [Shape]
```

7.3 Ympyröiden leikkauspisteet

Piirrämme kaksi ympyrää c_1 ja c_2, joiden keskipisteinä ovat pisteet $p_1 = (1, 1)$ ja $p_2 = (3, 2)$. Molempien ympyröiden säde on $r = 2.5$. Etsimme ympyröiden leikkauspisteet p_3 ja p_4 (kuva 8).

```
p1 = Point 1 1
p2 = Point 3 2

c1 = Circle 2.5 p1
c2 = Circle 2.5 p2
```



Kuva 8. Ympyröiden c_1 ja c_2 leikkauspisteet p_3 ja p_4.

Kahden ympyrän väliset leikkauspisteet saamme määrittelemällä function `circleCircleIntersections`. Funktio saa parametreinaan kaksi ympyrää tyyppiä `Circle`. Funktio palauttaa listan leikkauspisteistä tyyppiä `[Point]`.

```
-- | Intersection points of two circles
-- Algorithm from
-- http://paulbourke.net/geometry/circlesphere/
circleCircleIntersections circle1 circle2
  | d > r1 + r2        = [] -- none
  | d < abs (r1 - r2)  = [] -- none
  | d == 0 && r1 == r2 = [] -- infinitely many
  | otherwise = [Point x3 y3, Point x4 y4]
  where
    h = sqrt((r1*r1) - (a*a))
    a = (r1*r1 - r2*r2 + d*d) / (2*d)
    d = dist p1 p2
    Point x1 y1 = p1
    Point x2 y2 = p2
    Circle r1 p1 = circle1
    Circle r2 p2 = circle2
    (dx,dy) = (x2 - x1,y2 - y1)
    x = x1 + a * dx / d
    y = y1 + a * dy / d
```

```
    [x3,x4] = [x `mp` (h * dy / d) | mp <- [(-),(+)]]
    [y3,y4] = [y `pm` (h * dx / d) | pm <- [(+),(-)]]
```

Kun ympyröiden keskipisteiden välinen etäisyys on suurempi kuin säteiden summa, ovat ympyrät toisistaan erillään, eikä niillä ole leikkauspisteitä. Tämä ehto on algoritmissa kuvattu muodossa

```
d > r1 + r2
```

Tällöin algoritmin palautusarvo on tyhjä lista `[]`.

Ympyröillä ei myöskään ole leikkauspisteitä, jos ne ovat sisäkkäin ja keskipisteiden välinen etäisyys on pienempi kuin säteiden erotus. Tällöin on voimassa ehto

```
d < abs (r1 - r2)
```

Ympyröiden ollessa päällekkäin, niillä on äärettömän monta leikkauspistettä. Ehto saa muodon

```
d == 0 && r1 == r2
```

Olemme algoritmissa samaistaneet tilanteet, joissa ympyröillä ei ole lainkaan leikkauspisteitä tai niillä on äärettömän monta leikkauspistettä. Tällainen valinta on usein laskennan kannalta mielekkäin vaihtoehto.

Kun ympyröillä on kaksi leikkauspistettä, algoritmi palauttaa listan

```
[Point x3 y3, Point x4 y4]
```

Leikkauspisteiden yhtyessä pisteet `Point x3 y3` ja `Point x4 y4` ovat laskentatarkkuuden rajoissa samat.

Leikkauspisteiden laskennassa käyttämämme pisteiden p_1 ja p_2 välinen etäisyys on

```
d = dist p1 p2
```

Funktiossa `dist` laskemme kahden pisteen välisen euklidisen etäisyyden matematiikasta tutulla menetelmällä.

```
-- | The euclidian distance between two points.
dist (Point x0 y0) (Point x1 y1) =
  sqrt ((sqr dx) + (sqr dy))
```

```
  where
    sqr x = x * x
    dx = x1 - x0
    dy = y1 - y0
```

7.4 Kulmatyyppi `Angle`

Kulman esittämiseksi radiaaneina, asteina tai gooneina määrittelemme tietotyypin `AngleType a`. Tulemme käyttämään kulman arvoina kaksinkertaisen tarkkuuden liukulukuja tyyppiä `Double`, joten muodostamme avainsanalla `type` uuden tyypin `Angle`.

```
data AngleType a = RAD a | DEG a | GON a
  deriving Show

type Angle = AngleType Double
```

Uuden tyypin määrittely avainsanalla `type` ei tee arvoista automaattisesti uuden tyypin edustajia.

```
> deg a = DEG a
> :type deg
deg :: a -> AngleType a
```

Kun sen sijaan esitämme tyyppimäärittelyn funktiomäärittelyn yhteydessä, tulee myös määritellyn funktion tyypiksi uusi tyyppi.

```
> deg :: Double -> Angle; deg a = DEG a
> :type deg
deg :: Double -> Angle
```

7.5 Vakiofunktiot `halfpi` ja `twopi`

Matematiikasta tiedämme, että ympyrän neljännestä vastaava keskuskulma on radiaaneina $\pi/2$, puolikasta ympyrää vastaava keskuskulma π ja täyttä ympyrää vastaava keskuskulma 2π. Näistä Haskell-kieli tuntee ennalta funktion `pi` $= \pi$. Määrittelemme funktion `pi` avulla funktiot `halfpi` ja `twopi`.

```
halfpi = pi / 2
twopi  = 2 * pi
```

Voimme nyt esittää tekstuaalisessa muodossa radiaaneina ympyrän neljännestä, puolikasta ympyrää ja täyttä ympyrää vastaavat keskuskulmat.

```
> pi
3.141592653589793
> RAD halfpi
RAD 1.5707963267948966
> RAD pi
RAD 3.141592653589793
> RAD twopi
RAD 6.283185307179586
```

7.6 Funktiot `degrees`, `gons` ja `radians`

Voimme muuntaa kulman arvoja yksiköstä toiseen määrittelemällä funktiot `degrees`, `gons` ja `radians`.

```
degrees (RAD r) = DEG (r * 180 / pi)
degrees (GON g) = DEG (g * 180 / 200)
degrees (DEG d) = DEG d
gons (RAD r) = GON (r * 200 / pi)
gons (DEG g) = GON (g * 200 / 180)
gons (GON g) = GON g
radians (GON g) = RAD (g * pi / 200)
radians (DEG d) = RAD (d * pi / 180)
radians (RAD r) = RAD r
```

Saamme esimerkiksi

```
> radians (GON 100)
RAD 1.5707963267948966
> degrees (RAD halfpi)
DEG 90.0
> gons (DEG 360)
GON 400.0
```

7.7 Funktiot `addAngles` ja `subAngles`

Määrittelemme seuraavaksi funktiot kulmayksiköiden yhteen- ja vähennyslaskulle.

Mikäli operandeilla on yhteinen kulmayksikkö, käytämme kulman arvojen välisissä yhteen- ja vähennyslaskuissa sitä. Muussa tapauksessa muunnamme kulman arvot radiaaneiksi.

```
add (DEG a) (DEG b) = DEG (a + b)
add (RAD a) (RAD b) = RAD (a + b)
add (GON a) (GON b) = GON (a + b)
add a b = radians a `add` radians b

sub (DEG a) (DEG b) = DEG (a - b)
sub (RAD a) (RAD b) = RAD (a - b)
sub (GON a) (GON b) = GON (a - b)
sub a b = radians a `sub` radians b
```

Päättelemme, että ohjelman ymmärrettävyys saattaa parantua, jos käytämme funktionimien `add` ja `sub` sijasta funktionimiä `addAngles` ja `subAngles`.

```
addAngles = add
subAngles = sub
```

Saamme kahden neljänneskulman summaksi puolikkaan täyskulmasta sekä neljänneskulman ja puolikkaan summaksi 3/4 täyskulmasta.

```
> RAD halfpi `addAngles` DEG 90
RAD 3.141592653589793
> DEG 90 `addAngles` DEG 180
DEG 270.0
```

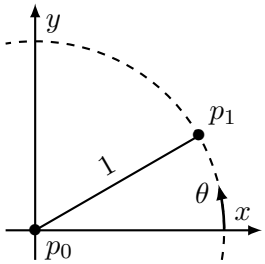
7.8 Funktiot `sin1`, `cos1` ja `tan1`

Trigonometriset funktiot toteutamme funktioina `sin1`, `cos1` ja `tan1`. Jos kulma ei ole radiaaneissa, muunnamme sen ensin radiaaneiksi ja kutsumme

saadulla arvolla standardikirjaston funktioita `sin`, `cos` ja `tan`, jotka laskevat arvot suoraan radiaaneina annetusta liukuluvusta ilman konstruktoria `RAD`, `DEG` tai `GON`.

```
tan1 (RAD r) = tan r
tan1 r = tan1 (radians r)

cos1 (RAD r) = cos r
cos1 r = cos1 (radians r)

sin1 (RAD r) = sin r
sin1 r = sin1 (radians r)
```

Esimerkiksi kavuttuamme yksikköympyrän kaarta matkan $\theta = \frac{2\pi/4}{3}$ keskipisteestä piirretyn itävektorin osoittamasta nollakulmasta ympyrän huipulle, olemme tulleet korkeudelle $\sin\theta = \sin\frac{2\pi/4}{3} = 0.5$ nollatasosta (kuva 9).



Kuva 9. Sinifunktio palauttaa korkeuden nollatasosta eli yksikköympyrän keskuskulmaa θ vastaavan pisteen y-koordinaatin.

```
> sin1 (DEG 30)
0.5
> sin1 (RAD (halfpi/3))
0.5
```

7.9 Kiertomatriisi

Kun haluamme kiertää koordinaatin (x_1, y_1) origon ympäri, kerromme kiertomatriisilla R koordinaattimatriisin.

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Haskell-kielessä esitämme matriisit listoina. Esimerkiksi voimme määritellä kiertomatriisin R funktiossa `rotationMatrix`.

```
rotationMatrix t = [[cos1 t,-sin1 t],[sin1 t,cos1 t]]
```

Matriisien välinen kertolasku yleistyy kaiken kokoisille matriiseille.

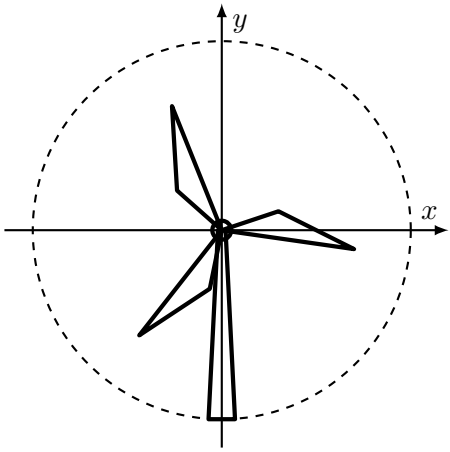
```
matrixTimes a b =
  [sum [x * y | (x,y) <- zip a1 b] | a1 <- a]
```

Nyt määrittelemme pisteen $(x1, y1)$ kierron origon ympäri kulman t verran funktiossa `rot1`.

```
rot1 t (Point x1 y1) = Point x y
  where
    [x,y] = matrixTimes (rotationMatrix t) [x1,y1]
```

Kun jaamme täysympyrän kolmeen osaan, kierrämme annetut pisteet origon ympäri ja lisäämme muutaman apuviivan, saamme tutun kuvion (kuva 10).

```
t1 = twopi / 3

blade alpha = Polygon pts2
  where
    pts2 = map (rot1 alpha) pts1
    pts1 = [Point 0 0,Point 0.7 (-0.1),Point 0.3 0.1]

tower = Polygon [p1,p2,p3,p4]
  where
    p1 = Point (-0.02) 0
    p2 = Point(-0.07) (-1)
    p3 = Point 0.07 (-1)
```



Kuva 10. Tuulimylly, jossa kiertomatriisin avulla muodostetut lavat.

```
    p4 = Point 0.02 0

rotor = Circle 0.05 (Point 0 0)

windMill = [blade (RAD alpha) | alpha <- [0,t1,2*t1]] ++
  [rotor] ++ [tower]
```

7.10 Tietotyyppi `Vector`

Määrittelemme vektoreille tietotyypin `Vector`. Vektoreilla ei ole lähtöpistettä, ainoastaan suunta ja suuruus. Jos ajattelemme vektorin lähtevän origosta, määrittelemme vektorin päätepisteen x- ja y-komponenttien avulla.

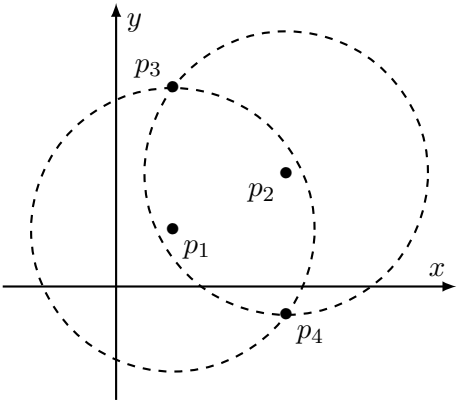
```
data Vector = Vector Double Double
  deriving Show
```

Kun viivan alkupiste on (x_0, y_0) ja loppupiste (x_1, y_1), voimme muodostaa alkupisteestä loppupisteeseen kulkevan vektorin $(x_1 - x_0, y_1 - y_0)$.

```
mkVector (Point x0 y0) (Point x1 y1) =
  Vector (x1 - x0) (y1 - y0)
```

7.11 Vektorien suuntakulmat

Etsimme seuraavaksi ympyröiden keskipisteiden p_1 ja p_2 sekä leikkauspisteiden p_3 ja p_4 välisten vektorien suuntakulmat (kuva 11).



Kuva 11. Ympyröiden keskipisteet p_1 ja p_2 sekä leikkauspisteet p_3 ja p_4.

Nollakulmaa vastaavan vektorin saamme suuntaamalla vektorin itään pisteestä (x, y) esimerkiksi pisteeseen $(x + 1, y)$.

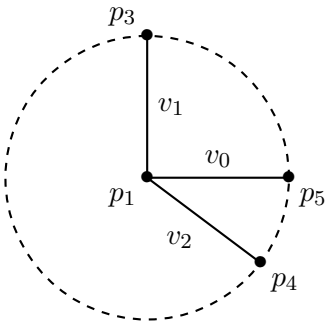
```
eastVector (Point x y) = mkVector
  (Point x y) (Point (x + 1) y)
```

Kahden vektorin välisen kulman voimme laskea standardikirjaston funktiolla `atan2`.

```
angleBt (Vector x1 y1) (Vector x2 y2) = RAD t
  where
    t = atan2 (x1*y2 - y1*x2) (x1*x2 + y1*y2)
```

Ympyrän c_1 kohdalla tilanne on kuvan 12 mukainen.

```
p1 = Point 1 1
p2 = Point 3 2
c1 = Circle 2.5 p1
c2 = Circle 2.5 p2
[p3,p4] = circleCircleIntersections c1 c2
v0 = eastVector p1
```



Kuva 12. Ympyrän c_1 keskipiste p_1, leikkauspisteet p_3 ja p_4 sekä nollakulmaa vastaava kehäpiste p_5.

```
v1 = mkVector p1 p3
v2 = mkVector p1 p4
t1 = angleBt v0 v1
t2 = angleBt v0 v2
```

Saamme

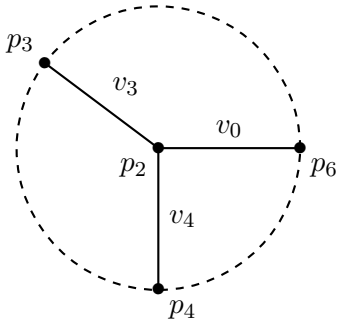
```
> t1
RAD 1.5707963267948966
> t2
RAD (-0.6435011087932844)
```

Ympyrän c_2 kohdalla voimme käyttää edellä saamiamme tuloksia ja etsiä vektorit v_3 ja v_4 (kuva 13).

```
v3 = mkVector p2 p3
v4 = mkVector p2 p4
t3 = angleBt v0 v3
t4 = angleBt v0 v4
```

Nyt saamme

```
> t3
RAD 2.498091544796509
> t4
RAD (-1.5707963267948966)
```



Kuva 13. Ympyrän c_2 keskipiste p_2, leikkauspisteet p_3 ja p_4 sekä nollakulmaa vastaava kehäpiste p_6.

7.12 Ellipsi

Ellipsin parametrimuotoinen esitys on

$$x = a \cdot \cos t$$
$$y = b \cdot \sin t$$

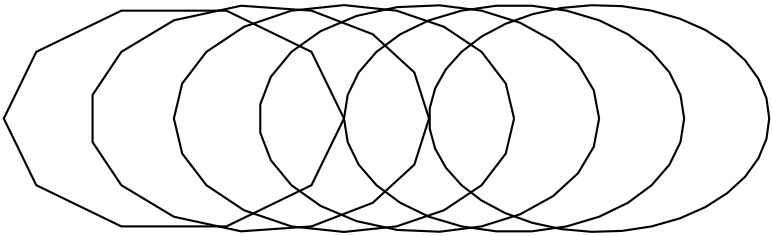
missä a ja b ovat isompi ja pienempi puoliakseli ja $t \in [0, 2\pi]$. Saamme näin ollen Haskell-kielellä pisteen ellipsin kehältä algoritmilla

```
pointOfEllipse a b t = Point x y
  where
    x = a * cos t
    y = b * sin t
```

Voimme piirtää ellipsin esimerkiksi viivajonona tyyppiä `PolyLine`. Mitä useampaan osaan jaamme viivajonon, sitä lähemmin se muistuttaa ellipsiä.

```
pls = [PolyLine (map (`addCoords` Point (1.5 * x) 0)
  (pts1 dv))
  | (x,dv) <- zip [1..] [10,15..35]]

pts1 n = [pointOfEllipse 3 2 t | t <- [0,dt..twopi]]
  where
    dt = twopi / n
```

Olemme kuvassa 14 piirtäneet ellipsit 10, 15, 20, 25, 30 ja 35 viivasegmentin avulla.



Kuva 14. Ellipsit 10, 15, 20, 25, 30 ja 35 viivasegmentin avulla piirrettyinä.

Määrittelemme funktion `lengthPL`, joka palauttaa viivajonon pituuden.

```
-- | Length of PolyLine
lengthPL pl = sum [dist p1 p2
  | (p1,p2) <- zip pts (tail pts)]
  where
    PolyLine pts = pl
```

Listan `pls` kuuden viivajonon pituudet ovat

```
> map lengthPL pls
[ 15.60, 15.75, 15.800, 15.824, 15.836, 15.8441 ]
```

7.13 Pisteet viivajonolla

Seuraavaksi haluamme sijoittaa n pistettä tasaisesti viivajonolle. Kun viivajonon pituus on l_1, tulee yhden välin pituudeksi $l_2 = l_1/n$.

```
dotsEllipse = pts
  where
    pts = [alongPL pl1 (s * l2) | s <- [1..n]]
    l2 = l1 / n
    l1 = lengthPL pl1
    n = 15
```

Viivajonon alkiot ovat tyyppiä `Point`. Määrittelimme tyypin `Point` aiemmin seuraavasti:

```
data Point = Point Double Double
```

Funktiossa `dist` teemme tyypin `Point` alkioille yhteen-, vähennys- ja kertolaskuoperaatioita, jotka säilyttävät alkioiden tyypin, joten myös funktio `dist` palauttaa arvon tyyppiä `Double`. Sama pätee funktioon `sum`. Näin ollen funktio `lengthPL` palauttaa arvon tyyppiä `Double`.

```
data Point = Point Double Double
> :t (+)
(+) :: Num a => a -> a -> a
> :t (*)
(*) :: Num a => a -> a -> a
> :t (-)
(-) :: Num a => a -> a -> a
> :t dist
dist :: Point -> Point -> Double
> :t sum
sum :: (Num a, Foldable t) => t a -> a
> :t lengthPL
lengthPL :: Shape -> Double
```

Nyt `l1` on kaksinkertaisen tarkkuuden liukuluku tyyppiä `Double`. Laskemme muuttujan `l2` arvon kaavalla `l2 = l1 / n`. Jakolaskun parametrien tulee olla samaa tyyppiä, joten Haskell-kääntäjä päättelee literaalin `n = 15` olevan tyyppiä `Double`. Näin ollen myös `l2` on tyyppiä `Double`. Nyt generaattorin `s <- [1..n]` täytyy tuottaa arvoja, joiden tyyppi on `Double`. Tämän seurauksena lauseke `(s * l2)` on tyyppiä `Double`.

```
> :t l1
l1 :: Double
> :t (/)
(/) :: Fractional a => a -> a -> a
> :t l2
l2 :: Double
```

Funktiossa `dotsEllipse` kutsumme funktiota `alongPL`, jonka tehtävä on asetella pisteet viivajonolle `pl` kun viivajonoa pitkin kuljettu etäisyys on `d`.

Toteutamme algoritmin rekursion avulla. Alussa kuljettu matka on 0, jäljellä oleva matka muuttujassa `d` ja käyttämättömät pisteet listassa `pts`.

Jos käyttämättömiä pisteitä on ainoastaan yksi, tiedämme, että olemme tulleet tiemme päähän, ja palautamme viimeisen pisteen koordinaatit.

Jos pisteitä on enemmän kuin yksi, ja jos jäljellä oleva matka on pidempi kuin ensimmäisten pisteiden väli, kuljemme tuon välin, vähennämme välin pituuden jäljellä olevasta matkasta, otamme hännän jäljellä olevista pisteistä ja kutsumme algoritmia uudelleen näillä arvoilla.

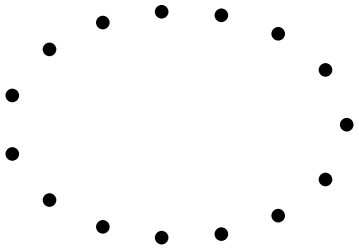
Muussa tapauksessa tiedämme, että jäljellä oleva matka on lyhyempi kuin ensimmäisten pisteiden välinen etäisyys. Tällöin siirrymme alkupisteestä kohti loppupistettä jäljellä olevan matkan ja pisteiden välisen etäisyyden suhteessa, mutta ei kuitenkaan loppupistettä edemmälle.

```
-- | A point with distance d along a PolyLine pl
alongPL pl d = along1 0 d pts
  where
    PolyLine pts = pl

-- | Recursive algorithm (internal)
along1 done left rest
  | length rest == 1 = head rest
  | left > d1 = along1 (done + d1) (left - d1) (tail rest)
  | otherwise = towards1 p1 p2 (left / d1)
  where
    d1 = dist p1 p2
    p1 = head rest
    p2 = head (tail rest)

-- | From point p1 towards p2 with respect to ratio
towards1 p1 p2 ratio = Point (x1 + r * x2) (y1 + r * y2)
  where
    r = ratio `min` 1.0
    Point x1 y1 = p1
    Point x2 y2 = p2
```

Haskell-kääntäjän interaktiivinen tulkki kertoo meille nyt, että funktio `dotsEllipse` palauttaa listan alkioita, joiden tyyppi on `Point`. Tämän tyypin tunnemme ja tiedämme, että kysymyksessä on koordinaattiarvo, jota voimme käyttää piirtämiseen.

```
> :t dotsEllipse
dotsEllipse :: [Point]
```

Esitämme syntyneen kuvion kuvassa 15.



Kuva 15. Pisteet viivajonon varrella.

7.14 Ympyrät, leikkauspisteet ja ympyränkaaret

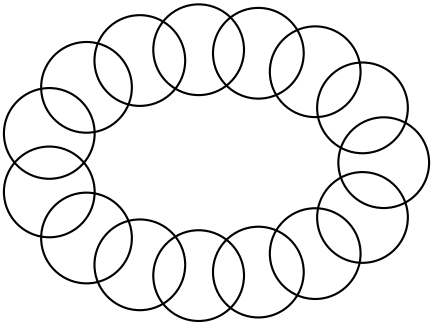
Piirrämme seuraavaksi edellä kuvatun algoritmin avulla ellipsille pisteiden sijasta ympyröitä. Ympyrän konstruktori on `Circle` ja ympyrä on tyyppiä `Shape`. Asetamme ympyrän säteeksi $r = 0.8$ (kuva 16).

```
circles1 = [Circle 0.8 p | p <- pts]
  where
    pts = [alongPL pl1 (s * l2) | s <- [1..n]]
    l2 = l1 / n
    l1 = lengthPL pl1
    n = 15
```

Viereisten ympyröiden leikkauspisteet saamme nyt funktiolla `circleCircleIntersections`. Käytämme piirroksessamme ainoastaan ulompia leikkauspisteitä. Kuvion keskipiste on pisteessä $(0, 0)$. Saamme kahdesta pisteestä ulompana sijaitsevan määrittelemällä funktion `maxDist`. Se saa parametreinaan keskipisteen ja kaksi verrattavaa pistettä.

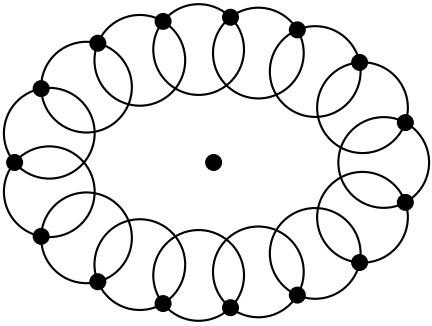
Olemme piirtäneet kuvion keskipisteen ja ulommat leikkauspisteet kuvaan 17.

```
dots2 = outer
  where
```



Kuva 16. Ympyrät viivajonolla.

```
    outer = [maxDist center p1 p2 | [p1,p2] <- xs]
    xs = [circleCircleIntersections c d
      | (c,d) <- zip c1 (tail c1)]
    c1 = circles1 ++ [head circles1]

center = Point 0 0

maxDist c a b =
  if dist c a >= dist c b then a else b
```



Kuva 17. Kuvion keskipiste ja ympyröiden ulommat leikkauspisteet.

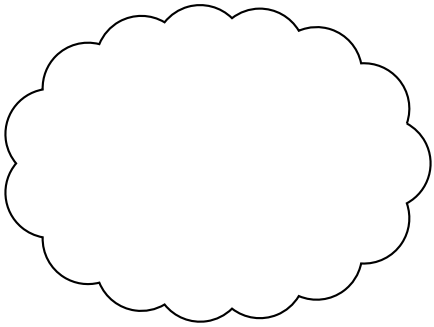
Tyypin `Shape` kuvioista kaari `Arc` saa parametreinaan kaaren säteen, keskipisteen, alkukulman ja loppukulman. Jätämme ympyröistä jäljelle vain ulompien leikkauspisteiden väliset kaaret.

```
arcs2 = [
  Arc
  (dist p0 p1)
  p0
  (angleBt (eastVector p0) (mkVector p0 p1))
  (angleBt (eastVector p0) (mkVector p0 p2))
  | (Circle r p0,p1,p2) <- zip3 circles1 dots3 dots2]
  where
    dots3 = [last dots2] ++ dots2
```

Algoritmissa lista `circles1` on ympyröiden muodostama lista, `dots2` leikkauspisteiden muodostama lista ja `dots3` listasta `dots2` muodostettu lista, jonka alkuun olemme lisänneet listan viimeisen alkion. Yhdistämme listat standardikirjaston funktiolla `zip3`, jolloin saamme kunkin kaaren keskipisteen `p0` listasta `circles1`, kaaren alkupisteen `p1` listasta `dots3` ja kaaren loppupisteen `p2` listasta `dots2`.

Ympyräkaaren säde on nyt pisteiden `p0` ja `p1` välinen etäisyys. Alkukulma on nollakulmaa vastaavan itävektorin ja keskipisteestä `p0` pisteeseen `p1` piirretyn vektorin välinen kulma. Loppukulma on itävektorin ja keskipisteestä `p0` pisteeseen `p2` piirretyn vektorin välinen kulma.

Esitämme syntyneen kuvion kuvassa 18.



Kuva 18. Ulompien leikkauspisteiden välisten kaarien muodostama kuvio.

7.15 Satunnaisluvut

Halutessamme kuvioon satunnaisuutta, voimme käyttää kirjaston `System.Random` funktioita. Otamme kirjaston käyttöön `import`-käskyllä.

```
import System.Random
```

Kirjastosta `System.Random` löydämme funktion `randomRs`, joka saa parametreinaan satunnaislukujen välin ala- ja ylärajan sekä satunnaisgeneraattorin. Alustamme satunnaisgeneraattorin vakioarvolla 42.

```
circles1 =
  [Circle (0.8+rand) p | (p,rand) <- zip pts rands]
  where
    pts = [alongPL pl1 (s * l2) | s <- [1..n]]
    l2 = l1 / n
    l1 = lengthPL pl1
    rands = randomRs (-0.15,0.15) g
    g = mkStdGen 42
    n = 15
```

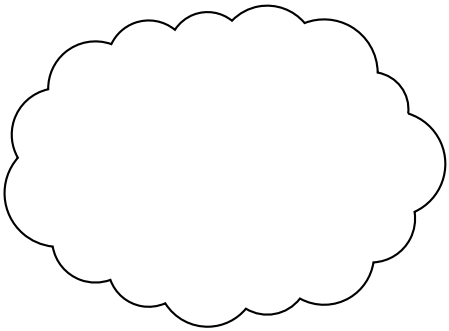
Kun alustamme satunnaisgeneraattorin vakioarvolla, ovat arvotut satunnaisluvut samat joka käynnistyskerralla. Tällä kertaa se sopii käyttötarkoitukseemme. Jos haluamme jokaisella käynnistyskerralla eri satunnaisluvut, voimme alustaa satunnaisgeneraattorin esimerkiksi järjestelmän kellonajalla.

Funktio `randomRs` tuottaa päättymättömän listan satunnaislukuja. Yhdistämme satunnaislukujen listan `rands` ympyröiden keskipisteiden listaan `pts` funktiolla `zip`. Keskipisteiden lista `pts` on äärellinen, joten myös lista `zip pts rands` on äärellinen.

Syntyneen kuvion olemme esittäneet kuvassa 19.

7.16 Kaaren piirron ongelmatilanteita

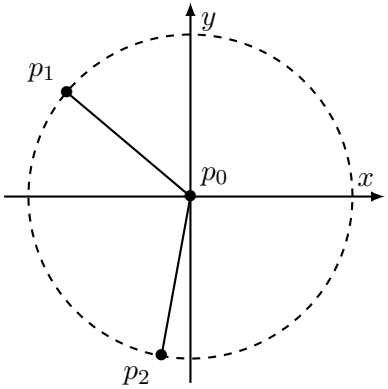
Edellä määrittelimme funktion `angleBt` palauttamaan kahden vektorin välisen suuntakulman standardikirjaston funktion `atan2` avulla. Funktio palauttaa näin ollen arvon väliltä $[-\pi, +\pi]$.



Kuva 19. Kaarien muodostama kuvio, kun ympyrän koko vaihtelee (säde $r = 0.8 \pm 0.15$).

Tyypillisesti piirtokirjastoissa ympyrän kaaren piirtäminen tapahtuu pienemmästä arvosta suurempaan riippumatta siitä, kumpi arvoista on asetettu alkukulmaksi ja kumpi loppukulmaksi.

Esimerkiksi kuvan 20 tilanteessa olemme löytäneet pisteet p_1 ja p_2, joiden suuntakulmat ovat $t(p_1) = 2.44$ rad ja $t(p_2) = -1.75$ rad, ja joiden välille haluamme piirtää ympyränkaaren. Tällöin piirtokirjasto tyypillisesti piirtää kaaren pidempää reittiä ympyrän oikeaa puolta pisteestä p_2 pisteeseen p_1, kun haluaisimme kaaren kulkevan lyhyempää reittiä ympyrän vasenta puolta.



Kuva 20. Kaaren piirrossa on varauduttava tilanteeseen, jossa pisteiden p_1 ja p_2 suuntakulmat ovat vastakkaismerkkiset, esimerkiksi $t(p_1) = 2.44$ rad ja $t(p_2) = -1.75$ rad.

Ratkaisu kaaren piirron ongelmatilanteeseen on tapauskohtainen. Tässä esimerkissä olemme ratkaisseet tilanteen lisäämällä negatiiviseen loppukulmaan yhden täyden kierroksen silloin, kun loppukulma on pienempi kuin alkukulma.

```
-- | If a2 < a1 then add DEG 360 to a2
validateArc (Arc r p a1 a2)
  | a1 <= a2  = Arc r p a1 a2
  | otherwise = Arc r p a1 (a2 `add` (RAD twopi))
```

7.17 Ympyrän ja viivan leikkauspisteet

Kun ympyrä sijaitsee origossa, ja viiva kulkee pisteiden `p1 = (Point x1 y1)` ja `p2 = (Point x2 y2)` kautta, saamme ympyrän ja viivan leikkauspisteet seuraavan algoritmin avulla:

```
-- | Intersection points of a circle at origo and a line.
-- circle = Circle r (Point 0 0)
-- line = Line (Point x1 y1) (Point x2 y2)
-- Algorithm from
-- mathworld.wolfram.com/Circle-LineIntersection.html
circleLineIntersections1 r (Point x1 y1) (Point x2 y2)
  | discr < 0  = []
  | discr == 0 = [Point x3 y3]
  | discr > 0  = [Point x3 y3, Point x4 y4]
  where
    sqr x = x * x
    dx = x2 - x1
    dy = y2 - y1
    dr = sqrt ((sqr dx) + (sqr dy))
    det = x1 * y2 - x2 * y1
    sign x
      | x < 0  = (-1)
      | otherwise = 1
    discr = sqr r * sqr dr - sqr det
    x3 = (det * dy + sign dy * dx * sqrt discr) / (sqr dr)
```

```
    y3 = ((-det) * dx + abs dy * sqrt discr) / (sqr dr)
    x4 = (det * dy - sign dy * dx * sqrt discr) / (sqr dr)
    y4 = ((-det) * dx - abs dy * sqrt discr) / (sqr dr)
```

Algoritmin käyttöalue laajenee, kun annamme ympyrän sijaita myös muualla kuin origossa.

```
-- | Intersection points of a circle and a line
-- circle = Circle r (Point x y)
-- line = Line (Point x1 y1) (Point x2 y2)
circleLineIntersections circle (Point x1 y1) (Point x2 y2) =
  [Point (x1+x0) (y1+y0) | Point x1 y1 <- pts1]
  where
    Circle r (Point x0 y0) = circle
    pts1 = circleLineIntersections1 r
      (Point (x1-x0) (y1-y0))
      (Point (x2-x0) (y2-y0))
```

Asetamme seuraavaksi pisteet `p1` ja `p2`. Pisteen `p1` koordinaatit ovat `(2,0)`. Haluamme pisteen `p2` sijaitsevan suoraan alaspäin pisteestä `p1`. Voimme laatia funktion `towards`, joka palauttaa pisteestä `p` etäisyydellä `r` olevan pisteen, kun pisteiden välinen suuntakulma on `a`.

```
towards a p r = Point (x + r * cos1 a) (y + r * sin1 a)
  where
    Point x y = p
```

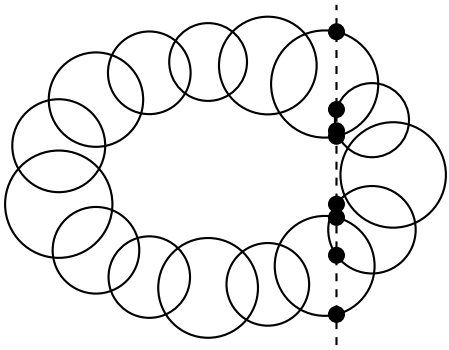
Käytämme aiemmin määrittelemiämme ympyröitä `circles1` ja määrittelemme apufunktiot `ics0` ja `ics1`.

```
ics1 = ics0 p1 p2 circles1
  where
    p1 = Point 2 0
    p2 = towards (DEG 270) p1 1

ics0 p1 p2 cs = concat
  [circleLineIntersections c p1 p2 | c <- cs]
```

Funktiossa `ics0` käytämme edellä määrittelemäämme ympyrän ja suoran leikkauspisteet laskevaa algoritmia `circleLineIntersections`. Algorit-

mi palauttaa leikkauspisteiden listan tyyppiä `[Point]`. Listamuodostin funktiossa `ics0` palauttaa siten listan tyyppiä `[[Point]]`.

Esitämme viivan, ympyrät ja niiden leikkauspisteet kuvassa 21.



Kuva 21. Viivan ja ympyröiden 8 leikkauspistettä.

7.18 Funktio `sortOn`

Kirjaston `Data.List` funktiokutsu `sortOn f xs` saa ensimmäisenä parametrinaan funktion `f`, jonka antaman säännön mukaan se poimii vertailtavat alkiot ja järjestää listan `xs`.

```
> import Data.List (sortOn)
> :t sortOn
sortOn :: Ord b => (a -> b) -> [a] -> [a]
```

Esimerkiksi funktio `snd` palauttaa tietueen toisen alkion. Nyt siis funktiokutsu `sortOn snd` järjestää listan tietueen toisen alkion mukaan.

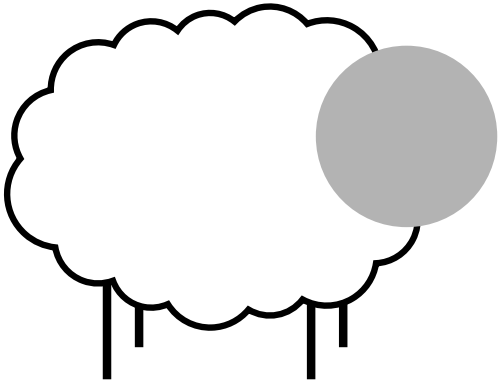
```
> sortOn snd [(5,20),(3,10),(1,30)]
[(3,10),(5,20),(1,30)]
```

Toimimme vastaavasti, kun etsimme epätyhjästä pistejoukosta `pts` pistettä, jolla on pienin y-koordinaatti. Järjestämme tällöin pistejoukon funktiokutsulla `sortOn coordY pts`, ja valitsemme listan pään funktiolla `head`.

```
minY pts = head srt
```

```
  where
    srt = sortOn coordY pts
    coordY (Point x y) = y
```

Olemme ohessa näin menetellen piirtäneet viivoja kuvioon (kuva 22) ja havaitsemme, että kuvion juoni alkaa hahmottua.



Kuva 22. Kuva alkaa hahmottua.

7.19 Funktiot `scanl` ja `scanl1`

Funktio `scanl` (*scan-left*) on läheisessä suhteessa funktioon `foldl`. Englannin kielen sanasta *scan* annetut suomennokset ”tutkia pala palalta” ja ”tutkia järjestelmällisesti” ovat varsin hyviä kuvaamaan funktion `scanl` toimintaa. Siinä missä funktio `foldl` palautti rekursiivisen taittelun lopputuloksen, palauttaa funktio `scanl` rekursion välitulokset listana.

```
> scanl (+) 1 [2,3,4]
[1,3,6,10]
```

Funktio `scanl1` toimii kuten `scanl`, mutta ottaa alkuarvoksi listan ensimmäisen alkion.

```
> scanl1 (+) [1,2,3,4]
[1,3,6,10]
```

Määrittelemme funktion `move0`, joka laskee kahden pisteen koordinaatit yhteen.

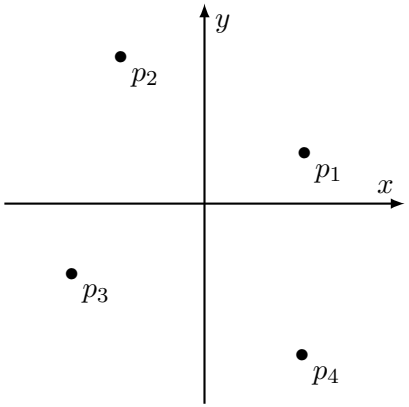
```
move0 (x1,y1) (x2,y2) = (x1 + x2,y1 + y2)
```

Määrittelemme pistejoukon `pts4` pisteet kunkin suhteessa edelliseen.

```
pts4 = [ (30.00,15.00), (-55.08,28.87), (-14.71,-65),
  (69.04,-24.41) ]
```

Saamme nyt pisteiden absoluuttisen sijainnin funktiokutsulla `scanl1 move0 pts4` (kuva 23).

```
sc1 = scanl1 move0 pts4
```



Kuva 23. Pistejoukon `pts4` pisteet siirrettynä funktiokutsulla `scanl1 move0 pts4`.

7.20 Bezier-käyrät

Sanomme kuutiolliseksi *Bezier-käyräksi* käyrää $B(t)$, jonka kulku määräytyy pisteiden p_0, p_1, p_2 ja p_3 mukaan painotettuna kaavalla

$$B(t) = (1-t)^3 \cdot p_0 + 3(1-t)^2 t \cdot p_1 + 3(1-t)t^2 \cdot p_2 + t^3 \cdot p_3$$

Tässä muuttuja t saa arvot väliltä $0 \leq t \leq 1$. Piste p_0 on käyrän alkupiste ja piste p_3 loppupiste. Kun $t = 0$, olemme käyrän alussa pisteessä p_0. Kun

$t = 1$, olemme käyrän lopussa pisteessä p_3. Pisteet p_1 ja p_2 ovat vetovoimapisteitä, joiden suuntaan käyrä kaartuu, kuitenkaan (yleensä) kulkematta niiden lävitse.

Määrittelemällä Haskell-kielisen funktion `bezier` voimme laskea pisteitä annetun Bezier-käyrän varrelta.

```
--| Cubic Bezier curve
-- https://en.wikipedia.org/wiki/B%C3%A9zier_curve
bezier p0 p1 p2 p3 t = foldr1 move0 [
  ((1 - t) ** 3) `scale0` p0,
  (3 * (1 - t) ** 2 * t) `scale0` p1,
  (3 * (1 - t) * t ** 2) `scale0` p2,
  (t ** 3) `scale0` p3 ]
```

Tässä funktio `scale0` on skalaarin `k` ja vektorin `(x1,y1)` välinen kertolasku.

```
scale0 k (x1,y1) = (k * x1,k * y1)
```

Jos haluamme käsitellä lukuparin `(x,y)` sijasta koordinaattipistettä `Point x y`, voimme määritellä funktiota `scale0` vastaavan funktion `scaleCoords`.

```
scaleCoords k (Point x1 y1) = Point (k * x1) (k * y1)
```

Määrittelemme jokaiselle pistevälille oman Bezier-käyränsä. Tätä varten tarvitsemme listan vetovoimapisteistä ja päätepisteet. Edellisen välin päätepiste toimii aina seuraavan välin alkupisteenä, joten selviämme määrittelemällä siirrokset kolmeen pisteeseen.

```
pts0 = [
 (-12.75,26.54),  (-35.17,37.72),  (-55.08,28.87),
 (-24.64,-13.44), (-23.61,-46.86), (-14.71,-65.56),
 (11.01,-22.87),  (46.80,-38.26),  (69.04,-24.41),
 (16.65,14.17),   (10.60,40.59),   (0.74,61.10) ]
```

7.21 Funktio `chunksOf`

Kirjaston `Data.List.Split` funktiokutsu `chunksOf n` jakaa listan alilistoiksi, joissa kussakin on `n` alkiota.

```
> chunksOf 3 [1..12]
[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
```

Kirjoitamme funktion `headPts1`, joka palauttaa Bezier-käyrän piirtoon vaadittavat koordinaatit tietueena muodossa `(p0,p1,p2,p3)`.

```
headPts1 = zip4 sc1 ex1 ex2 (tail sc1)
  where
   ex2 = [move0 a b | (a,b) <- (zip sc1 ext2)]
   ex1 = [move0 a b | (a,b) <- (zip sc1 ext1)]
   ext2 = [p1 !! 1 | p1 <- pts2]
   ext1 = [p1 !! 0 | p1 <- pts2]
   sc1 = scanl move0 (2,0) pts3
   pts3 = [p1 !! 2 | p1 <- pts2]
   pts2 = chunksOf 3 pts0
```

Tässä käytämme funktiota `chunksOf` listan `pts0` jakamiseen kolmen alkion alilistoiksi. Näistä lista `pts3` sisältää välin alkupisteen suhteelliset koordinaatit. Muunnamme suhteelliset koordinaatit absoluuttisiksi koordinaateiksi funktiokutsulla `scanl move0 (2,0) pts3`. Listat `ext1` ja `ext2` sisältävät vetovoimapisteiden suhteelliset koordinaatit. Muutamme myös ne absoluuttisiksi koordinaateiksi (listat `ex1` ja `ex`).

Kun nyt pakkaamme listat neljän alkion tietueiksi funktiolla `zip4`, voimme laskea tietueen `(p0,p1,p2,p3)` avulla pisteen Bezier-käyrältä (kuva 24).

```
sheepHead1 = [mkPoint (bezier p0 p1 p2 p3 t)
  | (p0,p1,p2,p3) <- headPts1, t <- [0.0,0.1..0.9]]
```

Määrittelemme funktion `mkPoint` palauttamaan lukuparin `(x,y)` koordinaatit muodossa `Point x y`.

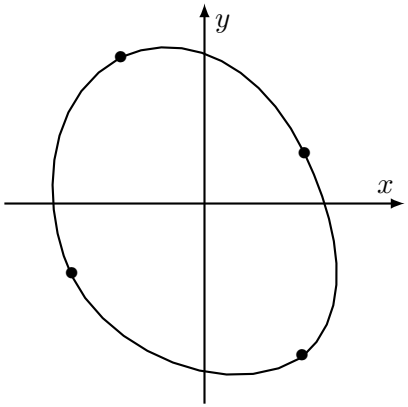
```
mkPoint (x,y) = Point x y
```

Jos haluamme tehdä muunnoksen vastakkaiseen suuntaan, voimme määritellä funktion `toTuple`.

```
toTuple (Point x y) = (x,y)
```

Voimme nyt pienentää ja siirtää kuvion oikeaan paikkaan.

```
sheepHeadB = pts3
```



Kuva 24. Neljän Bezier-käyrän muodostama kuvio.

```
  where
    pts3 = map (addCoords (Point 3.38 0.78)) pts2
    pts2 = map (scaleCoords 0.038) sheepHead1
```

7.22 Täytetyt ympyrät

Haluamme, että ainakin ympyrät (`Circle`) ja monikulmiot (`Polygon`) voivat olla myös täytettyjä (`Filled`). Tätä tarkoitusta varten määrittelemme rekursiivisen tietotyypin `Filled Shape`.

```
data Shape = Circle Double Point
  | Line Point Point
  | Polygon [Point]
  | PolyLine [Point]
  | Arc Double Point Angle Angle
  | Filled Shape
```

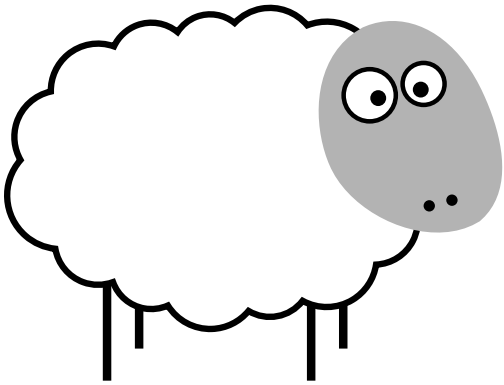
Periaatteessa määrittelymme mahdollistaa kaikkien kuvioiden täyttämisen, mutta käytännössä emme varmaankaan halua täyttää viivoja tai avoimia viivajonoja.

Mikäli rekursiivisella tyypillä on parametreja, joudumme luonnollisesti suluttamaan nämä erikseen, kuten täytetyn ympyrän `Filled (Circle r pt)`

tapauksessa.

```
head1 = [Filled (Circle 1.6 (Point 3.2 0.5))]
```

Voimme tehdä avoimista kuvioista täytettyjä kuvioita, tai halutessamme säilyttää molemmat (kuva 25).



Kuva 25. Pää, silmät ja nenä.

```
sheepHead2 = [Filled (Polygon sheepHeadB)]


eyesWhite  = map Filled (eyes1 1)
eyesBorder = map Filled (eyes1 2)

eyes1 i = [Circle r1 pt1, Circle r2 pt2]
  where
    pt1 = Point 2.55 1.25
    pt2 = Point 3.5 1.45
    (r1,r2) = if i==1 then (0.42,0.33) else (0.50,0.41)

pupils1 = map Filled [Circle r3 pt3, Circle r4 pt4]
  where
    pt3 = Point 2.7 1.2
    pt4 = Point 3.45 1.35
```

```
    (r3,r4) = (0.14,0.14)

nose1 = map Filled [Circle r1 pt1, Circle r2 pt2]
  where
    pt1 = Point 3.6 (-0.7)
    pt2 = Point 4.0 (-0.6)
    (r1,r2) = (0.10,0.10)
```

7.23 Käyrien tuonti vektorigrafiikkaohjelmasta

Kun piirrämme käyriä vektorigrafiikkaohjelmalla, tallentaa ohjelma käyristä tyypillisesti alkupisteen komennolla `m` (*move*) sekä kuutiollisen Bezier-käyrän pisteet komennolla `c` (*cubic*) suhteellisina koordinaatteina.

```
"m 94.95,138.60 c -3.01,-2.15 -5.58,-2.93 -8.17,-1.88
  -3.52,1.42 -4.33,4.44 -0.62,5.12 2.63,0.47 5.16,-3.41
  7.90,-1.41"
```

Alkupiste ei ole piirroksemme kannalta lainkaan oikea, joten voimme jättää sen pois listauksesta Haskell-kielellä. Saamme nyt

```
earR = [
  (-3.01,-2.15), (-5.58,-2.93), (-8.17,-1.88),
  (-3.52,1.42),  (-4.33,4.44),  (-0.62,5.12),
  (2.63,0.47),   (5.16,-3.41),  (7.90,-1.41) ]
```

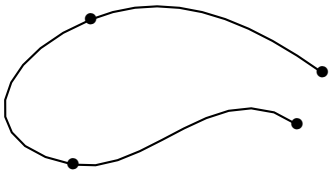
Kuviomme ei tällä kerta muodosta silmukkaa, joten piirrämme kunkin käyrän alusta loppuun (`t <- [0.0,0.1..1.0]`).

```
sheepEar earPts = [mkPoint (bezier p0 p1 p2 p3 t)
  | (p0,p1,p2,p3) <- earPts1 earPts, t <- [0.0,0.1..1.0]]
```

Olemme esittäneet syntyneen kuvion kuvassa 26.

Toimimme vasemman korvan suhteen samalla periaatteella.

```
earL = [
  (4.02,-3.10), (5.25,-2.31),  (7.95,-2.05),
  (3.36,0.31),  (4.34,5.09),   (0.99,5.20),
```

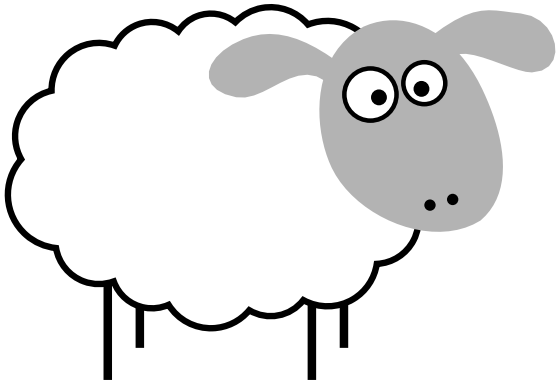


Kuva 26. Kolmen Bezier-käyrän muodostama avoin kuvio.

```
  (-1.57,0.05), (-5.52,-2.36), (-6.92,-1.44) ]
```

Muunnamme viivajonon täytetyksi monikulmioksi. Pienennämme ja siirrämme monikulmion pistejoukon kuvaamalla sen funktioilla `scaleCoords` ja `addCoords`.

Kuvamme on nyt valmis (kuva 27).



Kuva 27. Valmis kuva: Foci-lammas.

```
sheepEars2 = map (Filled . Polygon) [right,left]
  where
    right = map (addCoords ptR . scaleCoords 0.2) right0
    left  = map (addCoords ptL . scaleCoords 0.2) left0
    ptR = Point 1.94 1.85
    ptL = Point 3.61 2.27
    [right0,left0] = map sheepEar [earR,earL]
```

Luku 8

Teksti, sanat ja kirjaimet

8.1 Argumentit komentoriviltä

Kun käynnistämme ohjelman komentotulkissa, voimme antaa sille komentorivillä argumentteja. Argumenttien lukemiseksi tuomme kirjastosta `System.Environment` funktion `getArgs`.

```
import System.Environment (getArgs)
```

Kirjoitamme lyhyen ohjelman `get-arcs.hs`.

```
main = do
  args <- getArgs
  print args
```

Komentoriviltä käynnistettynä ohjelma tulostaa `print`-komennon avulla listan argumenteista.

```
$ runhaskell get-arcs.hs
[]
$ runhaskell get-arcs.hs arg1 arg2 arg3 arg4
["arg1","arg2","arg3","arg4"]
```

8.2 Tyyppikonstruktori `IO`

Funktion `getArgs` tyyppi on `IO [String]`. Tyyppi `IO a` sisältää tyyppimuuttujan `a`, joka voi olla mikä tahansa tyyppi. Kun tyyppimuuttujan `a` arvo on `a = [String]`, saamme tyypin `IO [String]`.

```
> :type getArgs
getArgs :: IO [String]
```

Haskell-kieli vaatii konstruktorin `IO` kaikilta operaatioilta, jotka lukevat syötettä standardisyötevirrasta tai tulostavat standarditulosvirtaan.

Haskell-kääntäjän interaktiivisen tulkin komennolla `:info` näemme, että tyyppi `IO a` on tyyppiluokkien `Monad`, `Functor` ja `Applicative` jäsen, eli sille on määritelty kunkin tyyppiluokan instanssi. Tyyppi `IO a` on myös tyyppiluokan `Monoid` jäsen sillä ehdolla että `a` on sen jäsen.

```
> :info IO
newtype IO a =
...
instance Monad IO
instance Functor IO
instance Applicative IO
instance Monoid a => Monoid (IO a)
```

`Do`-lausekkeen avulla voimme koota yhteen toimenpiteitä monadissa. Sijoitettuamme arvon listaan `args`, nimeämme listan ensimmäisen alkion muuttujaksi `fileName`. `Do`-lausekkeessa tämä toimenpide vaatii avainsanan `let`. Lopuksi kutsumme funktiota `analyse` argumentilla `fileName`.

```
main = do
  args <- getArgs
  let fileName = args !! 0
  analyse fileName
```

Sijoitusnuolta `<-` käytämme, kun luemme arvon monadista. Tarvitsemme sijoitusnuolta aina kun funktion palautusarvolla on monadinen konstruktori, kuten tässä tapauksessa `IO`. Sijoitusnuoli `<-` purkaa arvon muuttujaan, joka on nyt tyyppiä `[String]`.

Määrittelemme seuraavaksi do-lausekkeen avulla funktion analyse.

```
analyse fileName = do
  content <- readFile fileName
  putStrLn "frequency content = "
  putStrLn (count1 content)
```

Funktio saa parametrin fileName, joka on tiedoston nimi merkkijonona. Funktio lukee tiedoston sisällön komennolla readFile muuttujaan content ja tulostaa kaksi merkkijonoa, joista toinen määräytyy funktiokutsun count1 content tuloksena.

Kuten muistamme, saa tulostusfunktio putStrLn argumenttinaan merkkijonon, jonka se tulostaa standarditulosvirtaan. Standarditulosvirtaan tulostaminen merkitsee, että funktion putStrLn palautusarvo on tyyppiä IO a. Kun annamme tyyppimuuttujalle arvon a = () saamme abstraktista tyypistä IO a yksinkertaisimman mahdollisen konkreettisen tyypin IO ()

```
> :t putStrLn
putStrLn :: String -> IO ()
```

Myös funktion analyse palautusarvo on tyyppiä IO (). Funktio koostuu peräkkäisistä lausekkeista, joiden kaikkien palautusarvoilla on konstruktori IO.

```
> :t analyse
analyse :: FilePath -> IO ()
```

8.3 Funktiot readFile, writeFile ja appendFile

Funktiokutsu readFile f lukee tiedoston f ja palauttaa sen sisällön merkkijonona. Funktio saa ensimmäisenä parametrinaan arvon tyyppiä FilePath. Interaktiivisen tulkin komennolla :info saamme selville, että tyyppi FilePath on merkkijonotyypin String synonyymi. Nimi FilePath toimii tässä dokumentoinnin apuvälineenä. Vaikka tyypin FilePath arvo voi olla mikä tahansa merkkijono, kertoo nimi, että järjestelmä odottaa tuon merkkijonon olevan myös mielekäs tiedostopolku.

```
> :t readFile
```

```
readFile :: FilePath -> IO String
> :i FilePath
type FilePath = String
```

Funktiokutsu `writeFile f s` kirjoittaa tiedostoon `f` merkkijonon `s`.

```
> str1 = "Oceanus Procellarum\nMare Frigoris\nMare Imbrium\n"
> writeFile "moon1.txt" str1
```

Funktiokutsu `appendFile f s` lisää tiedostoon `f` merkkijonon `s`.

```
> str2 = "Mare Fecunditatis\n"
> appendFile "moon1.txt" str2
```

Olemme näin tallentaneet tiedostoon `moon1.txt` tekstirivit, jotka esitämme käyttöjärjestelmän komennolla `cat`.

```
> :!cat moon1.txt
Oceanus Procellarum
Mare Frigoris
Mare Imbrium
Mare Fecunditatis
```

8.4 Funktiot `lines` ja `words`

Luemme tekstin funktiokutsulla `readFile "moon1.txt"` muuttujaan `content`.

```
> content <- readFile "moon1.txt"
> content
"Oceanus Procellarum\nMare Frigoris\nMare Imbrium\nMare
 Fecunditatis\n"
```

Muuttuja `content` sisältää nyt merkkijonon, jossa kaikki tiedoston rivit ovat peräjälkeen rivinvaihtosymbolilla `'\n'` (*newline*) erotettuina.

Funktio `lines` palauttaa tekstin rivit listana. Funktio `words` palauttaa tekstin sanat listana.

```
> lines content
```

```
["Oceanus Procellarum","Mare Frigoris","Mare Imbrium",
 "Mare Fecunditatis"]
> words content
["Oceanus","Procellarum","Mare","Frigoris","Mare","Imbrium",
 "Mare","Fecunditatis"]
```

Funktio `words` soveltuu myös välilyöntimerkkien siivoamiseen sanojen ympäriltä.

```
> words "  lacus  "
["lacus"]
```

Molemmat funktiot saavat parametreinaan merkkijonon tyyppiä `String` ja palauttavat merkkijonojen listan tyyppiä `[String]`.

```
> :t lines
lines :: String -> [String]
> :t words
words :: String -> [String]
```

Aakkosmerkkien ulkopuolelle jäävät merkit voimme siivota pois myös määrittelemällä funktion `trim`.

```
trim = dropWhileEnd (not . isAlpha) .
  dropWhile (not . isAlpha)
```

Funktio tarvitsee kirjastosta `Data.Char` funktion `isAlpha` ja kirjastosta `Data.List` funktion `dropWhileEnd`. Funktio `dropWhile` pudottaa alkioita (merkkejä) pois listan (merkkijonon) alusta ja funktio `dropWhileEnd` listan (merkkijonon) lopusta.

```
> import Data.Char
> import Data.List
> trim = dropWhileEnd (not . isAlpha) .
|   dropWhile (not . isAlpha)
> trim "34Excellentiae. "
"Excellentiae"
```

Jos haluamme säilyttää aakkosmerkkien lisäksi myös numerot, käytämme funktion `isAlpha` sijasta funktiota `isAlphaNum`.

```
> trim2 = dropWhileEnd (not . isAlphaNum) .
```

```
|   dropWhile (not . isAlphaNum)
> trim2 " \t8.8 W.\n"
"8.8 W"
```

8.5 Funktiot `unwords` ja `unlines`

Funktiot `unwords` ja `unlines` suorittavat edellä kuvatut operaatiot vastakkaiseen suuntaan (olematta kuitenkaan täydellisiä käänteisfunktioita funktioille `words` ja `lines`).

```
> unwords ["Oceanus","Procellarum","Mare","Frigoris",
    "Mare","Imbrium","Mare","Fecunditatis"]
"Oceanus Procellarum Mare Frigoris Mare Imbrium Mare
 Fecunditatis"
> unlines ["Oceanus Procellarum","Mare Frigoris",
    "Mare Imbrium","Mare Fecunditatis"]
"Oceanus Procellarum\nMare Frigoris\nMare Imbrium\nMare
 Fecunditatis\n"
```

8.6 Kirjainten esiintymistiheyden laskeminen

Haluamme laskea kirjainten esiintymistiheyden Alexandre Dumas'n suomennetusta teoksessa Kolme muskettisoturia. Luemme tekstin funktiolla `readFile`.

```
> content <- readFile "kolme-muskettisoturia.txt"
```

Valitsemme sisällöstä kaikki merkit, jotka ovat kirjaimia. Tätä varten tuomme kirjastosta `Data.Char` funktion `isLetter`. Funktio `isLetter` palauttaa totuusarvon `True` mikäli merkki on Unicode-merkistön aakkosmerkki, muussa tapauksessa se palauttaa totuusarvon `False`.

Tuomme kirjastosta `Data.Char` myös funktion `toUpper`. Funktio `toUpper` muuntaa merkin suuraakkosiksi.

```
> import Data.Char (toUpper,isLetter)
> t = [(toUpper c, 1) | c <- content, isLetter c]
```

```
> length t
1039627
> take 10 t
[('K',1),('O',1),('L',1),('M',1),('E',1),('M',1),('U',1),
 ('S',1),('K',1),('E',1)]
```

Muuttuja `t` sisältää nyt vähän yli miljoonan tietueen listan, jossa kukin tietue koostuu kahdesta alkiosta: kirjaimesta suuraakkosin ja luvusta 1.

Muodostamme seuraavaksi hakupuun listasta `t`. Tätä varten tuomme kirjastosta `Data.Map` funktiot `toList` ja `fromListWith`.

```
> import Data.Map (toList,fromListWith)
```

Funktiokutsu `fromListWith f xs` muodostaa hakupuun `Map` listasta `xs`. Funktio saa parametrinaan funktion `f`, joka on kuvaus siitä, mitä tapahtuu, kun samalle avaimelle on jo olemassa aikaisempi arvo. Edellisessä esimerkissä näin tapahtuu ensimmäisen kerran avaimen `'M'` kohdalla, kun olemme löytäneet listasta alkiot `('M',1)` ja `('M',1)`. Kun nyt aikaisempi arvo on `a = 1` ja uusi arvo `b = 1`, haluamme laskea uuden alkuarvon funktiolla `a + b`, joka on prefix-muodossa `(+) a b` ja parametrittomassa muodossa `(+)`.

M-kirjaimen kohdalla sama toistuu valitsemamme tekstin puitteissa 33839 kertaa. Alkion loppuarvoksi tulee siten `('M',33839)`.

```
> fromListWith (+) t
fromList [('A',120742),('B',937),('C',1400),('D',11062),
...
> list1 = (toList . fromListWith (+)) t
> list1
[('A',120742),('B',937),('C',1400),('D',11062),('E',82849),
...
```

Kirjastossa `Data.Map` on myös funktio `fromList`, jolla ei ole parametria `f`. Funktio `fromList` ei huomioi päällekkäisiä arvoja, vaan viimeksi tullut arvo korvaa aikaisemmin tulleen. Näemme funktioiden `fromList` ja `fromListWith` tyypit vuorovaikutteisen tulkin komennolla `:type`. Huomaamme myös, että yhteenlaskuoperaation `(+)` tyyppi on parametrilta `f` vaadittavaa tyyppiä.

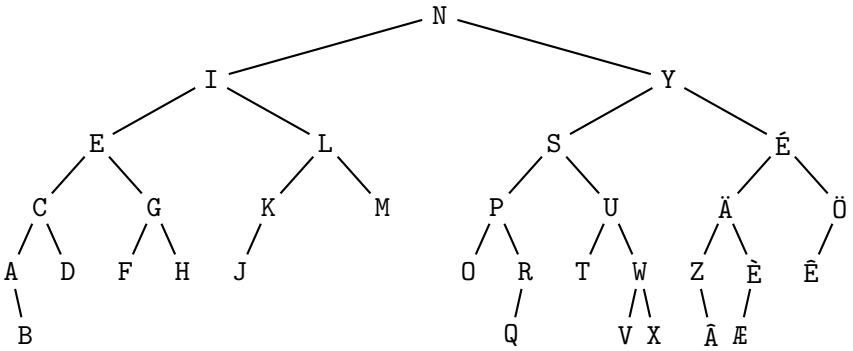
```
> :t fromList
fromList :: Ord k => [(k, a)] -> Map k a
```

```
> :t fromListWith
fromListWith :: Ord k => (a -> a -> a) -> [(k, a)] -> Map k a
> :t (+)
(+) :: Num a => a -> a -> a
> :t fromListWith (+)
fromListWith (+) :: (Num a, Ord k) => [(k, a)] -> Map k a
```

Osittaistoteutettu funktiokutsu `fromListWith (+)` kokoaa tyyppiehdot tyyppimuuttujille `k` ja `a`. Avaimen tyypin `k` tulee olla järjestyvä (*Orderable*) ja arvon tyypin `a` numeerinen (*Numeric*). Tyyppiehdot ovat siten `(Num a, Ord k)`. Nyt siis, sillä ehdolla että tyyppi `k` on järjestyvä ja tyyppi `a` numeerinen, saa funktiokutsu `fromListWith (+)` parametrinaan listan tyyppiä `[(k, a)]` ja palauttaa hakupuun tyyppiä `Map k a`.

Hakupuun tekstuaalinen esitys (tyyppiluokan `Show` instanssi) on määritelty muotoon `fromList [...]`. Voimme esittää tekstin perusteella syntyneen hakupuun avaimet myös graafisesti (kuva 28).

Hakupuun toiminta on yksinkertainen. Vertaamme etsittävää avainta juureen. Jos avain on järjestyksessä ennen juurta, siirrymme vasempaan oksaan. Jos avain on järjestyksessä juuren jälkeen, siirrymme oikeaan oksaan. Toistamme operaation kunnes avain löytyy tai saavutamme oksan kärjen avainta löytämättä.



Kuva 28. Kirjastorutiinin `fromListWith` muodostama hakupuu.

Voimme etsiä avaimeen liitetyn arvon myös funktiolla `(!)`.

```
> map1 = fromListWith (+) t
```

```
> map1 ! 'M'
33839
```

8.7 Pylväsdiagrammin piirtäminen

Laadimme seuraavaksi ohjelman, joka piirtää pylväsdiagrammin kirjainten esiintymistiheydestä. Tarvitsemme muun muassa kirjastot `Data.Map`, `Data.List` ja `Data.Char`.

```
import Data.Map (toList,fromListWith,(!))
import Data.List (sortOn)
import Data.Char (toUpper,isLetter)
```

Luemme esimerkkitekstin tiedostosta. Koska joidenkin kirjainten esiintymistiheys on varsin vähäinen, päätämme huomioida ainoastaan 24 yleisintä kirjainta.

```
fileName = "kolme-muskettisoturia.txt"

count fileName = do
  content <- readFile fileName
  return (fiProb (frequencyC content))

main = do
  c <- count fileName
  putStrLn (tpict (take 24 c))
```

Laskemme kirjainten esiintymistiheyden aiemman esimerkin mukaisesti.

```
frequencyC content = (reverse . sortOn snd) result
  where
    result = (toList . fromListWith (+)) t
    t = [(toLower c, 1) | c <- content, isLetter c]

sortBySnd =  reverse . sortOn snd
sumOfSnd fr = sum [n | (c,n) <- fr]

fiProb cnt =
```

```
  [([c],intToDouble n / intToDouble sm) | (c,n) <- fr]
  where
    sm = sumOfSnd fr
    fr = cnt
```

Piirrämme pylväät määrittelemällä funktion `blocks1`. Funktion ainoa parametri `prob` on lista merkkien esiintymistiheyksistä. Funktiossa `block` piirrämme pylvästä kuvaavan viivajonon, joka alkaa nollatasolta, kulkee pylvään kahden huippupisteen kautta ja päättyy takaisin nollatasolle.

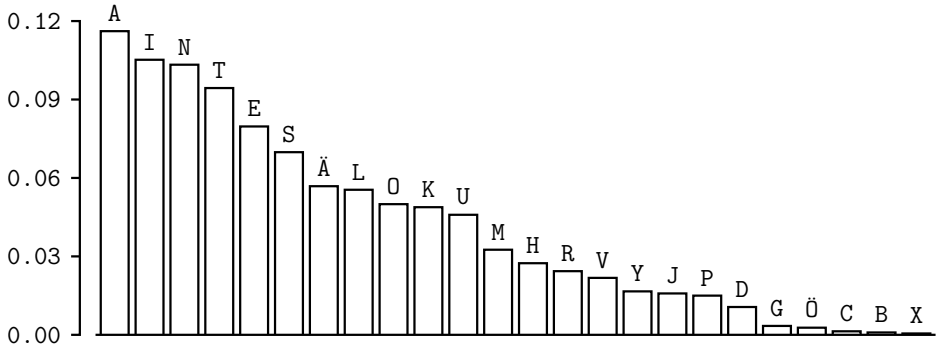
Nollatason viiva `baseline` on pylväille yhteinen, ja piirrämme sen siksi erikseen.

```
block c x1 x2 y1 y2 =
  [Texttt (Point xm y2) c "above"] ++
  [rct]
  where
    rct = PolyLine [p1,p2,p3,p4]
    p1 = Point x1 y1
    p2 = Point x1 y2
    p3 = Point x2 y2
    p4 = Point x2 y1
    xm = (x1+x2) / 2
    ym = (y1+y2) / 2

columns y yts = concat rs ++ [baseline]
  where
    baseline = Line (Point (x1 1 - baseExtra*eps) 0)
      (Point (x2 (length yts) + baseExtra*eps) 0)
    rs = [block c (x1 xk) (x2 xk) 0 (bscale * y2)
      | (xk,(c,y2)) <- zip [1..] yts]
    x1 xk = intToDouble xk * eps - (bsize*eps)
    x2 xk = intToDouble xk * eps + (bsize*eps)
    baseExtra = 0.15
    bsize = 0.40
    bscale = 3.00
    eps = 1.00 / (n + 1)
    n = intToDouble (length yts)
```

```
blocks1 prob =
  columns 0.0 prob
```

Piirrämme mittakaavajanan kuvion vasemmalle puolelle määrittelemällä funktion `scale1`. Funktion ensimmäinen viivajono sisältää mittakaavajanan pystyviivan ja uloimmat sakarat, seuraava listamuodostimella muodostettu viivojen lista sisemmät sakarat ja viimeinen tekstien lista jakovälit numeroina.

Olemme esittäneet valmiin kuvion kuvassa 29.



Kuva 29. Yleisimmät 24 kirjainta esiintymistiheyksineen Alexandre Dumas's suomennetussa teoksessa Kolme muskettisoturia.

```
scale x y dxs dys = [ PolyLine [
    sPoint (x + minimum dxs) (y + maximum dys),
    sPoint (x + maximum dxs) (y + maximum dys),
    sPoint (x + maximum dxs) (y + minimum dys),
    sPoint (x + minimum dxs) (y + minimum dys)] ] ++
  [Line
    (sPoint (x + minimum dxs) (y + d))
    (sPoint (x + maximum dxs) (y + d))
    | d <- (tail . init) dys] ++
  [Texttt (sPoint x (y + dy))
    (show2 dy) "left" | dy <- dys]
  where
```

```
    sPoint x1 y1 = Point x1 (bscale * y1)
    bscale = 3.00

scale1 =
  scale (-0.01) 0.0 [0,0.01] [0.00,0.03 .. 0.12]
```

Edellä kuvattuja menetelmiä käyttäen saamme myös kirjainjonon, jossa otoksessa esiintyvät kirjaimet ovat esiintymistiheyden mukaisessa järjestyksessä.

```
> content <- readFile "kolme-muskettisoturia.txt"
> putStrLn [c | (c,n) <- frequencyC content]
AINTESÄLOKUMHRVYJPDGÖCBXÉFWQZÈÊÂÆ
```

Sanojen esiintymistiheyden laskemiseen määrittelemme funktion `frequencyW`.

```
frequencyW content = (size1,size2,result)
  where
    size2 = length result
    size1 = length t
    result = (toList . fromListWith (+)) t
    t = [(w, 1) | w <- trimmed, not (null w)]
    trimmed = (map trim) ws
    ws = words c1
    c1 = (map toLower) content
```

Tässä muuttuja `size1` palauttaa sanojen yhteismäärän ja muuttuja `size2` eri sanojen määrän.

Jos haluamme muotoilla numerot kolmen ryhmiksi, voimme määritellä funktion `show3g`.

```
show3g n = reverse ic
  where
    ic = intercalate " " (chunksOf 3 r1)
    r1 = reverse (show n)
```

Perinteisen aakkosjärjestyksen sijasta voimme käyttää myös aakkosjärjestyksenä kirjainten esiintymistiheyttä.

```
let charCnt = countC content
    result1 = sortByContent fr1 charCnt

countC content = [c | (c,n) <- frequencyC content]

sortByContent fr charCnt = sortOn fun fr
  where
    elmIndex x =
      elemIndex (toLower x) charCnt
    fun (x,y) = [Just ((-1) * y)] ++ map elmIndex x
```

Määrittelemällä funktion `printStats` muotoilemme tulosteen sanojen yhteismäärästä.

```
printStats size1 size2 = do
  putStrLn ("\nYhteensä **" ++ show3g size1 ++
    "** sanaa, joista **" ++ show3g size2 ++
    "** erilaista.\n")
```

Tulokseksi saamme seuraavan taulukon:

1	ja	5 101	21	te	713	41	siis	412
2	hän	3 505	22	kun	702	42	minua	390
3	oli	2 680	23	häntä	604	43	vielä	389
4	on	2 145	24	minun	580	44	aramis	388
5	minä	1 704	25	teidän	562	45	siitä	381
6	että	1 614	26	nyt	538	46	hänet	381
7	hänen	1 571	27	ole	510	47	jo	378
8	sanoi	1 499	28	jos	495	48	porthos	378
9	d'artagnan	1 449	29	de	489	49	hänelle	374
10	ei	1 330	30	sitä	475	50	olivat	367
11	niin	1 281	31	olen	471	51	minulle	343
12	mutta	1 221	32	sillä	464	52	jotka	334
13	herra	1 156	33	mylady	456	53	sinä	332
14	joka	1 113	34	olisi	447	54	mitään	332
15	kuin	958	35	tuo	440	55	sitten	320
16	se	913	36	rouva	437	56	tai	317
17	vaan	891	37	kaikki	431	57	mies	309
18	sen	798	38	jonka	431	58	he	308
19	mitä	750	39	ollut	427	59	niinkuin	307
20	athos	739	40	en	423	60	huudahti	301

Yhteensä **160 367** sanaa, joista **28 364** erilaista.

Luku 9

Pallogeometriaa

9.1 Kuun mitat ja pinnanmuodot

Kuun keskimääräinen säde on 1737.1 kilometriä. Suurimmat pinnanmuodot ovat varhaisia törmäyskraattereita. Kappaleen törmätessä kuun pintaan sula basalttinen laava täytti kraatterin muodostaen tumman tasaisen alangon, joita nykyisin nimitämme kuun meriksi (*maria*), järviksi (*lacus*), lahdiksi (*sinus*) ja soiksi (*paludes*).

Oletamme, että käytössämme on kuvan 30 mukainen listaus kuun merkittävimmistä pinnanmuodoista tekstitiedostona. Kukin rivi koostuu tabulaattorimerkein erotetuista kentistä. Kentät ovat muodostuman latinankielinen nimi, suomenkielinen nimi, latitudi, longitudi ja halkaisija.

9.2 Funktio `splitOn`

Kun annamme muuttujan `str` arvoksi esimerkkirivin tiedostosta, voimme jakaa merkkijonon osiin kirjaston `Data.List.Split` funktiolla `splitOn`. Funktio saa argumentteinaan katkaisevan ja katkaistavan merkkijonon. Funktio palauttaa listan syntyneistä merkkijonon osista.

```
> import Data.List.Split
```

```
Oceanus Procellarum    Myrskyjen valtameri      18.4 N    57.4 W   2568
Mare Frigoris          Kylmyyden meri           56.0 N     1.4 E   1596
Mare Imbrium           Sateiden meri            32.8 N    15.6 W   1123
Mare Fecunditatis      Hedelmällisyyden meri     7.8 S    51.3 E    909
Mare Tranquillitatis   Rauhallisuuden meri       8.5 N    31.4 E    873
Mare Nubium            Pilvien meri             21.3 S    16.6 W    715
Mare Serenitatis       Hiljaisuuden meri        28.0 N    17.5 E    707
Mare Australe          Eteläinen meri           38.9 S    93.0 E    603
Mare Insularum         Saarten meri              7.5 N    30.9 W    513
Mare Marginis          Reunameri                13.3 N    86.1 E    420
Mare Crisium           Vaarojen meri            17.0 N    59.1 E    418
Mare Humorum           Kosteuden meri           24.4 S    38.6 W    389
Mare Cognitum          Tunnettu meri            10.0 S    23.1 W    376
Mare Smythii           Smythin meri              1.3 N    87.5 E    373
Mare Nectaris          Nektarinmeri             15.2 S    35.5 E    333
Mare Orientale         Itäinen meri             19.4 S    92.8 W    327
Mare Ingenii           Nerokkuuden meri         33.7 S   163.5 E    318
Mare Moscoviense       Moskovan meri            27.3 N   147.9 E    277
Mare Humboldtianum     Humboldtin meri          56.8 N    81.5 E    273
Mare Vaporum           Höyryjen meri            13.3 N     3.6 E    245
Mare Undarum           Aaltojen meri             6.8 N    68.4 E    243
Mare Anguis            Käärmeitten meri         22.6 N    67.7 E    150
Mare Spumans           Vaahdon meri              1.1 N    65.1 E    139
Lacus Veris            Kevään järvi             16.5 S    86.1 W    396
Lacus Somniorum        Unelmien järvi           38.0 N    29.2 E    384
Lacus Excellentiae     Erinomaisuuden järvi     35.4 S    44.0 W    184
Lacus Autumni          Syksyn järvi              9.9 S    83.9 W    183
Lacus Mortis           Kuoleman järvi           45.0 N    27.2 E    151
Lacus Solitudinis      Yksinäisyyden järvi      27.8 S   104.3 E    139
Lacus Temporis         Ajan järvi               45.9 N    58.4 E    117
Lacus Timoris          Pelon järvi              38.8 S    27.3 W    117
Lacus Gaudii           Ilon järvi               16.2 N    12.6 E    113
Lacus Doloris          Kärsimyksen järvi        17.1 N     9.0 E    110
Lacus Bonitatis        Hyvyyden järvi           23.2 N    43.7 E     92
Lacus Aestatis         Kesän järvi              15.0 S    69.0 W     90
Lacus Felicitatis      Onnellisuuden järvi      19.0 N     5.0 E     90
Lacus Lenitatis        Pehmeyden järvi          14.0 N    12.0 E     80
Lacus Spei             Toivon järvi             43.0 N    65.0 E     80
Lacus Odii             Vihan järvi              19.0 N     7.0 E     70
Lacus Perseverantiae   Sinnikkyyden järvi        8.0 N    62.0 E     70
Lacus Hiemalis         Talven järvi             15.0 N    14.0 E     50
Lacus Luxuriae         Ylellisyyden järvi       19.0 N   176.0 E     50
Lacus Oblivionis       Unohduksen järvi         21.0 S   168.0 W     50
Sinus Medii            Keskilahti                2.4 N     1.7 E    335
Sinus Aestuum          Helteen lahti            10.9 N     8.8 W    290
Palus Epidemiarum      Tautien suo              32.0 S    28.2 W    286
Sinus Iridum           Sateenkaarten lahti      44.1 N    31.5 W    236
Sinus Asperitatis      Kovuuden lahti            3.8 S    27.4 E    206
Sinus Roris            Aamukasteen lahti        54.0 N    56.6 W    202
Palus Putredinis       Mätänemisen suo          26.5 N     0.4 E    161
Palus Somni            Unien suo                14.1 N    45.0 E    143
Sinus Concordiae       Sopusoinnun lahti        10.8 N    43.2 E    142
Sinus Successus        Menestyksen lahti         0.9 N    59.0 E    132
Sinus Amoris           Rakkauden lahti          18.1 N    39.1 E    130
Sinus Lunicus          Lunan lahti              31.8 N     1.4 W    126
Sinus Honoris          Kunnian lahti            11.7 N    18.1 E    109
Sinus Fidei            Luottamuksen lahti       18.0 N     2.0 E     70
```

Kuva 30. Kuun merkittävimmät pinnanmuodot tekstitiedostona. Kentät on erotettu tabulaattorimerkein.

```
> str = "Mare Smythii\tSmythin meri\t1.3 N\t87.5 E\t373"
> splitOn "\t" str
["Mare Smythii","Smythin meri","1.3 N","87.5 E","373"]
```

9.3 Pallokoordinaatisto

Voimme muuntaa koordinaatteja pallokoordinaatistosta karteesiseen koordinaatistoon kaavalla (http://mathworld.wolfram.com/SphericalCoordinates.html)

$$x = r\cos\theta\sin\phi$$
$$y = r\sin\theta\sin\phi$$
$$z = r\cos\phi$$

Tässä r on säde eli etäisyys origosta, θ kulma x-akselista xy-tasossa ja ϕ kulma ylöspäin osoittavasta z-akselista.

Maantieteellisessä koordinaatistossa merkitsemme leveysastetta (*latitudi*) symbolilla δ, jolloin $\phi = 90° - \delta$ sekä pituusastetta (*longitudi*) symbolilla λ ($\lambda = \theta$).

Koordinaattilyhenteissä kirjain N (*north*) merkitsee pohjoista leveyttä, S (*south*) eteläistä leveyttä, E (*east*) itäistä pituutta ja W (*west*) läntistä pituutta. Maapallolla leveysaste δ kasvaa päiväntasaajalta pohjoiseen kuljettaessa ja pituusaste λ Greenwichin nollameridiaanilta itään kuljettaessa. Nimitykset leveys ja pituus juontuvat Välimeren alueen kulttuureista: Välimeri on ”pitkä” itä-länsi-suunnassa ja ”leveä” pohjois-etelä-suunnassa. Pituuspiirejä sanotaan myös meridiaaneiksi. Termi meridiaani johtuu latinan puolipäivää tai etelää merkitsevästä sanasta *meridies*.

Määrittelemme tietotyypin `Point3D` pisteelle kolmiulotteisessa karteesisessa xyz-koordinaatistossa. Pallokoordinaatistossa määrittelemme pisteen `Spheric3D` kulmien θ ja ϕ avulla.

```
-- | Point3D x y z, RH cartesian coordinates
data Point3D = Point3D Double Double Double

-- |  Spheric3D theta phi, where phi =
```

```
-- polar angle measured from a fixed zenith direction
data Spheric3D = Spheric3D Angle Angle
```

Asetamme kuun säteeksi $r = 1737.1$ km. Yksinkertaisimman muunnoksen kolmiulotteisesta koordinaatistosta kaksiulotteiseen koordinaatistoon saamme pudottamalla x-koordinaatin pois.

```
r = 1737.1

dropX (Point3D x y z) = Point y z

perspective = dropX

cartesian (Spheric3D lambda delta) = Point3D x y z
  where
    x = r * cos1 theta * sin1 phi
    y = r * sin1 theta * sin1 phi
    z = r * cos1 phi
    theta = lambda
    phi = (DEG 90) `subAngles` delta
```

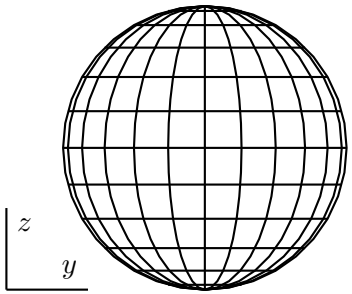
Saamme hahmotelman leveyspiireistä pallon etupuoliskolla algoritmilla

```
latitudes = [PolyLine [(perspective . cartesian)
  (Spheric3D (DEG l) (DEG d))
  | l <- lambda]
    | d <- delta]
  where
    delta = [-90,-75..90]
    lambda = [-90,-70..90]
```

Etupuoliskon pituuspiirit eli meridiaanit saamme algoritmilla

```
meridians = [PolyLine [(perspective . cartesian)
  (Spheric3D (DEG l) (DEG d))
  | d <- delta]
    | l <- lambda]
  where
    delta = [-90,-80..90]
    lambda = [-90,-75..90]
```

Olemme esittäneet leveys- ja pituuspiirien muodostaman kuvion kuvassa 31.



Kuva 31. Karttapallon puolisko, jossa kuvattuna leveyspiirit ja pituuspiirit eli meridiaanit 15 asteen välein.

9.4 Vinoprojektion perspektiivimatriisi

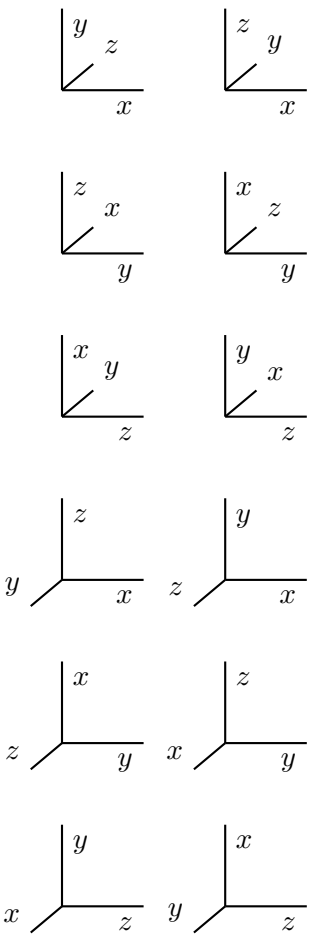
Niin sanotussa *vinoprojektiossa* kuvaamme kaksi akselia suoraan kulmaan toistensa kanssa ja kolmannen akselin tiettyyn kulmaan näiden välillä.

Kuvassa 32 esiintyvät vakiot a, b, c ja d olemme määritelleet seuraavasti kulman α avulla:

$$\begin{aligned} a &= {}^1\!/_2 \cdot \cos\alpha \\ b &= {}^1\!/_2 \cdot \sin\alpha \\ c &= -a \\ d &= -b \end{aligned}$$

Matriiseista järjestysluvultaan parittomat antavat kuvauskoordinaatistoksi vasenkätisen ja parilliset oikeakätisen koordinaatiston. Haskell-kielelle muunnettuna voimme esittää perspektiivimatriisit $M_{1..12}$ `case`-lauseen avulla.

```
matrix1 m alpha = case m of
  1 -> [ [1,0,a], [0,1,b], z]
  2 -> [ [1,a,0], [0,b,1], z]
  3 -> [ [a,1,0], [b,0,1], z]
  4 -> [ [0,1,a], [1,0,b], z]
```



$$M_1 = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 0 \end{pmatrix} \quad M_2 = \begin{pmatrix} 1 & a & 0 \\ 0 & b & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

$$M_3 = \begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad M_4 = \begin{pmatrix} 0 & 1 & a \\ 1 & 0 & b \\ 0 & 0 & 0 \end{pmatrix}$$

$$M_5 = \begin{pmatrix} 0 & a & 1 \\ 1 & b & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_6 = \begin{pmatrix} a & 0 & 1 \\ b & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$M_7 = \begin{pmatrix} 1 & c & 0 \\ 0 & d & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad M_8 = \begin{pmatrix} 1 & 0 & c \\ 0 & 1 & d \\ 0 & 0 & 0 \end{pmatrix}$$

$$M_9 = \begin{pmatrix} 0 & 1 & c \\ 1 & 0 & d \\ 0 & 0 & 0 \end{pmatrix} \quad M_{10} = \begin{pmatrix} c & 1 & 0 \\ d & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

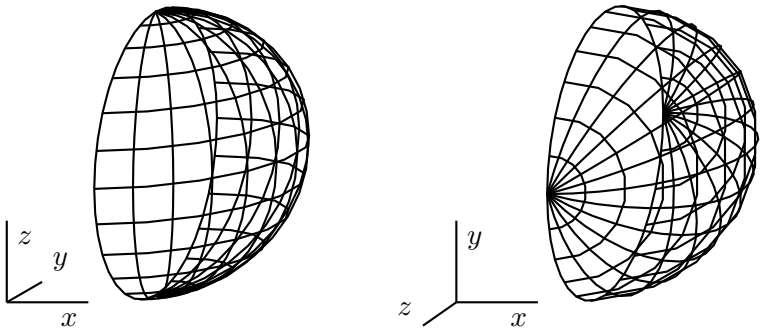
$$M_{11} = \begin{pmatrix} c & 0 & 1 \\ d & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad M_{12} = \begin{pmatrix} 0 & c & 1 \\ 1 & d & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Kuva 32. Vinoprojektioiden perspektiivimatriiseja.

```
  5 -> [ [0,a,1], [1,b,0], z]
  6 -> [ [a,0,1], [b,1,0], z]
  7 -> [ [1,c,0], [0,d,1], z]
  8 -> [ [1,0,c], [0,1,d], z]
  9 -> [ [0,1,c], [1,0,d], z]
  10 -> [ [c,1,0], [d,0,1], z]
  11 -> [ [c,0,1], [d,1,0], z]
  12 -> [ [0,c,1], [1,d,0], z]
  where
    a = 0.5 * cos1 alpha
    b = 0.5 * sin1 alpha
    c = -a; d = -b
    z = [0,0,0]
```

Asetamme nyt muunnosmatriiseiksi matriisit M_2 ja M_8 (kuva 33).

$$M_2 = \begin{pmatrix} 1 & 1/2 \cdot \cos 30^\circ & 0 \\ 0 & 1/2 \cdot \sin 30^\circ & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad M_8 = \begin{pmatrix} 1 & 0 & -1/2 \cdot \cos 35^\circ \\ 0 & 1 & -1/2 \cdot \sin 35^\circ \\ 0 & 0 & 0 \end{pmatrix}$$



Kuva 33. Karttapallon puoliskot muunnosmatriiseja M_2 ja M_8 käyttäen.

```
matrixTimes3 a b =
  [sum [x * y | (x,y) <- zip a1 b] | a1 <- a]

matr1 pv pAlpha (Point3D x1 y1 z1) = Point x y
  where
```

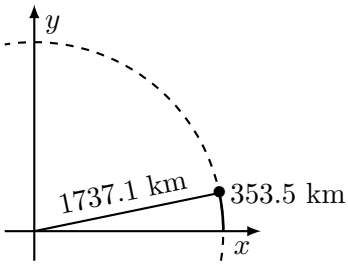
```
    [x,y,z] = matrixTimes3 (matrix1 pv alpha) [x1,y1,z1]
    alpha = DEG pAlpha

r = 1737.1

dropX (Point3D x y z) = Point y z

perspective = matr1 pv pAlpha
  where
    pv = 8
    pAlpha = 35
```

9.5 Mare Serenitatis

Haluamme seuraavaksi kuvata Hiljaisuuden meren karttapallolle. Kraatterin läpimitta on $d = 707$ km, ja säde näin ollen $r = d/2 = 353.5$ km. Hiljaisuuden meren keskipisteen koordinaatit ovat (28.0° N, 17.5° E).

Kuun säteen ollessa $r = 1737.1$ km, saamme kuvan 34 merkinnöillä keskuskulmaksi

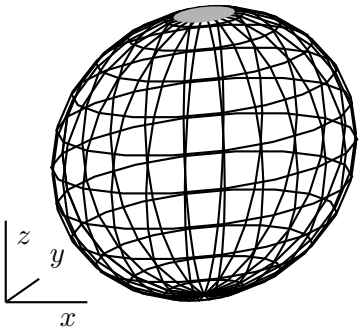
$$\theta = 2\pi \cdot \frac{353.5}{2\pi \cdot 1737.1} = 0.2035 \text{ rad} = 11.65^\circ$$



Kuva 34. Kraatterin säteen $r = 353.5$ km muodostama keskuskulma.

Aiemmin esitellyn perusteella osaamme jo sijoittaa pohjoisnavalle ympyrän, jonka säde on annettu. Käytämme tällöin Hiljaisuuden merelle laske-

maamme keskuskulmaa ylimääräisenä leveyspiirinä, jonka piirrämme täytettynä monikulmiona (kuva 35).



Kuva 35. Hiljaisuuden meri pohjoisnavalle sijoitettuna.

```
serenitatis = [ Filled $ Polygon [
  (perspective . cartesian) (Spheric3D (DEG l) t)
  | l <- lambda]]
  where
    t = RAD (halfpi - 0.2035)
    lambda = [-180,-160..160]
```

9.6 Kiertomatriisit kolmessa ulottuvuudessa

Kiertomatriisit kolmessa ulottuvuudessa kulman θ verran akselien x, y ja z suhteen ovat (https://en.wikipedia.org/wiki/Rotation_matrix)

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \qquad R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Haskell-kielelle muunnettuna nämä ovat

```
rotationX t = [
  [1,      0,       0],
  [0, cos1 t, -sin1 t],
  [0, sin1 t,  cos1 t]
  ]

rotationY t = [
  [cos1 t,  0, sin1 t],
  [0,       1,      0],
  [-sin1 t, 0, cos1 t]
  ]

rotationZ t = [
  [cos1 t, -sin1 t, 0],
  [sin1 t,  cos1 t, 0],
  [     0,       0, 1]
  ]
```

Latitudin δ komplementtikulma $\phi = 90° - \delta$ määrittää kierron y-akselin suhteen ja longitudi λ kierron z-akselin suhteen.

```
rotYZ delta lambda (Point3D x1 y1 z1) = Point3D x y z
  where
    [x,y,z] = foldr matrixTimes3 [x1,y1,z1] rts
    rts = [rotationZ lambda,rotationY phi]
    phi = DEG 90 `subAngles` delta
```

Valitsemme perspektiivimatriisin M_{10}.

```
perspective = matr1 pv pAlpha
  where
    pv = 10
    pAlpha = 35
```

Yleisessä muodossaan määrittelemme kuun meren piirtoalgoritmin funktiossa `mare`, joka saa parametrinaan `d` meren halkaisijan ja parametrinaan `pos` maantieteellisen pohjois-itä-koordinaatin tyyppiä `GeographicNE`.

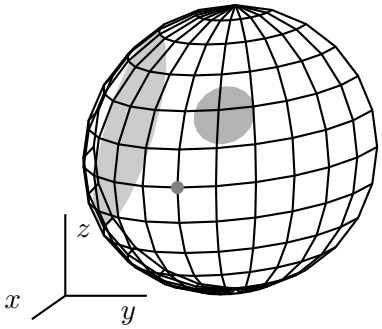
```
data GeographicNE = GeographicNE Angle Angle
```

```
mare d pos = [ Filled $ Polygon [
  (perspective .  rotYZ delta lambda . cartesian)
    (Spheric3D (DEG l) phi)
  | l <- lambdaRim]]
  where
    GeographicNE delta lambda = pos
    phi = DEG 90 `subAngles` (RAD theta)
    theta = (d/2) / r
    lambdaRim = [-180,-160..160]
```

Hiljaisuuden meri saa nyt muodon

```
serenitatis = mare d pos
  where
    d = 707
    pos = GeographicNE (DEG 28) (DEG 17.5)
```

Piirrämme kuvaan 36 myös Myrskyjen valtameren, jonka halkaisija on $d = 2568$ km, ja jonka keskipiste sijaitsee pisteessä (18.4° N, 57.4° W).



Kuva 36. Hiljaisuuden meri ja Myrskyjen valtameri.

```
procellarum = mare d pos
  where
    d = 2568
    pos = GeographicNE (DEG 18.4) (DEG (-57.4))
```

Merkitsemme myös koordinaatiston nollapisteen funktiolla `proto0`.

```
proto0 = mare 160 (GeographicNE (DEG 0) (DEG 0))
```

9.7 Monikulmion paloittelu

Olemme koordinaattimuunnoksissa huomioineet ainoastaan täytetyn monikulmion reunapisteet, joten esimerkiksi Myrskyjen valtameren keskiosat piirtyivät väärin kuvassa 36.

Parempaan tulokseen päädymme paloittelemalla monikulmiot asteverkon mukaisesti. Käytämme aluksi tasavälistä lieriöprojektiota (*equirectangular projection*), jossa pituus- ja leveysasteet kuvautuvat sellaisenaan koordinaattipisteiksi.

```
equirectangular (Spheric3D lambda delta) = Point l d
  where
    DEG l = degrees lambda
    DEG d = degrees delta
```

Mittakaavakertoimena on seuraavassa $\frac{2\pi \cdot r}{360}$, missä $r = 1737.1$ km on kuun säde.

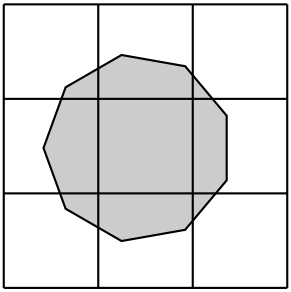
```
marePts d pos = [ pt0 `addCoords`
  pointFromPolar (DEG l) r2 | l <- lambdaRim ]
  where
    r2 = (d/2) / (twopi * r / 360)
    pt0 = equirectangular (Spheric3D lambda delta)
    GeographicNE delta lambda = pos
    lambdaRim = [-180,-140..140]
```

Muunnamme polaarikoordinaatit pisteeksi `Point` funktiolla `pointFromPolar`.

```
pointFromPolar t s = Point x y
  where
    x = s * cos1 t
    y = s * sin1 t
```

Olemme kuvassa 37 esittäneet suorakulmaisessa koordinaatistossa Hedelmällisyyden meren, jonka halkaisija on $d = 909$ km, ja jonka keskipiste sijaitsee pisteessä (7.8° S, 51.3° E).

```
fecunditatis = marePts d pos
  where
```



Kuva 37. Hedelmällisyyden meri suorakulmaisessa koordinaatistossa.

```
d = 909
pos = GeographicNE (DEG (-7.8)) (DEG 51.3)
```

9.8 Pisteet monikulmion sisä- ja ulkopuolella

Käytämme monikulmion paloitteluun *Sutherland-Hodgmanin algoritmia* (https://en.wikipedia.org/wiki/Sutherland-Hodgman_algorithm). Paloittelussa muokkaamme monikulmion kärkipistejoukkoa vertaamalla sitä leikkaavan monikulmion sivuihin sivu kerrallaan. Leikkauksen lähtöjoukkona toimii aina edellisessä vaiheessa saatu kärkipistejoukko. Kukin sivu leikkaa osan kärkipisteistä pois sekä muodostaa uusia kärkipisteitä paloiteltavan ja leikkaavan monikulmion sivujen leikkauspisteisiin. Paloittelualgoritmia varten tarvitsemme tiedon siitä, kummalla puolella annettua sivua tietty kärkipiste sijaitsee.

Saamme selville kummalla puolella sivua piste sijaitsee muodostamalla kolmion, jonka kärkipisteet ovat sivun alkupiste, sivun loppupiste ja vertailtava piste. Kun piste sijaitsee sivun oikealla puolella, muodostuneen kolmion kiertosuunta on myötäpäivään, jolloin sen ala determinanttisäännön mukaan on negatiivinen. Kun piste sijaitsee sivun vasemmalla puolella, kiertosuunta on vastapäivään ja muodostuneen kolmion ala positiivinen.

```
data InOut = In | Out
  deriving Show
```

```
sign x = if x < 0 then (-1) else 1
around xs = zip xs ((tail . cycle) xs)

inOut1 p1 p2 pts = [
  (inOut . sign . area . Polygon) [p1,p2,p3] | p3 <- pts]
  where
    inOut   1  = In
    inOut (-1) = Out

gridGreatCircles = concat [[[
  xpt l1 d1, xpt l2 d1, xpt l2 d2, xpt l1 d2]
  | (d1,d2) <- zip vb3 (tail vb3)]
    | (l1,l2) <- zip vb2 (tail vb2)]
  where
    xpt l d = equirectangular (Spheric3D (DEG l) (DEG d))
    vb3 = visible3 delta
    vb2 = visible2 lambda
    delta = [-90,-75..90]
    lambda = [-90,-75..90]

visible2 = filter (\l -> l > 29 && l < 76)
visible3 = filter (\d -> d > -31 && d < 16)
```

Determinanttisääntöä käytämme, kun määrittelemme funktion `area` monikulmiolle `Polygon`. Determinantin saamme matematiikasta tutulla kaavalla

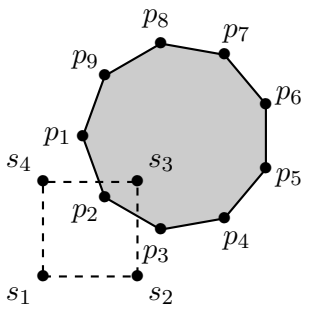
$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = a \cdot d - b \cdot c$$

```
-- | http://mathworld.wolfram.com/PolygonArea.html
-- Polygon area: vertices counterclockwise
area (Polygon pts) = half (sum [det [[x1,y1],[x2,y2]]
  | (Point x1 y1,Point x2 y2)
    <- zip pts (tail pts ++ take 1 pts)])

half = (0.5 *)
det [[a,b],[c,d]] = a * d - b * c
```

Aloitamme kuvion paloittelun neliöstä $(s_1 s_2 s_3 s_4)$ alueen vasemmassa ala-

nurkassa (kuva 38).

```
[s1,s2,s3,s4] = gridGreatCircles !! 0
fc0 = fecunditatis
```



Kuva 38. Ensimmäinen tarkasteltava ruutu.

Rajattavan monikulmion sivut jakautuvat neljään ryhmään suhteessa rajaavaan monikulmioon:

`(In,In)`: sivu alkaa sisäpuolelta ja päättyy sisäpuolelle.
`(In,Out)`: sivu alkaa sisäpuolelta ja päättyy ulkopuolelle.
`(Out,Out)`: sivu alkaa ulkopuolelta ja päättyy ulkopuolelle.
`(Out,In)`: sivu alkaa ulkopuolelta ja päättyy sisäpuolelle.

Sutherland-Hodgmanin algoritmin mukaiset toimenpiteet sivutyypeille ovat

`(In,In)`: säilytämme kärkipisteet.
`(In,Out)`: säilytämme lähtöpisteen ja siirrämme loppupisteen.
`(Out,Out)`: poistamme kärkipisteet.
`(Out,In)`: siirrämme alkupisteen ja säilytämme loppupisteen.

Haskell-kielelle muunnettuna saamme uudet kärkipisteet monikulmiosta `fc` suoran `(s1,s2)` suhteen funktiokutsulla `nextGen fc s1 s2`.

```
nextGen fc s1 s2 = concat [new i1 i2 p1 p2
  | ((i1,i2),(p1,p2)) <- zip io2 pts2]
  where
    io1 = inOut1 s1 s2 fc
    io2 = around io1
    Polygon pts1 = fc
```

```
pts2 = around pts1
new In In p1 p2 = [p1]
new In Out p1 p2 = [p1,
  fromJust (intersection s1 s2 p1 p2)]
new Out Out p1 p2 = []
new Out In p1 p2 = [
  fromJust (intersection s1 s2 p1 p2)]
```

Ensimmäinen rajaava suora on neliön alareuna $s_1 s_2$. Monikulmion `fc0` kaikki kärkipisteet kuuluvat alueen sisäpuolelle.

```
> io1 = inOut1 s1 s2 fc0
> io1
[In,In,In,In,In,In,In,In,In]
```

Monikulmion kaikki pisteet kuuluvat luokkaan `(In,In)`, joten alareuna säilyttää kaikki monikulmion pisteet $(p_1 \cdots p_9)$.

```
> around io1
[ (In,In),(In,In),(In,In),(In,In),(In,In),
  (In,In),(In,In),(In,In),(In,In) ]
```

Saamme uuden pistejoukon `fc1` funktiokutsulla `nextGen fc0 s1 s2`.

```
fc1 = nextGen fc0 s1 s2
```

Toinen rajaava suora on neliön oikea reuna $s_2 s_3$. Nyt rajaavan suoran vasemmalle puolelle eli alueen sisäpuolelle jäävät monikulmion pisteet $(p_1\, p_2\, p_9)$. Alueen ulkopuolelle jäävät monikulmion pisteet $(p_3 \cdots p_8)$.

```
> io2 = inOut1 s2 s3 fc1
> io2
[In,In,Out,Out,Out,Out,Out,Out,In]
```

Monikulmion sivut suhteessa rajaavan neliön oikeaan reunaan $s_2 s_3$ kuuluvat nyt seuraaviin luokkiin:

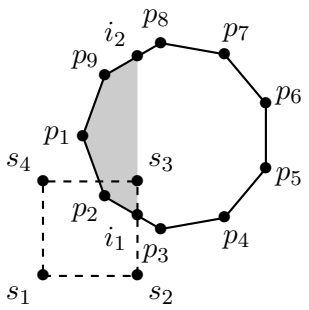
```
> around io2
[ (In,In),(In,Out),(Out,Out),(Out,Out),(Out,Out),
  (Out,Out),(Out,Out),(Out,In),(In,In) ]
```

Luokat `(In,Out)` ja `(Out,In)` tuottavat uuden kärkipisteen i_1 suorien $s_2 s_3$

ja $p_2\,p_3$ leikkauspisteeseen sekä pisteen i_4 suorien $s_2 s_3$ ja $p_8\,p_9$ leikkauspisteeseen.

```
i1 = intersection s2 s3 p2 p3
i2 = intersection s2 s3 p8 p9
```

Kuvan 39 merkinnöillä suoran $s_2 s_3$ suhteen leikattu toinen monikulmio koostuu kärkipisteistä $(p_1\,p_2\,i_1\,i_2\,p_9)$.



Kuva 39. Toinen leikkaus antaa monikulmion kärkipisteet $(p_1\,p_2\,i_1\,i_2\,p_9)$.

Saamme nyt uuden pistejoukon `fc2` funktiokutsulla `nextGen fc1 s2 s3`.

```
fc2 = nextGen fc1 s2 s3
```

Kolmas leikkaus tapahtuu suoran $s_3 s_4$ suhteen edellä saadulle kärkipistejoukolle $(i_1\,p_2\,p_3\,p_4\,i_2)$.

```
fc3 = nextGen fc2 s3 s4
```

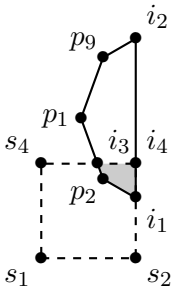
Saamme pisteväleille uudet luokat

```
> io3 = inOut1 s3 s4 fc2
> io3
[Out,In,In,Out,Out]
> around io3
[(Out,In),(In,In),(In,Out),(Out,Out),(Out,Out)]
```

Tässä luokat `(Out,In)` ja `(In,Out)` tuottavat uuden kärkipisteen i_3 suorien $s_3 s_4$ ja $p_1\,p_2$ leikkauspisteeseen sekä pisteen i_4 suorien $s_3 s_4$ ja $i_1\,i_2$ leikkauspisteeseen.

```
i3 = intersection s3 s4 p1 p2
i4 = intersection s3 s4 i1 i2
```

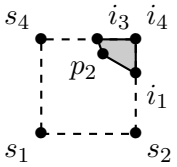
Leikattu monikulmio koostuu nyt kärkipisteistä $(i_3\, p_2\, i_1\, i_4)$ (kuva 40).



Kuva 40. Kolmas leikkaus antaa monikulmion kärkipisteet $(i_3\, p_2\, i_1\, i_4)$.

Viimeinen leikkaus tapahtuu suoran $s_4 s_1$ suhteen. Leikkaus säilyttää kärkipistejoukon $(i_3\, p_2\, i_1\, i_4)$ sellaisenaan (kuva 41).

```
fc3 = nextGen fc2 s3 s4
```



Kuva 41. Viimeinen leikkaus säilyttää kärkipistejoukon $(i_3\, p_2\, i_1\, i_4)$ sellaisenaan.

Luku 10

Gtk-käyttöliittymä

Tässä luvussa tutustumme Gtk-käyttöliittymän kirjoittamiseen. Gtk (*the Gimp toolkit*) on suosittu käyttöliittymäkirjasto, joka aikoinaan syntyi Gimp-kuvankäsittelyohjelman komponenttikirjastona, ja joka nykyisin toimii useimpien Linux-koneiden työpöytäympäristön perusrakennusosana.

10.1 Ohjelmaikkuna, jossa yksinkertainen painike

Ensimmäisessä esimerkkiohjelmassamme luomme ikkunan, joka sisältää yksinkertaisen painikkeen tekstillä ”Click me!” (kuva 42).

Kuva 42. Yksinkertainen painike.

```
import Graphics.UI.Gtk
import Control.Monad.Trans (liftIO)

main = do
  initGUI
  window <- windowNew
```

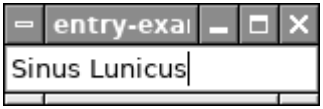
```
  button <- buttonNewWithLabel "Click me!"
  containerAdd window button
  widgetShowAll window
  button `on` buttonPressEvent $ tryEvent $ whenClicked
  onDestroy window mainQuit
  mainGUI

whenClicked = do
  liftIO $ putStrLn "Button was clicked."
```

Luomme ikkunan komennolla `windowNew` ja painikkeen komennolla `buttonNewWithLabel`. Liitämme painikkeen ikkunaan komennolla `containerAdd`. Esitämme ikkunan ja kaikki sen sisältämät alikomponentit komennolla `widgetShowAll`. Asetamme painikkeelle tapahtumankäsittelijän `whenClicked`. Ohjelma kutsuu tapahtumankäsittelijää, aina kun painike lähettää signaalin `buttonPressEvent` eli painiketta painettaessa. Ikkunan sulkeminen (`onDestroy`) päättää ohjelman suorituksen (`mainQuit`).

Tuomme kirjastosta `Control.Monad.Trans` funktion `liftIO`, jonka avulla voimme yhteensovittaa tapahtumankäsittelijän vaatiman tyypin `EventM` sekä syöte- ja tulostustyypin `IO`.

10.2 Ohjelmaikkuna, jossa tekstinsyöttökenttä

Lisäsimme edellä ohjelmaikkunaan painikkeen komennolla `buttonNewWithLabel`. Myös muiden komponenttien lisäys tapahtuu samaa nimeämislogiikkaa noudattaen. Esimerkiksi yksinkertaisen tekstinsyöttökentän (`Entry`) lisäämme komennolla `entryNew` (kuva 43).



Kuva 43. Tekstinsyöttökenttä.

```
import Graphics.UI.Gtk
```

```
main = do
  initGUI
  window <- windowNew
  entry <- entryNew
  containerAdd window entry
  widgetShowAll window
  onDestroy window mainQuit
  mainGUI
```

10.3 Ohjelmaikkuna puunäkymällä

Puunäkymän lisääminen on hieman monimutkaisempi toimenpide, sillä puunäkymä tarvitsee näkymän (*view*) lisäksi myös tietomallin (*model*). Tietomallin pohjana voi olla yksinkertainen lista. Muodostamme listasta tietomallin komennolla `listStoreNew`.

Kun haluamme lisätä ikkunaan useampia komponentteja, on meidän päätettävä komponenttien asettelusta. Käytämme seuraavassa komponenttien asettelemiseksi allekkain pystysuuntaista laatikkoa `VBox` (*vertical box*). Asettelemme komponentit laatikkoon komennolla `boxPackStart` (kuva 44).

Kuva 44. Tekstinsyöttökenttä ja puunäkymä.

```
import Graphics.UI.Gtk
import Graphics.UI.Gtk.ModelView as Model
```

```
main = do
  initGUI
  window <- windowNew
  vbox1 <- vBoxNew False 0
  entry <- entryNew
  containerAdd window vbox1
  list <- listStoreNew [
    "Lacus Somniorum", "Lacus Doloris", "Lacus Timoris",
    "Palus Putredinis", "Lacus Autumni"]
  treeview <- Model.treeViewNewWithModel list
  Model.treeViewSetHeadersVisible treeview False
  col <- Model.treeViewColumnNew
  renderer <- Model.cellRendererTextNew
  Model.cellLayoutPackStart col renderer False
  Model.cellLayoutSetAttributes col renderer list
    (\text -> [Model.cellText := text])
  Model.treeViewAppendColumn treeview col
  tree <- Model.treeViewGetSelection treeview
  boxPackStart vbox1 entry PackRepel 0
  boxPackStart vbox1 treeview PackRepel 0
  widgetShowAll window
  onDestroy window mainQuit
  mainGUI
```

10.4 Tuloslistan suodattaminen säännöllisillä lausekkeilla

Kirjoitamme seuraavaksi ohjelman, joka suodattaa tuloslistan tekstikentässä antamamme säännöllisen lausekkeen avulla.

Luemme ensin komennolla `readFile` tekstitiedoston, josta suodatamme pois tyhjät rivit.

Määrittelemme näppäimen vapautuksen (`keyReleaseEvent`) tapahtumankäsittelijäksi funktion `updateList1`, joka lukee tekstikentän sisällön, kun sitä on muutettu.

Säännöllisten lausekkeiden käsittelyyn tuomme kirjaston `Text.Regex.Posix`, joka tarjoaa funktion `(=~)`. Funktio `(=~)` on monimuotoinen ja vaatii siksi yleisessä muodossaan tyyppimäärittelyn. Kun valitsemme lausekkeen `a =~ b` tyypiksi `Bool`, palauttaa lauseke arvon `True`, mikäli säännöllinen lauseke `b` esiintyy merkkijonossa `a`. Mikäli valitsemme lausekkeen `a =~ b` tyypiksi `String`, lauseke palauttaa ensimmäisen osuman. Listauksessamme käytämme funktiota `(=~)` funktion `filter` ensimmäisenä argumenttina, joten kääntäjä päättelee tyypin olevan `Bool`.

```
> import Text.Regex.Posix
> "abcd" =~ "a" :: Bool
True
> "abcd" =~ "a" :: String
"a"
```

Kuvassa 45 olemme syöttäneet tekstikenttään säännöllisen lausekkeen `(.)\1`. Säännöllisissä lausekkeissa piste `.` vastaa mitä tahansa merkkiä. Sulkumerkit `()` muodostavat ensimmäisen alilausekkeen. Merkintä `\1` vastaa ensimmäistä alilauseketta. Kokonaisuudessaan säännöllinen lauseke `(.)\1` vastaa siten kahta peräkkäistä samaa merkkiä. Esimerkissämme tekstirivejä, joilla esiintyy kaksi peräkkäistä samaa merkkiä, löytyy yhteensä 10 kappaletta.

10.5 Poikkeuksien käsittely

Kun säännöllinen lauseke ei ole kelvollinen, nostaa järjestelmä poikkeuksen (*exception*). Esimerkiksi keskeneräiset lausekkeet kuten "`(.`" eivät ole kelvollisia. Poikkeuksien käsittelyyn tuomme kirjaston `Control.Exception`, joka tarjoaa muun muassa funktiot `try` ja `evaluate`. Näistä `try` on monimuotoinen funktio ja vaatii siksi yleisessä muodossaan tyyppimäärittelyn. Annamme tyyppimäärittelyn funktion `tryFilter` tyyppiallekirjoituksessa.

```
tryFilter :: String -> [String]
          -> IO (Either SomeException [String])
tryFilter txt list = do
  result <- try (evaluate (filter (=~ txt) list))
  return result
```

Kuva 45. Tuloslista suodatettuna säännöllisellä lausekkeella `(.)\1`.

Tyypin `Either` mahdolliset arvot ovat `Left x` ja `Right y`. Funktio `try` palauttaa arvon `Left x` silloin, kun argumenttina antamamme funktion suoritus on keskeytynyt poikkeukseen `x` sekä arvon `Right y` silloin, kun antamamme funktio onnistuneesti palauttaa arvon `y`. Esimerkissämme samaistamme virheellisen säännöllisen lausekkeen lausekkeeseen, jonka tulosjoukko on tyhjä lista `[]`.

```
filter1 txt list = do
  result <- tryFilter txt list
  return $ case result of
    Left ex -> []
    Right val -> val
```

Esitämme seuraavassa ohjelmakoodin kokonaisuudessaan.

```
import Graphics.UI.Gtk
import Graphics.UI.Gtk.ModelView as Model
import Control.Monad.Trans (liftIO)
import Text.Regex.Posix
import Control.Exception
```

```
main = do
  content <- readFile "moon-latin.txt"
  let moon = filter (not . null) (lines content)
  initGUI
  window <- windowNew
  vbox1 <- vBoxNew False 0
  entry <- entryNew
  containerAdd window vbox1
  listore <- listStoreNew []
  treeview <- Model.treeViewNewWithModel listore
  entry `on` keyReleaseEvent $ do
    liftIO $ updateList1 entry listore moon
  Model.treeViewSetHeadersVisible treeview False
  col <- Model.treeViewColumnNew
  renderer <- Model.cellRendererTextNew
  Model.cellLayoutPackStart col renderer False
  Model.cellLayoutSetAttributes col renderer listore
    (\text -> [Model.cellText := text])
  Model.treeViewAppendColumn treeview col
  tree <- Model.treeViewGetSelection treeview
  boxPackStart vbox1 entry PackNatural 0
  boxPackStart vbox1 treeview PackNatural 0
  widgetShowAll window
  liftIO $ updateList1 entry listore moon
  onDestroy window mainQuit
  mainGUI

tryFilter :: String -> [String]
          -> IO (Either SomeException [String])
tryFilter txt list = do
  result <- try (evaluate (filter (=~ txt) list))
  return result

filter1 txt list = do
  result <- tryFilter txt list
  return $ case result of
```

```
    Left ex -> []
    Right val -> val

updateList1 entry listore list = do
  listStoreClear listore
  txt <- entryGetText entry
  list1 <- listStoreToList listore
  list2 <- filter1 txt list
  let
    n = 8
    list3 = take (n + 1) list2
    l2 = length list2
    l3 = length list3
    list4
      | l2 == l3 = take (n + 1) (list2 ++ repeat "")
      | l2 > l3  = take n list3 ++
        ["(+" ++ show (l2 - l3 + 1) ++ " others)"]
  mapM_ (listStoreAppend listore) list4
  return True
```

10.6 Ohjelma Png-kuvan näyttämiseen

Viimeisessä esimerkkiohjelmassamme emme käytä moniakaan Gtk-kirjaston visuaalisia komponentteja vaan kirjoitamme lyhyen yleiskäyttöisen ohjelman Png-muotoisen kuvan esittämiseksi ruudulla.

```
import Control.Concurrent.MVar
import System.IO.Unsafe
import System.Environment (getArgs)
import Graphics.UI.Gtk
import qualified Graphics.Rendering.Cairo as C
import qualified Graphics.UI.Gtk.Gdk.EventM as M
import System.Glib.UTFString (glibToString)

firstArg args =
  case args of
```

```
    [] -> error "must supply a file to open"
    [arg] -> arg
    _ -> error "too many arguments"

main = do
  args <- getArgs
  let arg1 = firstArg args
  initGUI
  var <- newMVar (1.0,0.0,0.0)
  vPos <- newMVar (None,0.0,0.0)
  window <- windowNew
  canvas <- drawingAreaNew
  surf <- return $ unsafeLoadPNG arg1
  widgetAddEvents canvas [Button1MotionMask]
  widgetSetSizeRequest canvas 300 200
  centerImg var surf canvas
  canvas `on` motionNotifyEvent $ do
    (mouseX,mouseY) <- eventCoordinates
    t <- M.eventTime
    C.liftIO $
      changePos vPos var surf canvas mouseX mouseY
    C.liftIO $ logMsg 0 ("Motion Time: " ++ s t)
    return False
  window `on` keyPressEvent $ tryEvent $ do
    key <- eventKeyName
    keyInput var surf canvas (glibToString key)
    C.liftIO $ updateCanvas1 var canvas surf
    return ()
  canvas `on` buttonPressEvent $ tryEvent $ do
    (mouseX,mouseY) <- printMouse
    C.liftIO $ printPointer canvas
    C.liftIO $ printMVar var mouseX mouseY
    C.liftIO $ modifyMVar_ vPos (\_ ->
      return (Press,mouseX,mouseY))
  canvas `on` buttonReleaseEvent $ tryEvent $ do
    (mouseX,mouseY) <- M.eventCoordinates
    m <- M.eventModifier
```

```
    b <- M.eventButton
    (cause,vPosX,vPosY) <- C.liftIO $ readMVar vPos
    C.liftIO $ release cause b var vPosX vPosY
  canvas `on` scrollEvent $ tryEvent $ do
    (mouseX,mouseY) <- M.eventCoordinates
    m <- M.eventModifier
    d <- M.eventScrollDirection
    t <- M.eventTime
    C.liftIO $ changeRef var d mouseX mouseY
    C.liftIO $ updateCanvas1 var canvas surf
    C.liftIO $ logMsg 0 ("Scroll: " ++ s t ++ s mouseX ++
      s mouseY ++ s m ++ s d)
  onDestroy window mainQuit
  onExpose canvas $ const (updateCanvas1 var canvas surf)
  set window [containerChild := canvas]
  widgetShowAll window
  mainGUI

data EvtType = Press | Release | Move | Scroll | None

release Press button var mouseX mouseY = do
  (varS,varX,varY) <- readMVar var
  let
    x = (mouseX - varX) / varS
    y = (mouseY - varY) / varS
  C.liftIO $ logMsg 0
    ("Add point: " ++ s x ++ s y ++ s button)
  C.liftIO $ logMsg 1 (s x ++ s y)

release _ button var x y = do
  C.liftIO $ logMsg 0 ("Release (other): " ++ s x ++ s y)

changePos vPos var surf canvas mouseX mouseY = do
  (cause,vPosX,vPosY) <- readMVar vPos
  (scaleOld,oldX,oldY) <- readMVar var
  let
    dx = vPosX - mouseX
```

```
    dy = vPosY - mouseY
  modifyMVar_ var (\_ ->
    return (scaleOld,oldX - dx,oldY - dy))
  modifyMVar_ vPos (\_ ->
    return (Move,mouseX,mouseY))
  updateCanvas1 var canvas surf

intToDouble :: Int -> Double
intToDouble i = fromRational (toRational i)

s x = show x ++ " "

printMouse = do
  (mouseX,mouseY) <- M.eventCoordinates
  C.liftIO $ logMsg 0 ("Mouse: " ++ s mouseX ++ s mouseY)
  return  (mouseX,mouseY)

printPointer canvas = do
  (widX,widY) <- widgetGetPointer canvas
  logMsg 0 ("Widget: " ++ s widX ++ s widY)

printMVar var mouseX mouseY = do
  (varS,varX,varY) <- readMVar var
  let
    x = (mouseX - varX) / varS
    y = (mouseY - varY) / varS
  logMsg 0 ("MVar: " ++ s varS ++ s varX ++ s varY)
  logMsg 0 ("Calc: " ++ s x ++ s y)

centerImg var surf canvas = do
  w1 <- C.imageSurfaceGetWidth surf
  h1 <- C.imageSurfaceGetHeight surf
  (w2,h2) <- widgetGetSizeRequest canvas
  let
    dh = intToDouble (h2 - h1)
    dw = intToDouble (w2 - w1)
  modifyMVar_ var (\_ -> return (1.0,dw / 2,dh / 2))
```

```
keyInput var surf canvas key = do
  C.liftIO $ print key
  case key of
    "q" -> do
      C.liftIO $ mainQuit
    "1" -> do
      C.liftIO $ centerImg var surf canvas

changeRef var d mouseX mouseY = do
  (scaleOld,oldX,oldY) <- readMVar var
  let
    scaleD = scale1 d
    scaleNew = scaleD * scaleOld
    dx = (mouseX - oldX) * (scaleD - 1)
    dy = (mouseY - oldY) * (scaleD - 1)
    newX = oldX - dx
    newY = oldY - dy
    result =  (scaleNew,newX,newY)
  modifyMVar_ var (\_ -> return result)
  logMsg 0 ("Change MVar: " ++ s scaleNew ++
    s newX ++ s newY)
  where
    factor = 5 / 4
    scale1 ScrollUp   = factor
    scale1 ScrollDown = 1 / factor

updateCanvas1 var canvas surf = do
  win <- widgetGetDrawWindow canvas
  (width, height) <- widgetGetSize canvas
  renderWithDrawable win $
    paintImage1 var surf
  return True

imageSurfaceCreateFromPNG :: FilePath -> IO C.Surface
imageSurfaceCreateFromPNG file =
  C.withImageSurfaceFromPNG file $ \png -> do
```

```
    C.liftIO $ logMsg 0 "Load Image"
    w <- C.renderWith png $ C.imageSurfaceGetWidth png
    h <- C.renderWith png $ C.imageSurfaceGetHeight png
    surf <- C.createImageSurface C.FormatRGB24 w h
    C.renderWith surf $ do
      C.setSourceSurface png 0 0
      C.paint
    return surf

unsafeLoadPNG file = unsafePerformIO $
  imageSurfaceCreateFromPNG file

paintImage1 var surf = do
  (sc,x,y) <- C.liftIO $ readMVar var
  C.setSourceRGB 1 1 1
  C.paint
  C.translate x y
  C.scale sc sc
  C.liftIO $ logMsg 0 ("Paint Image: " ++
    s sc ++ s x ++ s y)
  C.setSourceSurface surf 0 0
  C.paint

logMsg 0 s = do
  return ()
logMsg 1 s = do
  putStrLn s
  return ()
```

10.7 Piirtoalue `DrawingArea`

Luomme ikkunaan piirtoalueen komennolla `drawingAreaNew`. Annamme piirtoalueelle nimen `canvas`. Piirtoalueen kokopyynnön asetamme komennolla `widgetSetSizeRequest`.

Luemme piirtoalueelle kuvan määrittelemällä komennon `imageSurfaceCre-`

`ateFromPNG`. Kuvan keskittämiseksi piirtoalueelle määrittelemme komennon `centerImg`.

Luomme tapahtumankäsittelijöiden avulla käyttäjälle mahdollisuuden siirtää kuvaa ja muuttaa kuvan kokoa. Kun kuvaa on siirretty tai sen kokoa muutettu, piirrämme kuvan uudestaan käyttäen määrittelemäämme komentoa `updateCanvas1`. Tämä komento kutsuu piirtotyön suorittavaa rutiinia `paintImage1`, jossa kutsumme piirtokirjasto Cairon tarjoamia piirtokomentoja. Tuomme piirtokirjaston nimettynä (`qualified ... as C`), joten kaikki sen tarjoamat komennot ohjelmalistauksessa alkavat etuliitteellä "`C.`".

10.8 Tapahtumankäsittelijät

Pääohjelmassa olemme määritelleet tapahtumankäsittelijän hiiren liikkeelle (`motionNotifyEvent`), näppäimen painallukselle (`keyPressEvent`), hiiren näppäimen painallukselle (`buttonPressEvent`), hiiren näppäimen vapauttamiselle (`buttonReleaseEvent`), hiiren rullan pyörittämiselle (`scrollEvent`), ikkunan sulkupainikkeen painamiselle (`onDestroy`) sekä piirtoalueen uudelleenpiirtämiselle (`onExpose`).

Osa tapahtumankäsittelijöistämme on hyvin yksinkertaisia, ja ne kirjoittavat ainoastaan viestin komentoikkunaan funktiokutsulla `logMsg 1` tai jättävät kirjoittamatta funktiokutsulla `logMsg 0`.

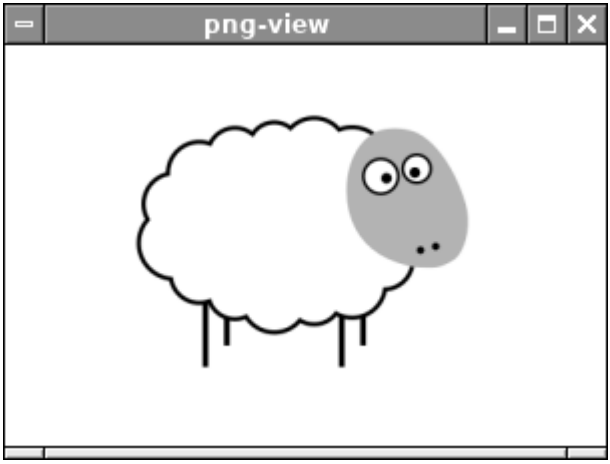
Tavallisesti hiiren liikuttaminen piirtoalueella ei tuota tapahtumasignaalia, mutta voimme halutessamme tuottaa sellaisia esimerkiksi silloin, kun hiiren ykköspainike on painettuna. Teemme näin pääohjelmassa komennolla `widgetAddEvents`.

```
widgetAddEvents canvas [Button1MotionMask]
```

Annamme ohjelmalle nimeksi png-view ja käynnistämme ohjelman komentotulkissa. Ohjelma saa argumenttinaan kuvatiedoston nimen.

```
$ runhaskell png-view foci-4.png
```

Avautuvassa ikkunassa näemme valitsemamme kuvan (kuva 46). Voimme suurentaa ja pienentää näkymää hiiren rullalla.



Kuva 46. Ohjelmaikkuna.

10.9 `MVar`-muuttujaviittaukset

Käytämme ohjelmassamme `MVar`-muuttujaviittauksia tapahtumankäsittelijöissä. Tämä on käytännöllinen tapa välittää tietoa graafisen käyttöliittymän sisällä.

Luomme uuden muuttujan komennolla `newMVar`. Muuttujan sisältämän tiedon luemme komennolla `readMVar`. Olemassaolevaa muuttujaa muutamme komennolla `modifyMVar_`.

Käyttämämme muuttujaviittaukset ovat `var`, joka sisältää kuvan suurennoksen sekä vasemman ylänurkan x- ja y-koordinaatit, ja `vPos`, joka sisältää hiiritapahtuman syyn (`Press` (*painallus*), `Release` (*vapautus*), `Move` (*siirto*), `Scroll` (*rullaus*) tai `None` (*ei mikään*) sekä tapahtumahetken hiiren osoittimen x- ja y-koordinaatit.

Muuttujaviittausten sisältämän tiedon avulla ohjelma kykenee laskemaan esitettävälle kuvalle uudet koordinaatit ja suurennoksen aina tapahtumankäsittelijän sitä pyytäessä.

10.10 Gtk-käyttöliittymäkirjastojen hierarkia

Graphics.UI.Gtk

- Abstract
  Bin, Box, ButtonBox, Container, IMContext, Misc, Object, Paned, Range, Scale, Scrollbar, Separator, Widget.
- ActionMenuToolbar
  Action, ActionGroup, RadioAction, RecentAction, ToggleAction, UIManager.
- Builder
- Buttons
  Button, CheckButton, LinkButton, RadioButton, ScaleButton, ToggleButton, VolumeButton.
- Cairo
- Display
  AccelLabel, Image, InfoBar, Label, ProgressBar, Spinner, StatusIcon, Statusbar.
- Embedding
  Embedding, Plug, Socket, Types.
- Entry
  Editable, Entry, EntryBuffer, EntryCompletion, HScale, SpinButton, VScale.
- Gdk
  AppLaunchContext, Cursor, Display, DisplayManager, DrawWindow, Drawable, EventM, Events, GC, Gdk, Keymap, Keys, Pixbuf, PixbufAnimation, Pixmap, Region, Screen.
- General
  Clipboard, Drag, Enums, General, IconFactory, IconTheme, RcStyle, Selection, Settings, StockItems, Style.
- Layout
  Alignment, AspectFrame, Expander, Fixed, HBox, HButtonBox, HPaned, Layout, Notebook, Table, VBox, VButtonBox, VPaned.
- MenuComboToolbar
  CheckMenuItem, Combo, ComboBox, ComboBoxEntry, ImageMenuItem, Menu, MenuBar, MenuItem, MenuShell, MenuToolButton, OptionMenu, RadioMenuItem, RadioToolButton, SeparatorMenuItem, Separa-

torToolItem, TearoffMenuItem, ToggleToolButton, ToolButton, ToolItem, ToolItemGroup, ToolPalette, Toolbar.

Misc

Accessible, Adjustment, Arrow, Calendar, DrawingArea, EventBox, HandleBox, IMContextSimple, IMMulticontext, SizeGroup, Tooltip, Tooltips, Viewport.

ModelView

CellEditable, CellLayout, CellRenderer, CellRendererAccel, CellRendererCombo, CellRendererPixbuf, CellRendererProgress, CellRendererSpin, CellRendererSpinner, CellRendererText, CellRendererToggle, CellView, CustomStore, IconView, ListStore, TreeDrag, TreeModel, TreeModelFilter, TreeModelSort, TreeRowReference, TreeSelection, TreeSortable, TreeStore, TreeView, TreeViewColumn.

Multiline

TextBuffer, TextIter, TextMark, TextTag, TextTagTable, TextView.

Ornaments

Frame, HSeparator, VSeparator.

Printing

PageSetup, PaperSize, PrintContext, PrintOperation, PrintSettings.

Recent

RecentChooser, RecentChooserMenu, RecentChooserWidget, RecentFilter, RecentInfo, RecentManager.

Scrolling

HScrollbar, ScrolledWindow, VScrollbar.

Selectors

ColorButton, ColorSelection, ColorSelectionDialog, FileChooser, FileChooserButton, FileChooserDialog, FileChooserWidget, FileFilter, FileSelection, FontButton, FontSelection, FontSelectionDialog, HSV.

Special

HRuler, Ruler, VRuler.

Windows

AboutDialog, Assistant, Dialog, Invisible, MessageDialog, OffscreenWindow, Window, WindowGroup.

10.11 Kirjasto `Graphics.UI.Gtk.Abstract.Widget`

Useimpien komponenttien (kuten `Button`) perustyyppi on `Widget`. Kun etsimme esimerkiksi painikkeelle tapahtumankäsittelijöitä, on meidän etsittävä niitä komponentin perustyypin dokumentaatiosta.

Luettelemme seuraavassa tyypin `Widget` tarjoamat metodit, attribuutit ja tapahtumat.

Metodit:

`widgetShow`, `widgetShowNow`, `widgetHide`, `widgetShowAll`, `widgetHideAll`, `widgetDestroy`, `widgetQueueDraw`, `widgetQueueResize`, `widgetQueueResizeNoRedraw`, `widgetSizeRequest`, `widgetGetChildRequisition`, `widgetSizeAllocate`, `widgetAddAccelerator`, `widgetRemoveAccelerator`, `widgetSetAccelPath`, `widgetCanActivateAccel`, `widgetActivate`, `widgetIntersect`, `widgetHasIntersection`, `widgetGetIsFocus`, `widgetGrabFocus`, `widgetGrabDefault`, `widgetSetName`, `widgetGetName`, `widgetSetSensitive`, `widgetSetSensitivity`, `widgetGetParentWindow`, `widgetGetDrawWindow`, `widgetDelEvents`, `widgetAddEvents`, `widgetGetEvents`, `widgetSetEvents`, `widgetSetExtensionEvents`, `widgetGetExtensionEvents`, `widgetGetToplevel`, `widgetGetAncestor`, `widgetGetColormap`, `widgetSetColormap`, `widgetGetPointer`, `widgetIsAncestor`, `widgetTranslateCoordinates`, `widgetSetStyle`, `widgetGetStyle`, `widgetPushColormap`, `widgetPopColormap`, `widgetSetDefaultColormap`, `widgetGetDefaultColormap`, `widgetGetDefaultStyle`, `widgetSetDirection`, `widgetGetDirection`, `widgetSetDefaultDirection`, `widgetGetDefaultDirection`, `widgetShapeCombineMask`, `widgetInputShapeCombineMask`, `widgetGetSnapshot`, `widgetPath`, `widgetClassPath`, `widgetGetCompositeName`, `widgetModifyStyle`, `widgetGetModifierStyle`, `widgetModifyFg`, `widgetModifyBg`, `widgetModifyText`, `widgetModifyBase`, `widgetModifyFont`, `widgetRestoreFg`, `widgetRestoreBg`, `widgetRestoreText`, `widgetRestoreBase`, `widgetCreatePangoContext`, `widgetGetPangoContext`, `widgetCreateLayout`, `widgetRenderIcon`, `widgetQueueDrawArea`, `widgetResetShapes`, `widgetSetAppPaintable`, `widgetSetDoubleBuffered`, `widgetSetRedrawOnAllocate`, `widgetSetCompositeName`, `widgetMnemonicAc-`

tivate, widgetSetScrollAdjustments, widgetRegionIntersect, widgetGetAccessible, widgetChildFocus, widgetGetChildVisible, widgetGetParent, widgetGetSettings, widgetGetClipboard, widgetGetDisplay, widgetGetRootWindow, widgetGetScreen, widgetHasScreen, widgetGetSizeRequest, widgetSetChildVisible, widgetSetSizeRequest, widgetSetNoShowAll, widgetGetNoShowAll, widgetListMnemonicLabels, widgetAddMnemonicLabel, widgetRemoveMnemonicLabel, widgetGetAction, widgetIsComposited, widgetErrorBell, widgetKeynavFailed, widgetGetTooltipMarkup, widgetSetTooltipMarkup, widgetGetTooltipText, widgetSetTooltipText, widgetGetTooltipWindow, widgetSetTooltipWindow, widgetGetHasTooltip, widgetSetHasTooltip, widgetTriggerTooltipQuery, widgetGetWindow, widgetReparent, widgetGetCanFocus, widgetSetCanFocus, widgetGetAllocation, widgetGetAppPaintable, widgetGetCanDefault, widgetSetCanDefault, widgetGetHasWindow, widgetSetHasWindow, widgetGetSensitive, widgetIsSensitive, widgetGetState, widgetGetVisible, widgetGetHasDefault, widgetGetHasFocus, widgetHasGrab, widgetIsDrawable, widgetIsToplevel, widgetSetWindow, widgetSetReceivesDefault, widgetGetReceivesDefault, widgetSetState, widgetGetSavedState, widgetGetSize.

Attribuutit:

widgetName, widgetParent, widgetWidthRequest, widgetHeightRequest, widgetMarginLeft, widgetMarginRight, widgetMarginTop, widgetMarginBottom, widgetVisible, widgetOpacity, widgetSensitive, widgetAppPaintable, widgetCanFocus, widgetHasFocus, widgetIsFocus, widgetCanDefault, widgetHasDefault, widgetReceivesDefault, widgetCompositeChild, widgetStyle, widgetState, widgetEvents, widgetExtensionEvents, widgetExpand, widgetHExpand, widgetHExpandSet, widgetVExpand, widgetVExpandSet, widgetNoShowAll, widgetChildVisible, widgetColormap, widgetCompositeName, widgetDirection, widgetTooltipMarkup, widgetTooltipText, widgetHasTooltip, widgetHasRcStyle, widgetGetRealized, widgetGetMapped, widgetSetRealized, widgetSetMapped, realize, unrealize, mapSignal, unmapSignal, sizeRequest, sizeAllocate, showSignal, hideSignal, focus, stateChanged, parentSet, hierarchyChanged, styleSet, directionChanged, grabNotify, popupMenuSignal, showHelp, accelClosuresChanged, screenChanged,

queryTooltip.

Tapahtumat:

buttonPressEvent, buttonReleaseEvent, configureEvent, deleteEvent, destroyEvent, enterNotifyEvent, exposeEvent, focusInEvent, focusOutEvent, grabBrokenEvent, keyPressEvent, keyReleaseEvent, leaveNotifyEvent, mapEvent, motionNotifyEvent, noExposeEvent, proximityInEvent, proximityOutEvent, scrollEvent, unmapEvent, visibilityNotifyEvent, windowStateEvent.

10.12 Kirjasto Graphics.UI.Gtk.Gdk.EventM

Funktioita:

eventWindow, eventSent, eventCoordinates, eventRootCoordinates, eventModifier, eventModifierAll, eventTime, eventKeyVal, eventKeyName, eventHardwareKeycode, eventKeyboardGroup, eventButton, eventScrollDirection, eventIsHint, eventArea, eventRegion, eventVisibilityState, eventCrossingMode, eventNotifyType, eventCrossingFocus, eventFocusIn, eventPosition, eventSize, eventWindowStateChanged, eventWindowState, eventChangeReason, eventSelection, eventSelectionTime, eventKeyboardGrab, eventImplicit, eventGrabWindow, currentTime, tryEvent, stopEvent.

10.13 Piirtokirjasto Graphics.Rendering.Cairo

Komennot

Piirtokomennot:

renderWith, save, restore, status, withTargetSurface, pushGroup, pushGroupWithContent, popGroupToSource, setSourceRGB, setSourceRGBA, setSource, setSourceSurface, getSource, setAntialias, setDash, setFillRule, getFillRule, setLineCap, getLineCap, setLineJoin, getLineJoin, setLineWidth, getLineWidth, setMiterLimit, getMiterLimit, setOperator, getOperator, setTolerance, getTolerance, clip, clipPreserve, clipExtents, resetClip,

fill, fillPreserve, fillExtents, inFill, mask, maskSurface, paint, paintWithAlpha, stroke, strokePreserve, strokeExtents, inStroke, copyPage, showPage.

Polut:

getCurrentPoint, newPath, closePath, arc, arcNegative, curveTo, lineTo, moveTo, rectangle, textPath, relCurveTo, relLineTo, relMoveTo.

Kuviointi:

withRGBPattern, withRGBAPattern, withPatternForSurface, withGroupPattern, withLinearPattern, withRadialPattern, patternAddColorStopRGB, patternAddColorStopRGBA, patternSetMatrix, patternGetMatrix, patternSetExtend, patternGetExtend, patternSetFilter, patternGetFilter.

Koordinaattimuunnokset:

translate, scale, rotate, transform, setMatrix, getMatrix, identityMatrix, userToDevice, userToDeviceDistance, deviceToUser, deviceToUserDistance.

Teksti:

selectFontFace, setFontSize, setFontMatrix, getFontMatrix, setFontOptions, showText, fontExtents, textExtents.

Kirjasin:

fontOptionsCreate, fontOptionsCopy, fontOptionsMerge, fontOptionsHash, fontOptionsEqual, fontOptionsSetAntialias, fontOptionsGetAntialias, fontOptionsSetSubpixelOrder, fontOptionsGetSubpixelOrder, fontOptionsSetHintStyle, fontOptionsGetHintStyle, fontOptionsSetHintMetrics, fontOptionsGetHintMetrics.

Pinnat:

withSimilarSurface, createSimilarSurface, renderWithSimilarSurface, surfaceGetFontOptions, surfaceFinish, surfaceFlush, surfaceMarkDirty, surfaceMarkDirtyRectangle, surfaceSetDeviceOffset.

Kuvapinnat:

withImageSurface, withImageSurfaceForData, formatStrideForWidth, createImageSurfaceForData, createImageSurface, imageSurfaceGetWidth, imageSurfaceGetHeight, imageSurfaceGetFormat, imageSurfaceGetStride, imageSurfaceGetData, imageSurfaceGet-

`Pixels`.

Png-kuvatiedostot:
`withImageSurfaceFromPNG`, `imageSurfaceCreateFromPNG`, `surfaceWriteToPNG`.

Pdf-pinnat:
`withPDFSurface`, `pdfSurfaceSetSize`.

PostScript-pinnat:
`withPSSurface`,
`psSurfaceSetSize`.

Svg-vektorikuvat:
`withSVGSurface`.

Alueet:
`regionCreate`, `regionCreateRectangle`, `regionCreateRectangles`, `regionCopy`, `regionGetExtents`, `regionNumRectangles`, `regionGetRectangle`, `regionIsEmpty`, `regionContainsPoint`, `regionContainsRectangle`, `regionEqual`, `regionTranslate`, `regionIntersect`, `regionIntersectRectangle`, `regionSubtract`, `regionSubtractRectangle`, `regionUnion`, `regionUnionRectangle`, `regionXor`, `regionXorRectangle`.

Muut komennot:
`liftIO`, `version`, `versionString`.

Tietotyypit

```
data Render m
data Matrix
data Surface
data Pattern
data Status
data Operator = OperatorClear | OperatorSource
  | OperatorOver | OperatorIn | OperatorOut | OperatorAtop
  | OperatorDest | OperatorDestOver | OperatorDestIn
  | OperatorDestOut | OperatorDestAtop | OperatorXor
  | OperatorAdd | OperatorSaturate | OperatorMultiply
  | OperatorScreen | OperatorOverlay | OperatorDarken
  | OperatorLighten | OperatorColorDodge
```

```
  | OperatorColorBurn | OperatorHardLight
  | OperatorSoftLight | OperatorDifference
  | OperatorExclusion | OperatorHslHue
  | OperatorHslSaturation | OperatorHslColor
  | OperatorHslLuminosity
data Antialias = AntialiasDefault | AntialiasNone
  | AntialiasGray | AntialiasSubpixel | AntialiasFast
  | AntialiasGood | AntialiasBest
data FillRule = FillRuleWinding | FillRuleEvenOdd
data LineCap = LineCapButt | LineCapRound | LineCapSquare
data LineJoin = LineJoinMiter | LineJoinRound
  | LineJoinBevel
data ScaledFont
data FontFace
data Glyph
data TextExtents = TextExtents {
  textExtentsXbearing, textExtentsYbearing,
  textExtentsWidth, textExtentsHeight,
  textExtentsXadvance, textExtentsYadvance }
data FontExtents = FontExtents {
  fontExtentsAscent, fontExtentsDescent,
  fontExtentsHeight, fontExtentsMaxXadvance,
  fontExtentsMaxYadvance }
data FontSlant = FontSlantNormal | FontSlantItalic
  | FontSlantOblique
data FontWeight = FontWeightNormal | FontWeightBold
data SubpixelOrder
data HintStyle = HintStyleDefault | HintStyleNone
  | HintStyleSlight | HintStyleMedium | HintStyleFull
data HintMetrics = HintMetricsDefault | HintMetricsOff
  | HintMetricsOn
data FontOptions
data Path
data RectangleInt = RectangleInt { x, y, width, height }
data RegionOverlap = RegionOverlapIn | RegionOverlapOut
  | RegionOverlapPart
data Region
```

```
data Content = ContentColor | ContentAlpha
  | ContentColorAlpha
data Format = FormatARGB32 | FormatRGB24 | FormatA8
  | FormatA1
data Extend = ExtendNone | ExtendRepeat | ExtendReflect
  | ExtendPad
data Filter = FilterFast | FilterGood | FilterBest
  | FilterNearest | FilterBilinear | FilterGaussian
```

Luku 11

Standardikirjasto `Prelude`

11.1 Operaattoreita

```
($!) :: (a -> b) -> a -> b
(!!) :: [a] -> Int -> a
($) :: (a -> b) -> a -> b
(&&) :: Bool -> Bool -> Bool
(++) :: [a] -> [a] -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
(=<<) :: Monad m => (a -> m b) -> m a -> m b
(^) :: (Num a, Integral b) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(||) :: Bool -> Bool -> Bool
```

11.2 Perustietotyypit

```
data Bool = False | True
type String = [Char]
data Either a b = Left a | Right b
data Maybe a = Nothing | Just a
```

11.3 Funktioluettelo

```
all :: (a -> Bool) -> [a] -> Bool
and :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
appendFile :: FilePath -> String -> IO ()
asTypeOf :: a -> a -> a
break :: (a -> Bool) -> [a] -> ([a], [a])
catch :: IO a -> (IOError -> IO a) -> IO a
concat :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
const :: a -> b -> a
curry :: ((a, b) -> c) -> a -> b -> c
cycle :: [a] -> [a]
drop :: Int -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
either :: (a -> c) -> (b -> c) -> Either a b -> c
elem :: Eq a => a -> [a] -> Bool
error :: [Char] -> a
even :: Integral a => a -> Bool
filter :: (a -> Bool) -> [a] -> [a]
flip :: (a -> b -> c) -> b -> a -> c
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl1 :: (a -> a -> a) -> [a] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
fromIntegral :: (Integral a, Num b) => a -> b
fst :: (a, b) -> a
gcd :: Integral a => a -> a -> a
getChar :: IO Char
getContents :: IO String
getLine :: IO String
head :: [a] -> a
id :: a -> a
init :: [a] -> [a]
interact :: (String -> String) -> IO ()
```

```
ioError :: IOError -> IO a
iterate :: (a -> a) -> a -> [a]
last :: [a] -> a
lcm :: Integral a => a -> a -> a
length :: [a] -> Int
lex :: ReadS String
lines :: String -> [String]
lookup :: Eq a => a -> [(a, b)] -> Maybe b
map :: (a -> b) -> [a] -> [b]
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
maximum :: Ord a => [a] -> a
maybe :: b -> (a -> b) -> Maybe a -> b
minimum :: Ord a => [a] -> a
not :: Bool -> Bool
notElem :: Eq a => a -> [a] -> Bool
null :: [a] -> Bool
odd :: Integral a => a -> Bool
or :: [Bool] -> Bool
otherwise :: Bool
print :: Show a => a -> IO ()
product :: Num a => [a] -> a
putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
read :: Read a => String -> a
readFile :: FilePath -> IO String
readIO :: Read a => String -> IO a
readLn :: Read a => IO a
readParen :: Bool -> ReadS a -> ReadS a
reads :: Read a => ReadS a
realToFrac :: (Real a, Fractional b) => a -> b
repeat :: a -> [a]
replicate :: Int -> a -> [a]
reverse :: [a] -> [a]
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr1 :: (a -> a -> a) -> [a] -> [a]
seq :: a -> b -> b
sequence :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
showChar :: Char -> ShowS
showParen :: Bool -> ShowS -> ShowS
showString :: String -> ShowS
shows :: Show a => a -> ShowS
snd :: (a, b) -> b
span :: (a -> Bool) -> [a] -> ([a], [a])
splitAt :: Int -> [a] -> ([a], [a])
subtract :: Num a => a -> a -> a
sum :: Num a => [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
uncurry :: (a -> b -> c) -> (a, b) -> c
undefined :: a
unlines :: [String] -> String
until :: (a -> Bool) -> (a -> a) -> a -> a
unwords :: [String] -> String
unzip :: [(a, b)] -> ([a], [b])
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
userError :: String -> IOError
words :: String -> [String]
writeFile :: FilePath -> String -> IO ()
zip :: [a] -> [b] -> [(a, b)]
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

11.4 Luokka Bounded

```
class Bounded a where
```

```
  minBound :: a
  maxBound :: a
```

11.5 Luokka `Floating`

```
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  sqrt :: a -> a
  log :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  tan :: a -> a
  cos :: a -> a
  asin :: a -> a
  atan :: a -> a
  acos :: a -> a
  sinh :: a -> a
  tanh :: a -> a
  cosh :: a -> a
  asinh :: a -> a
  atanh :: a -> a
  acosh :: a -> a
```

11.6 Luokka `Fractional`

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
```

11.7 Luokka `Integral`

```
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

11.8 Luokka `Num`

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

11.9 Luokka `Ord`

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

11.10 Luokka RealFloat

```
class (RealFrac a, Floating a) => RealFloat a where
  floatRadix :: a -> Integer
  floatDigits :: a -> Int
  floatRange :: a -> (Int, Int)
  decodeFloat :: a -> (Integer, Int)
  encodeFloat :: Integer -> Int -> a
  exponent :: a -> Int
  significand :: a -> a
  scaleFloat :: Int -> a -> a
  isNaN :: a -> Bool
  isInfinite :: a -> Bool
  isDenormalized :: a -> Bool
  isNegativeZero :: a -> Bool
  isIEEE :: a -> Bool
  atan2 :: a -> a -> a
```

11.11 Luokka RealFrac

```
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
```

11.12 Luokka Monad

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```