Luku 1

Geometrisia kuvioita

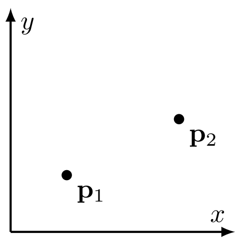
1.1 Piste `Point`

Määrittelemme tyypin `Point`, joka kuvaa pistettä $\mathbf{p} = (x, y)$ kaksiulotteisessa avaruudessa.

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

Pisteet `p1` ja `p2` sijaitsevat koordinaateissa $\mathbf{p}_1 = (1, 1)$ ja $\mathbf{p}_2 = (3, 2)$ (kuva 1).

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



Kuva 1. Pisteet $\mathbf{p}_1 = (1, 1)$ ja $\mathbf{p}_2 = (3, 2)$.

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]
```

1.2 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).

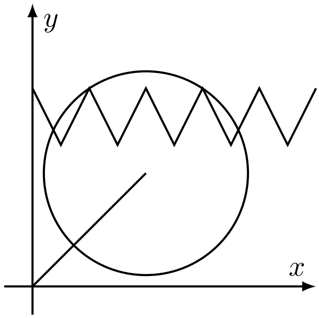
Viivajonon `PolyLine` parametri on koordinaattipisteiden lista tyyppiä `[Point]`.

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 2).

```
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 2. 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`.

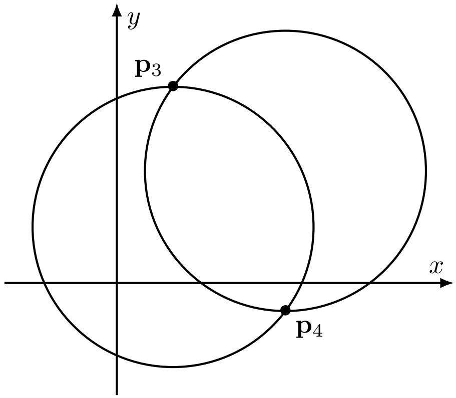
```
> shapes1 = [line1,circle1,polyline1]
> :t shapes1
shapes1 :: [Shape]
```

1.3 Ympyröiden leikkauspisteet

Piirrämme kaksi ympyrää $\mathbf{c}_1$ ja $\mathbf{c}_2$, joiden keskipisteinä ovat pisteet $\mathbf{p}_1 = (1,1)$ ja $\mathbf{p}_2 = (3,2)$. Molempien ympyröiden säde on $r = 2.5$. Etsimme ympyröiden leikkauspisteet $\mathbf{p}_3$ ja $\mathbf{p}_4$ (kuva 3).

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

c1 = Circle 2.5 p1
```

```
c2 = Circle 2.5 p2
```



Kuva 3. Ympyröiden $\mathbf{c}_1$ ja $\mathbf{c}_2$ leikkauspisteet $\mathbf{p}_3$ ja $\mathbf{p}_4$.

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 $\mathbf{p}_1$ ja $\mathbf{p}_2$ 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
```

1.4 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
```

1.5 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
```

1.6 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
```

1.7 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.

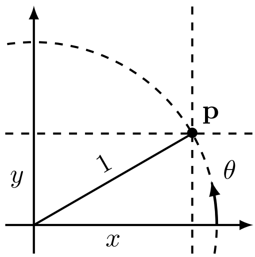
```
> RAD halfpi `addAngles` DEG 90
RAD 3.141592653589793
> DEG 90 `addAngles` DEG 180
DEG 270.0
```

1.8 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 $\theta = \dfrac{2\pi/4}{3}$ keskipisteestä piirretyn itävektorin osoittamasta nollakulmasta ympyrän huipulle, olemme tulleet korkeudelle $\sin\theta = \sin\dfrac{2\pi/4}{3} = 0.5$ nollatasosta (kuva 4).



Kuva 4. Sinifunktio palauttaa korkeuden nollatasosta eli yksikköympyrän keskuskulmaa θ vastaavan pisteen $\mathbf{p}$ y-koordinaatin.

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

1.9 Kiertomatriisi

Kun haluamme kiertää pisteen $\mathbf{p} = (x_1, y_1)$ origon ympäri, kerromme kiertomatriisilla $\mathbf{R}$ koordinaattimatriisin.

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

Haskell-kielessä esitämme matriisit listoina. Esimerkiksi voimme määritellä kiertomatriisin $\mathbf{R}$ funktiossa `rotationMatrix`.

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

Matriisin ja paikkavektorin välinen kertolasku yleistyy kaiken kokoisille matriiseille.

```
matrixTimes1 a b =
  [sum [x * y | (x,y) <- zip a1 b] | a1 <- a]
```

Määrittelemämme funktion `matrixTimes1` tyyppi on `[[a]] -> [a] -> [a]`, missä tyypin `a` tulee olla tyyppiluokan `Num` jäsen.

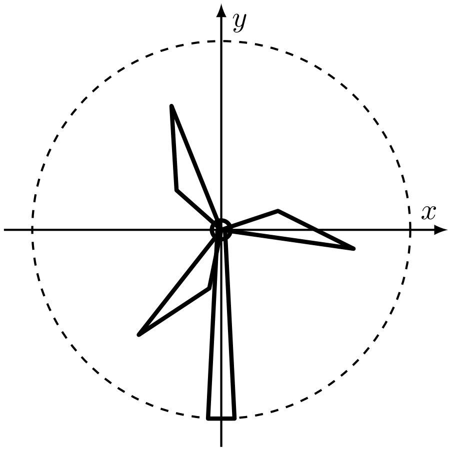
```
> :t matrixTimes1
matrixTimes1 :: Num a => [[a]] -> [a] -> [a]
```

Nyt määrittelemme pisteen $\mathbf{p} = (x_1, y_1)$ kierron origon ympäri kulman t verran funktiossa `rot1`.

```
rot1 t (Point x1 y1) = Point x y
  where
    [x,y] = matrixTimes1 (rotationMatrix t) [x1,y1]
```

Kun jaamme täysympyrän kolmeen osaan, kierrämme annetut pisteet origon ympäri ja lisäämme muutaman apuviivan, saamme tutun kuvion (kuva 5).

```
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]
```



Kuva 5. Tuulimylly, jossa kiertomatriisin avulla muodostetut lavat.

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

rotor = Circle 0.05 (Point 0 0)

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

1.10 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.

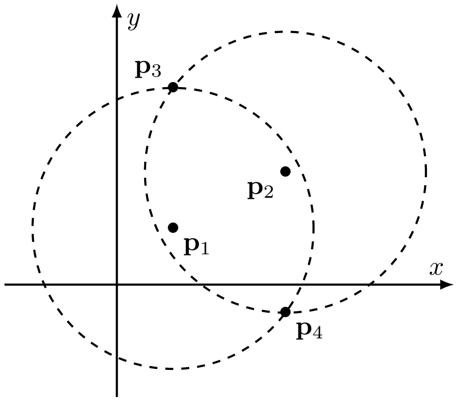
```
data Vector = Vector Double Double
  deriving Show
```

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)
```

1.11 Vektorien suuntakulmat

Etsimme seuraavaksi ympyröiden keskipisteiden $\mathbf{p}_1$ ja $\mathbf{p}_2$ sekä leikkauspisteiden $\mathbf{p}_3$ ja $\mathbf{p}_4$ välisten vektorien suuntakulmat (kuva 6).



Kuva 6. Ympyröiden keskipisteet $\mathbf{p}_1$ ja $\mathbf{p}_2$ sekä leikkauspisteet $\mathbf{p}_3$ ja $\mathbf{p}_4$.

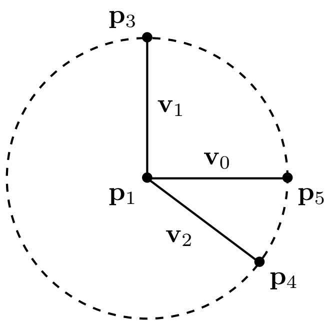
Nollakulmaa vastaavan vektorin saamme suuntaamalla vektorin itään pisteestä (x, y) esimerkiksi pisteeseen $(x + 1, y)$.

```
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 $\mathbf{c}_1$ kohdalla tilanne on kuvan 7 mukainen.



Kuva 7. Ympyrän $\mathbf{c}_1$ keskipiste $\mathbf{p}_1$, leikkauspisteet $\mathbf{p}_3$ ja $\mathbf{p}_4$ sekä nollakulmaa vastaava kehäpiste $\mathbf{p}_5$.

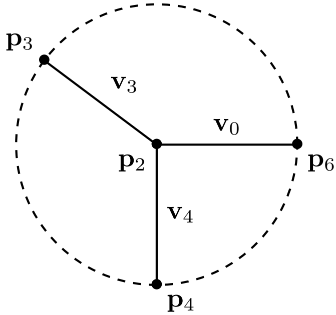
```
p1 = Point 1 1
p2 = Point 3 2
c1 = Circle 2.5 p1
c2 = Circle 2.5 p2
[p3,p4] = circleCircleIntersections c1 c2
v0 = eastVector p1
v1 = mkVector p1 p3
v2 = mkVector p1 p4
t1 = angleBt v0 v1
t2 = angleBt v0 v2
```

Saamme

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

Ympyrän $\mathbf{c}_2$ kohdalla voimme käyttää edellä saamiamme tuloksia ja etsiä vektorit v_3 ja v_4 (kuva 8).

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



Kuva 8. Ympyrän $\mathbf{c}_2$ keskipiste $\mathbf{p}_2$, leikkauspisteet $\mathbf{p}_3$ ja $\mathbf{p}_4$ sekä nollakulmaa vastaava kehäpiste $\mathbf{p}_6$.

Nyt saamme

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

1.12 Ellipsi

Ellipsin parametrimuotoinen esitys on

$$
\begin{aligned}
x &= a \cdot \cos t \\
y &= b \cdot \sin t
\end{aligned}
$$

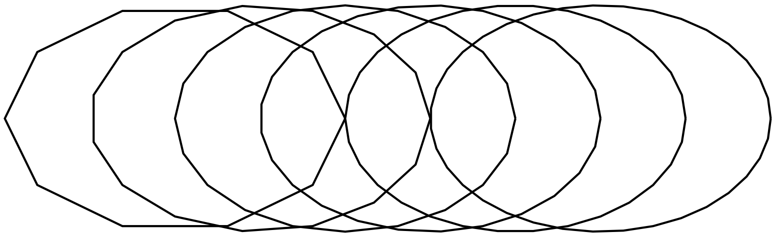
missä a ja b ovat isompi ja pienempi puoliakseli ja $t \in [0, 2\pi]$. Saamme näin ollen Haskell-kielellä pisteen ellipsin kehältä algoritmilla

```
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 9 piirtäneet ellipsit 10, 15, 20, 25, 30 ja 35 viivasegmentin avulla.



Kuva 9. 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 ]
```

1.13 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.

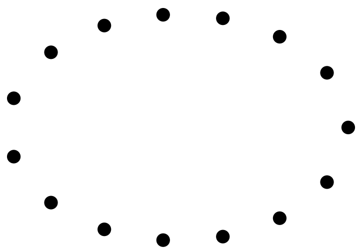
```
-- | A point with distance d along a PolyLine pl
alongPL pl d = along1 0 d pts
  where
    PolyLine pts = pl

-- | 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 10.

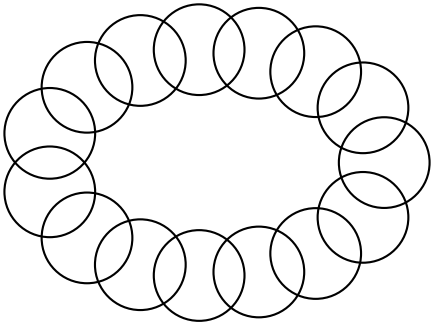


Kuva 10. Pisteet viivajonon varrella.

1.14 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 11).

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



Kuva 11. Ympyrät viivajonolla.

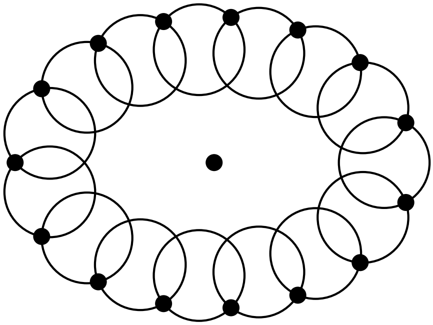
Viereisten ympyröiden leikkauspisteet saamme nyt funktiolla `circleCircleIntersections`. Käytämme piirroksessamme ainoastaan ulompia leikkauspisteitä. Kuvion keskipiste on pisteessä $(0, 0)$. Saamme kahdesta pisteestä ulompana sijaitsevan määrittelemällä funktion `maxDist`. Se saa parametreinaan keskipisteen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

Olemme piirtäneet kuvion keskipisteen ja ulommat leikkauspisteet kuvaan 12.

```
dots2 = outer
  where
    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
```

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

```
arcs2 = [
  Arc
```



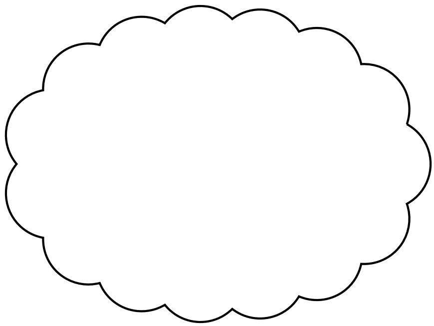
Kuva 12. Kuvion keskipiste ja ympyröiden ulommat leikkauspisteet.

```
(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 13.



Kuva 13. Ulompien leikkauspisteiden välisten kaarien muodostama kuvio.

1.15 Satunnaisluvut

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

```
import System.Random
```

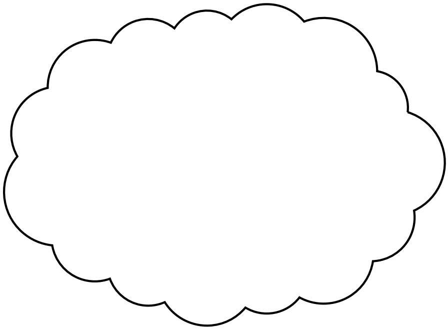
Kirjastosta `System.Random` löydämme funktion `randomRs`, joka saa parametreinaan satunnaislukujen välin ala- ja ylärajan sekä satunnaisgeneraattorin. Alustamme satunnaisgeneraattorin vakioarvolla 42.

```
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 14.



Kuva 14. Kaarien muodostama kuvio, kun ympyrän koko vaihtelee (säde $r = 0.8 \pm 0.15$).

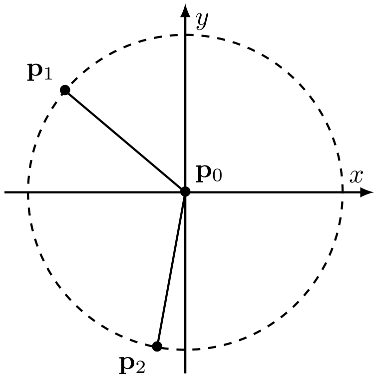
1.16 Kaaren piirron ongelmatilanteita

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

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

Esimerkiksi kuvan 15 tilanteessa olemme löytäneet pisteet $\mathbf{p}_1$ ja $\mathbf{p}_2$, joiden suuntakulmat ovat $t\left(\mathbf{p}_1\right) = 2.44$ rad ja $t\left(\mathbf{p}_2\right) = -1.75$ rad, ja joiden välille haluamme piirtää ympyränkaaren. Tällöin piirtokirjasto tyypillisesti piirtää kaaren pidempää reittiä ympyrän oikeaa puolta pisteestä $\mathbf{p}_2$ pisteeseen $\mathbf{p}_1$, kun haluaisimme kaaren kulkevan lyhyempää reittiä ympyrän vasenta puolta.

Ratkaisu kaaren piirron ongelmatilanteeseen on tapauskohtainen. Tässä esimerkissä olemme ratkaisseet tilanteen lisäämällä negatiiviseen loppukulmaan



Kuva 15. Kaaren piirrossa on varauduttava tilanteeseen, jossa pisteiden $\mathbf{p}_1$ ja $\mathbf{p}_2$ suuntakulmat ovat vastakkaismerkkiset, esimerkiksi $t\,(\mathbf{p}_1) = 2.44$ rad ja $t\,(\mathbf{p}_2) = -1.75$ rad.

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))
```

1.17 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`.

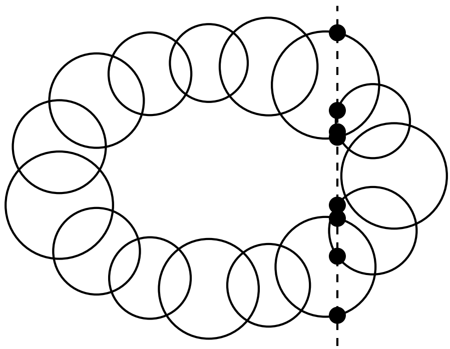
```
towards a p r = Point (x + r * cos1 a) (y + r * sin1 a)
  where
    Point x y = p
```

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

Esitämme viivan, ympyrät ja niiden leikkauspisteet kuvassa 16.



Kuva 16. Viivan ja ympyröiden 8 leikkauspistettä.

1.18 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]
```

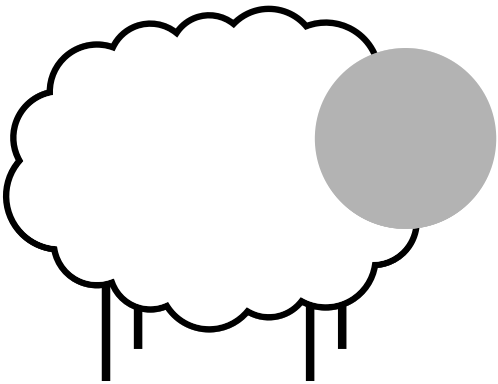
Esimerkiksi funktio `snd` palauttaa tietueen toisen alkion. Nyt siis funktiokutsu `sortOn snd` järjestää listan tietueen toisen alkion mukaan.

```
> 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 17) ja havaitsemme, että kuvion juoni alkaa hahmottua.



Kuva 17. Kuva alkaa hahmottua.

1.19 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.

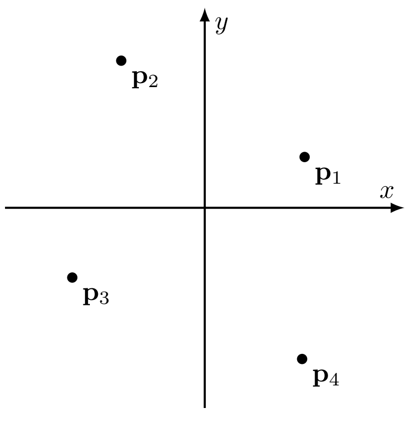
```
move0 (x1,y1) (x2,y2) = (x1 + x2,y1 + y2)
```

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 18).

```
sc1 = scanl1 move0 pts4
```



Kuva 18. Pistejoukon `pts4` pisteet siirrettynä funktiokutsulla `scanl1 move0 pts4`.

1.20 Bezier-käyrät

Sanomme kuutiolliseksi *Bezier-käyräksi* käyrää $\mathbf{B}(t)$, jonka kulku määräytyy pisteiden $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ ja $\mathbf{p}_3$ mukaan painotettuna kaavalla

$$\mathbf{B}(t) = (1-t)^3 \cdot \mathbf{p}_0 + 3(1-t)^2 t \cdot \mathbf{p}_1 + 3(1-t)t^2 \cdot \mathbf{p}_2 + t^3 \cdot \mathbf{p}_3$$

Tässä muuttuja t saa arvot väliltä $0 \leq t \leq 1$. Piste $\mathbf{p}_0$ on käyrän alkupiste ja piste $\mathbf{p}_3$ loppupiste. Kun $t = 0$, olemme käyrän alussa pisteessä $\mathbf{p}_0$. Kun $t = 1$, olemme käyrän lopussa pisteessä $\mathbf{p}_3$. Pisteet $\mathbf{p}_1$ ja $\mathbf{p}_2$ 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) ]
```

1.21 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]]
```

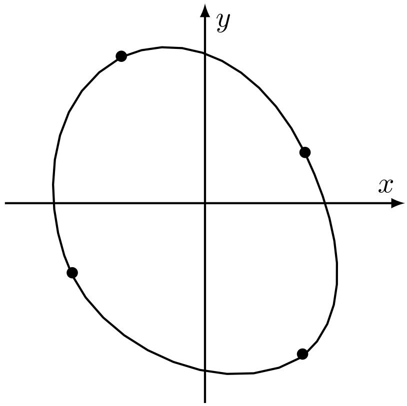
Kirjoitamme funktion `headPts1`, joka palauttaa Bezier-käyrän piirtoon vaadittavat koordinaatit tietueena muodossa `(p0,p1,p2,p3)`.

```
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 19).

```
sheepHead1 = [mkPoint (bezier p0 p1 p2 p3 t)
  | (p0,p1,p2,p3) <- headPts1, t <- [0.0,0.1..0.9]]
```



Kuva 19. Neljän Bezier-käyrän muodostama kuvio.

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
  where
    pts3 = map (addCoords (Point 3.38 0.78)) pts2
    pts2 = map (scaleCoords 0.038) sheepHead1
```

1.22 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
```

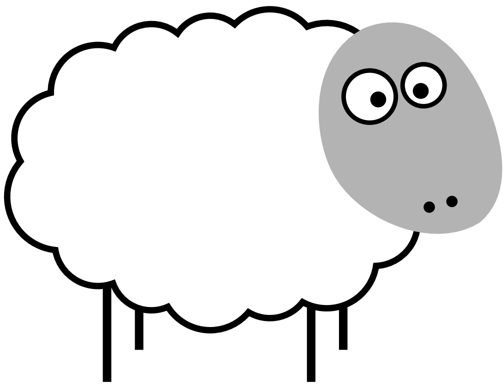
```
  | Line Point Point
  | Polygon [Point]
  | PolyLine [Point]
  | Arc Double Point Angle Angle
  | Filled Shape
```

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 20).



Kuva 20. 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)
```

1.23 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"
```

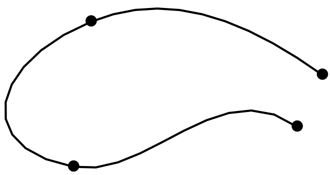
Alkupiste ei ole piirroksemme kannalta lainkaan oikea, joten voimme jättää sen pois listauksesta Haskell-kielellä. Saamme nyt

```
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 21.



Kuva 21. Kolmen Bezier-käyrän muodostama avoin kuvio.

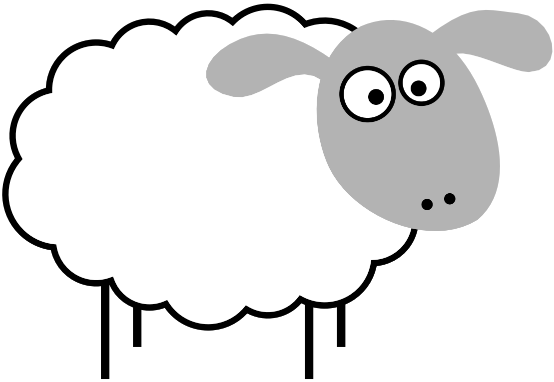
Toimimme vasemman korvan suhteen samalla periaatteella.

```
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),
  (-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`.

```
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]
```

Kuvamme on nyt valmis (kuva 22).



Kuva 22. Valmis kuva: Foci-lammas.