- Yhden *muuttujan* kysymys

$$\texttt{let}\ x\ \texttt{=}\ g\ \texttt{in}\ e \overset{?}{::} \alpha$$

  - ratkaistaan oleellisesti samoin: parametrittömänä funktiona
  - *mutta* Haskell ei yleistäkään sellaisia muuttujia, joille ilmaantui tyyppiluokkia lisäehdoiksi.

- Sitä mutkikkaamman *hahmon* kysymyksessä

$$\texttt{let}\ p\ \texttt{=}\ g\ \texttt{in}\ e \overset{?}{::} \alpha$$

  - hahmo p käsitellään kuten `case`-lausekkeessa
  - Haskell ei yleistäkään *yhtään mitään*.

- Nämä esteet Haskellin itse tekemille yleistyksille seuraavat sen *monomorfismirajoituksesta* (monomorphism restriction).

- Kaikissa `let`-monimuotoisissa tyypin päättelevissä kielissä on jokin tällainen rajoite
  - mutta ne vaihtelevat kielestä toiseen
  - koska `let`in tyypinpäättely poikkeaa hieman sen suoritusmekanismista.

- Jos törmäät rajoitteeseen, niin kirjoita haluamasi polymorfinen tyyppi näkyviin:

  Silloin Haskell vain tarkastaa päteekö se.

- Jos tehdään yhtä aikaa kaksi (tai useampia) *toisiinsa viittaavaa* määritelmää kysymyksellä

$$\texttt{let \{ } f_1 \ldots \texttt{ = } g_1 \texttt{ ; } f_2 \ldots \texttt{ = } g_2 \texttt{ \} in } e \overset{?}{::} \alpha$$

  - ratkaistaan *yhtä aikaa* molemmat kysymykset

$$g_1 \overset{?}{::} \gamma_1$$
$$g_2 \overset{?}{::} \gamma_2$$

  - yleistetään kumpikin silti *erikseen* jolloin mahdolliset yhteiset polymorfiset muuttujat t saavat kummassakin oman kvanttorin.

  - Vertaa looginen totuus

$$(\forall t.G_1(t) \wedge G_2(t)) \Leftrightarrow (\forall t_1.G_1(t_1)) \wedge (\forall t_2.G_2(t_2))$$

- Jos määrittely f_2 viittaa määrittelyyn f_1 mutta ei päinvastoin, niin tehdään

  1. ensin f_1

  2. sitten f_2

  jotta jälkimmäinen voisi hyödyntää edellisen polymorfista lopputulosta.

- Jos ne eivät viittaa toisiinsa, niin silloin niiden keskinäinen käsittelyjärjestyskin on vapaa.

- Haskell'-standardiehdotus ja GHC sallivat samaan tapaan tyypitykset ' :: *tyyppi*' myös lausekkeiden ja hahmojen sisällä ohjaamassa tyypinpäättelyä.

4.5.3 Tyyppiyhtälöt

- Tyyppiyhtälöt ovat siis muotoa

$$tyyppi = tyyppi'.$$

- Ne ilmaisevat, että näiden kahden *tyypin* pitää olla *samat*. . .

- siinä mielessä, että ne molemmat ovat syntaktisesti yksi ja sama ilmaus, joka voisi esiintyä Haskell-tyypityksen ' :: ' oikeana puolena.

- Yhtälön kumpikin *tyyppi* koostuu
  - perustyypeistä kuten `Integer`
  - tyyppimuuttujista
  - tyyppikonstruktorin C sovelluksesta

    $$C\ \mathit{tyyppi}_1\ \mathit{tyyppi}_2\ \mathit{tyyppi}_3\ \ldots\ \mathit{tyyppi}_k$$

    toisiin *tyyppeihin*, esimerkiksi

    $$(\texttt{->})\ \alpha\ \beta$$

    funktiotyypille, jonka parametrin tyyppi on α ja tuloksen β.

1. Yhtälö

$$tyyppi = tyyppi,$$

jonka molemmat puolet ovat täsmälleen samanlaiset, voidaan unohtaa tarpeettomana.

2. Yhtälöt

$$tyyppi = tyyppi' \quad \text{ja} \quad tyyppi' = tyyppi$$

tarkoittavat samaa, eli yhtälön puolien keskinäisellä järjestyksellä ei ole merkitystä.

3. Yhtälön

$$C\ \mathit{tyyppi}_1\ \mathit{tyyppi}_2\ \mathit{tyyppi}_3\ \ldots\ \mathit{tyyppi}_k = C\ \mathit{tyyppi}'_1\ \mathit{tyyppi}'_2\ \mathit{tyyppi}'_3\ \ldots\ \mathit{tyyppi}'_k$$

voi purkaa pienemmiksi yhtälöiksi

$$\mathit{tyyppi}_1 = \mathit{tyyppi}'_1 \quad \mathit{tyyppi}_2 = \mathit{tyyppi}'_2 \quad \mathit{tyyppi}_3 = \mathit{tyyppi}'_3 \quad \ldots \quad \text{ja} \quad \mathit{tyyppi}_k = \mathit{tyyppi}'_k.$$

Tyypinpäättelyesimerkissämme yhtälö (☞ 4.5.2)

$$\tau_1 \texttt{ -> } \tau_2 = \tau_{12} \texttt{ -> } \tau_{10}$$

siis purkautuisi yhtälöiksi

$$\tau_1 = \tau_{12}$$
$$\tau_2 = \tau_{10}$$

ja niin edelleen.

4. Jos tällaisen yhtälön vasen ja oikea puoli

   - alkavatkin eri tyyppikonstruktoreilla $C \neq C'$
   - tai sisältävätkin eri määrän *tyyppejä*

   niin silloin tämä yhtälöryhmä *ei ratkea*.

   Vastaavasti tämän tyyppiyhtälöryhmän synnyttänyt Haskell-lähdekoodi *ei läpäise tyypinpäättelyä*.

5. Jos yhtälöistä muodostuu ketju

$$x = \cdots = C \ \ldots \ x \ \ldots$$

jonka x esiintyy oman tyyppinsä sisällä, niin silloinkaan tämä yhtälöryhmä ei ratkea:

- Tyypin x kirjoitusasu olisi *äärettömän* pitkä:

$$C \ \ldots \ (C \ \ldots \ (C \ldots) \ \ldots) \ \ldots$$

- Esimerkiksi

$$x \neq y \text{ -> } x$$

koska tyyppi x olisi auki kirjoitettuna

$$y \text{ -> } y \text{ -> } y \text{ -> } \ldots$$

- Tyypinpäättelyn yleistysvaiheessa:
  - Kysymys "Täytyikö tyyppimuuttujalle a_i antaa arvo?" on "Luotiinko jokin yhtälö $a_i = \eta$?"
  - Jos luotiin, niin *poistetaan* a_i korvaamalla sen jokainen esiintymä tyypillä η.
  - Näin käsitellään jokainen $a_1, a_2, a_3, \ldots, a_r$ joka syntyi yleistettävän lausekkeen käsittelyn aikana.
  - Sen jälkeen kysymys "Mikä on tyyppi β_i'?" saadaan seuraavasti:
    1. Aloitetaan tyyppimuuttujasta $\tau = \beta_i$.
    2. Jos τ:ssa esiintyy tyyppimuuttuja b, jolle on yhtälö $b = \delta$, niin korvataan nämä esiintymät δ:lla.

4.6 Tyyppiluokat

- Tyyppiluokka on sellainen kokoelma tyyppejä, joilla kaikilla on jokin *sama rajapinta*.
- Lähin Java-vastine ovatkin `interface`-luokat.
- Saman luokan kaikki tyypit tarjoavat jonkin saman operaation eli metodin.
  - Esimerkiksi kaikki järjestetyt tyypit tarjoavat jopa *saman nimisen* metodin (<=)...
  - mutta jokaisella järjestetyllä tyypillä on sille oma toteutuksensa.

- *Tyyppikonteksti* ilmaistaan syntaktisesti rakenteella

  (*ehto*,*ehto*,*ehto*,...,*ehto*) => ...

  - Jos *ehtoja* on vain yksi, niin sulut saa jättää poiskin.
  - Jokainen *ehto* on muotoa *Luokka tyyppi* jossa *Luokka* on jonkin tyyppiluokan nimi (ISOlla alkukirjaimella).
  - Silloin tämän *tyypin* täytyy kuulua tähän *Luokkaan*.
  - Käytännössä se siis rajoittaa tässä *tyypissä* esiintyvien tyyppimuuttujien luvallisia arvoja.

- Esimerkiksi järjestämisfunktion tyypissa

```
List.sort :: (Ord a) => [a] -> [a]
```

on rajoitus, että tyyppimuuttujalle `a` voi antaa vain sellaisen tyypin, joka kuuluu luokkaan `Ord` — eli joka on *järjestetty*.

- Tällainen tyyppikonteksti voi esiintyä

funktion (tms.) tyypin määrittelyissä yllä nähtävässä kohdassa

uuden tietotyypin määrittelyssä heti `data`-sanan jälkeen, jolloin se rajoittaa konstruktoreiden tyyppejä.

- Tyyppiluokat on suunniteltu toimimaan yhteistyössä tyypinpäättelyn (☞ 4.5) kanssa.

- Siitä seuraa joitakin rajoituksia:

  – Jos tyyppikonteksti sisältää jonkin tyyppimuuttuujan u, niin tämän saman u pitää esiintyä myös implikaationuolen '=>' oikealla puolella.

    Koska nuolen oikealla puolella esiintyy se varsinainen tyyppi, jonka päättelystä saadaan tietoa muuttujan u arvoista.

  – Eri luokkien metodeilla pitää olla eri nimet.

    Jotta tyypinpäättelyn nimisääntö olisi yksiselitteinen.

4.6.1 Johdetut jäsenyydet

- Yksinkertaisin tapa liittää `data`-määritelty tyyppi haluttuihin tyyppiluokkiin on päättää sen määritelmä määreeseen

  `deriving` (*tyyppiluokat*)

  jossa *tyyppiluokat* on pilkuin eroteltu luettelo kyseisten tyyppiluokkien nimistä.

- Jos luokkia on luettelossa vain yksi, niin sulut voi jättää pois.

- Tämä tapa soveltuu kuitenkin vain tietyille Haskellin valmiina tarjoamille tyyppiluokille. . .

- ja käyttää tällaiselle luokalle kiinnitettyä *oletustapaa* olla luokan jäsen.

- Oletustavan sijasta voi jäsenyyden määritellä itsekin (☞ 4.6.2).

- Kuvaillaan seuraavaksi tällaiset luokat ja oletustavat vakiokirjastosta `Prelude`.

`Eq` t: Tyypille t laaditaan oletusarvoinen samuustesti

```
(==) :: t -> t -> Bool.
```

- Oletustestissä x `==` y jos ja vain jos arvoilla x ja y on sama konstruktori ja niiden kentätkin ovat pareittain `(==)`.
- Erityisesti siis `deriving Eq` on mahdollista *vain* jos myös tyypin t kaikkien kenttien tyypeillä u on `Eq` u.
- Samalla tulee määriteltyä myös sen vastakohta `(/=)`.

`Ord` t: Tyypille t laaditaan oletusarvoinen järjestys ja sen operaatiot kuten (<=).

- `Ord` t vaatii että `Eq` t.
- Oletusjärjestyksessä x <= y jos ja vain jos

  joko arvon x konstruktori on ennen arvon y konstruktoria tyypin t `data`-määritelmässä

  tai konstruktorit ovat samat ja kentät pareittain (<=) vasemmalta oikealle.
- Esimerkiksi listojen (☞ 3) sanakirjajärjestys:
  - `[]` <= kaikki listat
  - `x:xs` <= `y:ys` jos `x` <= `y` ja `xs` <= `ys`.

Enum t**:** t on *lueteltu* (enumerated) tyyppi.

- Vertaa Java 5.0:n `enum`.
- `data`-määrittelyn konstruktoreissa ei saa olla kenttiä, esimerkiksi:

```
data Vko
    = Ma | Ti | Ke | To | Pe | La | Su
    deriving (Show,Enum)
```

- Sen jälkeen tällä tyypillä voi käyttää aritmeettisia sekvensseja (☞ 3.7), esimerkiksi:

  `[Su,To ..]` on nyt sallittu ja antaa `[Su,To,Ma]`.

`Bounded` t: Tyypin t arvoissa on pienin ja suurin

```
minBound :: t
maxBound :: t
```

- Esimerkiksi `maxBound :: Int = 2147483647` on toteutuksen suurin äärellinen kokonaisluku.
- Jos tietotyypillä on vain yksi konstruktori C ja sen jokainen kenttä on myös `Bounded`, niin silloin `minBound` on kentittäin

  C `minBound minBound minBound ... minBound`

  ja `maxBound` vastaavasti.

- Esimerkiksi siis `minBound :: (Int,Int)` on `(-2147483648,-2147483648)`.

- Jos tyyppi t on lueteltu kuten `Enum`issa, niin `minBound` ja `maxBound` ovat sen ensimmäinen ja viimeinen konstruktori.

- Viikonpäiväesimerkissä siis

```
data Vko
    = Ma | Ti | Ke | To | Pe | La | Su
    deriving (Show,Bounded)
minBound :: Vko = Ma
maxBound :: Vko = Su
```

`Show` t: Tyypin t arvon x voi kirjoittaa merkkijonoksi.

- Erityisesti Haskell-tulkki voi tulostaa tyypin t arvoja käyttäjälle vain jos se voi ensin kirjoittaa ne merkkijonoiksi.

- Merkkijonon kirjoittaa funktio

```
show :: t -> String
```

  jonka `deriving Show` siis tuottaa.

- Oletustulostusasu noudattaa Haskellin syntaksia.

- Jos ohjelmoija haluaakin tyypilleen t oman tulostusasun, niin hän voi määritellä käsin (☞ 4.6.2) perusfunktion

  $$\texttt{showsPrec}\ d\ x\ r$$

  jossa $x :: t$ on tulostettava arvo.

- $0 \leq d \leq 11$ on presedenssi (☞ 2.3).
  - Operaattorisyntaksi sallitaan myös konstruktorille.
  - Konstruktori on sellainen täsmälleen silloin kun sen nimi alkaa kaksoispisteellä ':'.
  - Erityisesti listojen (☞ 3) konstruktori on tällainen.

- Merkkijono r on kerääjäparametri (☞ 4.3.2):
  - Arvon x merkkijonoesitys liitetään merkkijonon r *alkuun*.
  - Toisin sanoen seuraavan ehdon täytyy päteä:

    showsPrec d x r ++ s = **showsPrec** d x (r ++ s)

  - Kerääjäparametrilla r tehdään (tässäkin) puumaisen rakenteen x "listoitus" lineaarisessa ajassa (neliöisen sijasta).

`Read` t: Tyypin t arvon x voi lukea merkkijonosta.

- Haskell-tulkkia varten *ei* tarvitsekaan sanoa omille `data`-tyypeilleen `deriving Read`:

  tulkki lukee komentoriviltään sen arvoja Haskell-lähdekoodin mukaisessa kirjoitusasussa.

- Jos ohjelmoija onkin määritellyt tyypilleen t oman `Show`-asun, niin silloin hänen on syytä määritellä myös sellainen `Read`-asu, joka lukee sitä.

  – Tätä periaatetta kutsutaan *luku-kirjoitus-muuttumattomuudeksi* (read-write invariance).

  – Sen voi ajatella sanovan, että viittausten läpikuultavuus (☞ 1.3.1) säilyisi myös syötössä ja tulostuksessa.

- Keskeinen määriteltävä funktio on `readsPrec` d s:
  - Presedenssi d on kuten funktiossa `showsPrec`.
  - Se yrittää lukea arvon $x :: t$ merkkijonon s alusta.
  - Palautusarvo on samanlainen kuin jäsenninyhdistelijöissä (☞ 3.6):
    * Se on lista pareja (x_i, s_i) joissa $x_i :: t$ on onnistuneesti luettu arvo ja s_i lukematta jäänyt loppuosa merkkijonosta s.
    * `Show`-tulostussyntaksi voi olla moniselitteinen (vaikka sitä kannattaisikin varoa), siksi lista kaikista eri vaihtoehdoista.
    * Epäonnistuminen on `[]`.

- Silloin luku-kirjoitus-muuttujattomuus saa muodon

  (x,"") kuuluu listaan `readsPrec` d (`showsPrec` d x "")

  eli ”arvon x `Show`-tulostusasu voidaan `Read`-lukea muun muassa takaisin arvoksi x”.

- Jos merkkijono s koostuu varmasti tasan *yhdestä yksikäsitteisestä tyypin* t *arvosta* x *ja tyhjistä* merkeistä (= välilyönneistä, rivinvaihdoista,. . .) niin apufunktiokutsu `read` s antaa tämän arvon x.

  Muuten sen kutsu johtaa suoritusaikaiseen virheeseen.

`Ix` t: Tyypin t arvoilla voi indeksoida taulukkoa (☞ 3.8).

- Tämä luokka onkin omassa moduulissaan (☞ 6) `Ix`.

  Myös taulukkomoduuli `Array` lataa sen automaattisesti ohjelmoijan vaivojen säästämiseksi.

- Jos tyyppi t on lueteltu kuten `Enum`issa, niin sen arvoja voi käyttää indekseinä luonnolliseen tapaan.

- Jos tyypillä t on vain yksi konstruktori C, ja sen jokaisen k kentän tyyppi t_i on `Ix` t_i, niin sen arvoja

  $$C\ x_i\ x_2\ x_3\ \ldots\ x_k$$

  voi käyttää indekseinä.

- Erityisesti kun C on monikon konstruktori $(\underbrace{,,,\ldots,}_{k-1 \text{ kpl}})$ niin saadaan k-ulotteiset taulukot.

4.6.2 Tyyppiluokkaan liittyminen

- Jos `deriving`-mekanismia ei voida tai haluta käyttää, niin `instance`-määrittelyllä ilmaistaan, että tämä tyyppi halutaan tämän tyyppiluokan jäseneksi.

- Syntaksi on

  `data` *Tyyppi* $u_1 \ldots u_k$ `=` ...

  `instance` *konteksti* `=>` *Luokka* `(`*Tyyppi* $u_1 \ldots u_k$`)`
  `where` ...

  jonka *konteksti* voi rajoittaa tyyppimuuttujia $u_1, \ldots, u_k$.

- `where`-osan sisällä määritellään tämän *Luokan* metodien toteutukset tälle *Tyypille*.

- Metodien tyyppejä ei kuitenkaan kirjoiteta:

  Ne tulevat *Luokan* määrittelystä (☞ 4.6.3).

- Määritellään esimerkkinä tyyppikonstruktori `Nurinpain a` jonka mahdollinen suuruusjärjestys on päinvastainen kuin parametrityypin `a`.

```
1   data Nurinpain a
2       = Nurin a
3         deriving (Eq,Show)
4
5   instance Ord a => Ord (Nurinpain a)
6       where (Nurin x) <= (Nurin y) = x >= y
```

- Rivi 5 sanoo: "*Jos* tyyppi `a` on järjestetty *niin* myös tyyppi `Nurinpäin a` on järjestetty."

- Rivillä 6 ilmaistaan *miten* se on tässä tapauksessa järjestetty.

- Rivillä 1 konteksti `Ord a =>` olisikin sanonut: "Jos tyyppi `a` on järjestetty niin tyyppi `Nurinpäin a` on *olemassa*."

- Oliotermein `instance` ilmoittaa miten tietty polymorfinen *Tyyppi* (eli konkreettisten termien joukko) toteuttaa tämän *Luokan* spesifioiman rajapinnan.

- Tässä esimerkissä olisimme voineet käyttää `data`-määrittelyn tilalla `newtype`-määrittelyä:
  - Sillä voi määritellä uuden tyypin t, jolla on *vain yksi konstruktori* C, jolla taas on *vain yksi kenttä*.

    Olkoon t' tuon ainoan kentän tyyppi.
  - Näin konstruktorilla C voi ilmoittaa, että nyt onkin kyseessä uusi tyyppi t, vaikka sen sisäinen toteutus onkin vanhaa tyyppiä t'.
  - Erityisesti t voidaan nyt liittää tyyppiluokkiin eri tavalla kuin t'.
  - Samalla tämä on vihje Haskell-toteutukselle, että käännetyn koodin ei tarvitse enää sisältää konstruktoria C.

4.6.3 Uuden tyyppiluokan määrittely

- Uuden tyyppiluokan määrittelysyntaksi on

  `class` *konteksti* `=>` *Luokka* u
  `where ...`

  jossa tyyppimuuttuja u edustaa tähän luokkaan kuuluvaa tyyppiä.

- Sen *konteksti* ilmoittaa mitä rajoitteita tämän tyyppimuuttujan u arvoille asetetaan — eli mihin muihin luokkiin sen pitää kuulua voidakseen kuulua myös tähän.

- `where`-osa koostuu *vähintään* tämän luokan metodien nimien ja niiden tyyppien esittelyistä.
  - Jokaisessa tyypissä *täytyy* käyttää myös tätä u: Miksi muuten tämä metodi kuuluisi tähän luokkaan?
  - Minkään metodin tyyppikontekstissa *ei saa* mainita tätä u (mutta muita saa): Kaikki tyypin u ehdot täytyy mainita jo `class`-*kontekstissa*.
- Lisäksi se *voi* sisältää metodien *oletusmäärittelyjä*.
  - Siltä varalta että `instance`-määrittely (☞ 4.6.2) jättää jotakin määrittelemättä.
  - Siinä saa käyttää kaikkia tämän *Luokan* ja sen *kontekstin* luokkien metodeja.

- Tarkastellaan esimerkkinä luokan `Ord` määrittelyä vakiokirjastosta `Prelude`.

- Rivillä 4 määritellään, että kuuluakseen luokkaan `Ord` tyyppimuuttujan `a` arvon täytyy kuulua myös luokkaan `Eq`, eli omata myös yhtäsuuruusmetodin `(==)`.

  Tämä vaaditaan siksi, että x `<=` y `&&` y `<=` x on itse asiassa x`==`y.

- Riveillä 5–7 esitellään luokan metodit tyyppeineen.

- Riveillä 9–24 määritellään jokaiselle niistä oletus siten, että riittää määritellä joko `compare` tai `(<=)`: muut käyttävät niitä.

```
1   data  Ordering  =  LT | EQ | GT
2             deriving (Eq, Ord, Enum, Read, Show, Bounded)
3
4   class  (Eq a) => Ord a  where
5       compare              :: a -> a -> Ordering
6       (<), (<=), (>=), (>) :: a -> a -> Bool
7       max, min             :: a -> a -> a
8
9       compare x y
10          | x == y    =  EQ
11          | x <= y    =  LT
12          | otherwise =  GT
13
14      x <= y          =  compare x y /= GT
15      x <  y          =  compare x y == LT
16      x >= y          =  compare x y /= LT
17      x >  y          =  compare x y == GT
18
19      max x y
20          | x <= y    =  y
21          | otherwise =  x
22      min x y
23          | x <= y    =  x
24          | otherwise =  y
```

- Oliotermein `class` esittelee rajapintaluokan:
  - *Konteksti* listaa tämän luokan yläluokat.
  - *Moniperintä* (multiple inheritance) on siis mahdollista.
  - Tämä luokka *laajentaa* näitä yläluokkiaan uusilla metodeilla.
  - Koska nämä uudet metodit ovat eri nimisiä kuin vanhat, niin moniperintä on yksinkertaista.
  - Tämä luokka *ei voi määritellä uudelleen* yläluokan metodia:

    Kunkin luokan metodit määritellään täydellisesti ja lopullisesti silloin kun siihen liitytään (☞ 4.6.2).

4.6.4 Konstruktoriluokat

- Haskellin tyyppijärjestelmä salliin myös sellaisen tyyppiluokkamäärittelyn (☞ 4.6.3), jossa määritelläänkin ehtoja tyyppi*konstruktorille*.

- Sellaista tyyppiluokkaa kutsutaankin (tyyppi)*konstruktoriluokaksi* (constructor class).

- Otetaan esimerkiksi `Preluden` tyyppiluokka `Monad`.

```
1   class  Monad m  where
2       (>>=)  :: m a -> (a -> m b) -> m b
3       (>>)   :: m a -> m b -> m b
4       return :: a -> m a
5       fail   :: String -> m a
6
7           -- Minimal complete definition:
8           --      (>>=), return
9       m >> k  =  m >>= \_ -> k
10      fail s  = error s
```

- Huomaa luokkamuuttujan `m` paikka metodien tyypeissä:
  - Se onkin sillä kohdalla, jossa esiintyisi tyyppikonstruktori kuten `Maybe`
  - koska se saa tyyppiargumentin `a`.

- Tyyppikonstruktori `Monad` kuuluukin tähän luokkaan:

```
instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
    return           =  Just
    fail s           =  Nothing
```

- Siten x >>= f tarkoittaa: "Jos $x =$ `Just` y niin laske $f\ y$; jos taas $x =$ `Nothing`, niin sivuuta f."

- Muita `Monad`eja ovat esimerkiksi listat (☞ 3) (joilla nämä metodit muistuttavat jäsenninyhdistelijöitä (☞ 3.6)) ja I/O-toiminnot (☞ 5).

- Haskell 98 -standardissa konstruktoriluokan muuttujalla (kuten tässä `m`) sai olla vain yksi parametri (kuten tässä `a`).

- Esimerkiksi `Array` (☞ 3.8) onkin 2-parametrinen tyyppikonstruktori:

  1. indeksityyppi

  2. sisältötyyppi.

- Siksi Haskell'-standardiehdotus ja GHC sallivatkin myös moniparametriset konstruktorit luokissa.

- Tyyppikonstruktoreilla onkin omat ”tyypit” eli *lajit* (kinds):
  - tavalliset tyypit ovat lajia `'*'`
  - 1-parametriset tyyppikonstruktorit kuten `Maybe` ovat lajia `* -> *`
  - 2-parametriset kuten `Array` taas lajia `* -> * -> *`,...
- Haskell 98 käyttää lajeja vain sisäisesti, mutta Haskell' ja GHC sallivat myös niiden kirjoittamisen näkyviin.

- Kyseessä on tyyppijärjestelmän seuraava abstraktiotaso:

  1. tyyppi on kokoelma arvoja, jotka on luotu erilaisilla arvokonstruktoreilla

  2. tyyppiluokka on kokoelma tyyppejä, jotka on luotu erilaisilla tyyppikonstruktoreilla

  3. konstruktoriluokka on kokoelma erilaisia tyyppikonstruktoreita, joilla on sama laji.

5 Monadinen syöttö ja tulostus

- Jo kurssin alussa (☞ 1.3.4) huomattiin, että tilattomassa ohjelmoinnissa syöttö ja tulostus ovat ongelmallisia:

  tulostus muuttaa ja

  syöte riippuu

  *ulkomaailman tila*sta.

- Tämän vuoksi meidän täytyy pystyä määräämään se *järjestys* jossa syöttö- ja tulostustoiminnot tapahtuvat ohjelmassamme.

- Eräs Haskell 98 -standardin keskeisistä uutuuksista oli soveltaa jo olemassaolevaa tyyppijärjestelmää järjestyksen määräämiseen!

- Oivalluksen takana oli tapa, jolla tutkija Eugenio Moggi sovelsi 1980-luvun lopulta alkaen kategoriateorian (☞ 3.4.3) rakennetta nimeltä *monadi* abstraktin laskettavuuden tutkimukseen.

- Sen perusteella havaittiin, että I/O voidaan mallintaa sopivana tapauksena konstruktoriluokasta `Monad` (☞ 4.6.4).

- Sittemmin huomattiin, että monet muutkin asiat ovat sen saman luokan eri tapauksia. Uusia asioita löydetään yhä. . .

- Ulkomaailman tila on *piilossa* Haskell-ohjelmalta: Sen kätkee tyyppikonstruktori `IO` t kuten `Maybe`-esimerkissä kätkettiin `Nothing`.

- Ohjelmassa voi käyttää I/O-*toimintoja* (actions) kuten

```
print  :: Show a => a -> IO ()
readLn :: Read a => IO a
```

- Toiminnon tyyppi on `IO` u jossa u on sen vastaustyyppi:

  Tulostus kuten `print` x on toiminto "tulosta arvon x merkkijonoesitys (vakiokanavaan `stdout`)" (☞ 4.6.1).

  Syöttö kuten `readLn :: Int` on toiminto "lue seuraava rivi (vakiokanavasta `stdin`) kokonaislukuna".

- Monadioperaatio (>>=) liittää näitä toimintoja peräkkäin:

```
readLn >>= (\ x -> print (x+1))
```

on isompi toiminto (tyyppiä `IO ()`) joka

1. ensin lukee rivin kokonaislukuna
2. sitten lähettää lukemansa luvun parametrina `x` kasvatettavaksi yhdellä ja tulostettavaksi.

- Siten syöttöä ja tulostusta tekevä Haskell-ohjelma on deklaratiivisesti ottaen vain määrittely massiiviselle I/O-toiminnolle — ei niiden suoritus tai suorituksen tulos!

- Synkronoinnin *yhden* toiminnon sisällä hoitaa `IO`:
  - tulostus pääsee alkamaan vasta kun se saa ulkomaailman tilan jota muokata
  - joten sen pitää odottaa, kunnes syötteen lukeminen on lopettanut saamansa alkutilan muokkaamisen.

- Entä *kahden eri* toiminnon välillä?
  - Laiska suoritus saattaisi lomittaa ne keskenään, vaikka ohjelmaa ei edes suoritettaisi rinnakkain.
  - Ainoa *suoritettava* toiminto on päälauseke (☞ 5.1).
  - Ei ole (turvallista) funktiota tyyppiä `IO` a `->` a joka siis pystyisi suorittamaan jonkin toisen toiminnon.

- Näin Haskell-ohjelma jakautuu kahteen osaan:

  Ulkomaailman kanssa kommunikoiva osa koostuu niistä funktioista

  – joiden tulostyyppi on jokin `IO` a

  – joita kutsutaan päälausekkeesta toisten sellaisen funktioiden kautta.

  Puhtaan funktionaaliseen osaan siirrytään siitä

  – kutsumalla funktiota, jonka tulostyyppi on jokin muu

  – joka palauttaa sen typpisen tuloksensa ulkomaailman kanssa kommunikoivan osan käyttöön (esimerkiksi tulostettavaksi).

- Ulkomaailman kanssa kommunikoiva `IO`-osa suoritetaan *käytännössä ahkerasti askel askeleelta*.

- Haskell-kielellä on kuitenkin vain laiska suoritus:
  - `IO`-monadi ei jätä laiskalle suoritukselle muuta mahdollisuutta kuin edetä kuten ahkera.
  - Esimerkiksi vaikka lausekkeessa

    putStrLn $ show *lista*

    tulostettava *lista* onkin muodostettu puhtaan funktionaalisessa osassa laiskana, niin sen koko sisältö pitää muodostaa ja tulostaa ennen kuin suoritus voi edetä seuraavaan ulkomaailman kanssa kommunikoivan osan askeleeseen.

5.1 Päälauseke

- Tarkastellaan (ensimmäistä kertaa tällä kurssilla!) Haskell-ohjelmaa, joka toimii *itsenäisesti käännettynä tulkin ulkopuolella*.

- Se tarvitsee kohdan, josta suoritus alkaa.

- Tämä kohta on *päälauseke*

```
main :: IO ()
```

- Päälauseke on siis I/O-toiminto, jonka vastausarvo on (), jonka voi siis sisällöttömänä jättää huomiotta.

- GHC kääntää Haskell-ohjelman komennolla

  `ghc --make` *tiedosto*

  – *tiedosto*`.hs` on se lähdekooditiedosto, joka sisältää päälausekkeen `main`

  – valitsimella `--make` kääntäjä `ghc` jäljittää tästä *tiedosto*sta alkaen ohjelman tarvitsemat moduulit (☞ 6) ja kääntää vain ne, joita on muutettu sitten edellisen käännöksen.

5.2 Imperatiivinen minikieli

- Haskellin I/O-ohjelmointi on siis periaatteessa toimintojen yhdistelemistä peräkkäin monadioperaatiolla (>>=).

- Koska tämä notaatio tulisi nopeasti varsin mutkikkaaksi, niin kieleen on lisätty *oma syntaksi* jolla on luontevaa ilmaista tämä(kin) *imperatiivisesti etenevä ohjelman osa*.

- Intuitiivisesti tämä notaatio lisää siis Haskell-kielen sisään pienen "perinteisen" ohjelmointikielen.

- Tämä syntaksi on muotoa

$$\begin{array}{ll} \texttt{do} & \mathit{komento}_1 \\ & \mathit{komento}_2 \\ & \vdots \\ & \mathit{komento}_k \end{array}$$

- Sen suoritus on tuttu:

  1. Ensimmäinen $\mathit{komento}_1$ saa ulkomaailman alkutilan, ja muokkaa sitä tällä I/O-toiminnolla.

  2. Toinen $\mathit{komento}_2$ saa tämän muokatun tilan, ja muokkaa sitä edelleen tällä I/O-toiminnolla.

  3. ...

```
1   module Main where
2   import IO
3   import Tekijat
4
5   main :: IO ()
6   main =
7       do rivi <- readLn
8          if rivi < 1
9             then return ()
10            else do putStrLn $ tulostin $ tekijat rivi
11                    main
```

- Funktio `Tekijät.tekijat` palauttaa saamansa luvun alkulukutekijät listana.

- `Tekijät.tulostin` muuntaa tekijälistan merkkijonoksi.

```
1   module Tekijat where
2   import Eratosthenes
3   import List
4
5   tekijat =
6       tekijat' alkuluvut
7       where tekijat' _ 1 =
8                 []
9             tekijat' p'@(p:ps) n
10                | r == 0 =
11                    p : tekijat' p' q
12                | otherwise =
13                    tekijat' ps n
14                where (q,r) =
15                          n `quotRem` p
16
17  tulostin :: [Int] -> String
18  tulostin =
19      concat . intersperse "*" . map show
```

- Rivin 17 tyyppimäärittely tarvittiin monomorfismirajoituksen (☞ 4.5.2) vuoksi.

- Rivillä 7 on *syötteenlukutoiminto* muotoa

  muuttuja <- *syötteenlukukutsu*

  – Kutsun tulostyyppi on `IO` a . . .

  – kun taas *muuttuja* :: a . . .

  – ja *muuttuja*an luetaan seuraava syöte muunnettuna tyyppiin a sen `Read`-metodilla (☞ 4.6.1).

- Notaation samuus listakoosteen generaattorin (☞ 3.7) kanssa on tarkoituksellista:
  - Listakin kuuluu samaan konstruktoriluokkaan `Monad`.
  - Samoin *muuttuja* näkyy tämän `do`-lausekkeen loppuun.
- Myös `do`-notaatiossa on samanlainen `let` kuin listakoosteessakin.

- Rivillä 10 on *tulostustoiminto:*
  - kutsutaan *tulostusfunktio*ta...
  - jonka tulostyyppi on `IO ()`...
  - jolloin "`() <-`" voidaan jättää turhana pois.

- *Tulostusfunktio*n parametri
  - on merkkijono
  - joka pohjimmiltaan saadaan tulostettavan arvon tyypin `Show`-metodilla (☞ 4.6.1).

- Rivi 11 osoittaa, että `do`-notaatiossa voi viitata
  - myös rekursiivisesti
  - funktioihin joiden tulostyyppi on jokin `IO` a.

- Kyseessä on kuitenkin yhä edelleen funktionaalinen eikä imperatiivinen kieli:

  Seuraavan rekursiokutsun rivillä 7 luetaan seuraava syöte *uuden* muuttujan `rivi` arvoksi, eikä päivitetä vanhaa muuttujaa uudella arvolla.

- Kutsu on takarekursiivinen (☞ 3.4.2) joten tämä päälauseke voi käsitellä mielivaltaisen monta syötettä täyttämättä niillä muistia.

- Rivillä 9 käytetään monadimetodia `return` päättämään päälausekkeen rekursio paluuarvolla `()`.

- Lisäksi kuten rivit 8–11 osoittavat, niin `do`-notaatiossa voi käyttää sellaisia Haskell-lausekkeita, joiden tulostyyppi kuuluu tähän monadiin.
  - Tämän `if`-lausekkeen kummankin haaran tyyppi on `IO ()`.
  - Sama `do` ei kuitenkaan jatku näissä haaroissa:

    Siksi rivillä 10 täytyy aloittaa uusi `do`-lauseke.
  - Rivillä 9 ei ole pakko aloittaa uutta `do`-lauseketta, koska `do` *ainoa komento* on se itse.

5.3 Tiedosto listana

- `do`-notaatio (☞ 5.2) mahdollistaa perinteisen I/O-ohjelmoinnin.
- Se on kuitenkin oleellisesti *ahkeraa ja tilaperustaista* suoritusta.
- Haskell tarjoaa myös *laiskan ja tilattoman* syötteen ja tulostuksen käsittelyn.
- Se soveltuu ohjelmiin, jotka lukevat yhdestä tekstitiedostosta (kuten oletussyötettä **`stdin`**) ja tulostavat toiseen (kuten oletustulosteeseen **`stdout`**).

- I/O-toiminto

  merkkijono `<- getContents`

  asettaa listan *merkkijono* sisällöksi *koko oletussyötteen*.

- Syötettä kuitenkin *luetaan laiskasti* vain sitä mukaa kuin ohjelma etenee listalla *merkkijono*.

- Tämä lista päättyy vasta aivan syötetiedoston lopussa (UNIXeissa `CTRL-d`, Windowseissa `CTRL-z`).

- Jos se taas ei vielä ole lopussa, mutta siinä ei vielä ole uusia merkkejä ohjelman luettavaksi, niin toteutus jää odottamaan käyttäjältä lisää syötettä.

- Syöte*merkkijono*a lukiessaan ohjelma voi tuottaa tulostetta, joka näkyy käyttäjälle heti sitä mukaa kun se valmistuu.

- Näin voimme kirjoittaa *interaktiivisenkin* ohjelman ikään kuin sen syöte olisi tavallinen lista.

- Haskell tarjoaakin kirjastofunktion

```
interact :: (String -> String) -> IO ()
```

  jonka argumentti on käyttäjän antama syöte tällaisena ja tulos hänelle näytettävä tuloste.

- Tällä kirjastofunktiolla esimerkkiohjelmamme voidaan kirjoittaa seuraavasti.

- Edellinen versio päättyi, kun käyttäjän syöttämä `rivi` oli < 1.

- Tämä versio taas päättyy, kun syöte loppuu.

```
1   module Main where
2   import IO
3   import Tekijat
4
5   main :: IO ()
6   main =
7       interact (unlines .
8                 map (tulostin . tekijat . read) .
9                 lines)
```

5.4 Poikkeuksista

- Haskell tarjoaa I/O-virheiden käsittelyyn *poikkeukset* (exceptions).

- Ohjelmoija ei voi määritellä omia uusia poikkeuksiaan, vaan on pärjättävä niillä, jotka Haskell tarjoaa.
  - Uudenlaisten poikkeusten salliminen rikkoisi Haskellin perusideaa, että tietotyypin kaikki vaihtoehdot kuvataan kokonaisuudessaan jo sen määrittelyhetkellä.
  - Esimerkiksi SML-kieli sallii uusien poikkeusten määrittelyn, mutta siellä poikkeustyypii onkin perustavasti erilainen kuin tavalliset tyypit.

- Näiden ennalta määriteltyjen poikkeusten joukossa on ohjelmoijaa varten oma poikkeus ja sen nostava kirjastofunktio

```
userError :: String -> IOError
```

  jonka parametriksi voi antaa virhetilannetta kuvailevan merkkijonon.

- Käsittelemätön poikkeus päättää koko ohjelman suorituksen (virheilmoitukseen).

- Poikkeusten käsittelyyn on kirjastofunktio

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

1. joka siis yrittää tehdä ensimmäisenä parametrinaan saamansa I/O-toiminnon
2. mutta poikkeuksen sattuessa sen aikana kutsuukin jälkimmäistä parametriaan käsittelemään virhetilanteen.

- Poikkeuksia voi siis käsitellä vain `IO`-monadissa.

  Tämä onkin perusteltua, onhan poikkeusten synty- ja käsittelyjärjestys olennaisesti sarjallinen käsite.