3.4.1 Funktion valitsemisen vaikeudesta

- Saman funktion eri muotojen *ajan*tarpeessa voi tietenkin olla suuria eroja:
  - Olkoot edellä (☞ 3.4) $\oplus = (+\!\!+)$ ja $0_l = 0_r = \texttt{[]}$.
  - Kumpikin toteuttaa vakiokirjaston funktion `concat`.
  - `foldr`-toteutus toimii *lineaarisesti*, `foldl`-toteutus toimisi taas *neliöisesti*.
  - Ero johtuu siitä, että $(+\!\!+)$ käy aina läpi koko ensimmäisen argumenttinsa.
- Toisaalta *tilan*tarve riippuu suoritusmekanismista.

Ahkerassa suoritusmekanismissa `foldl` olisi usein *parempi:*

- Se tekisi operaatiot $\oplus$ heti.
- Siten se kokoaisi välitulosta jo kulkiessaan listaa a pitkin.
- `foldr` taas kopioisi koko listan a *rekursiopinoon*, ennen kuin se alkaisi kokoamaan lopputulosta.

Laiskassa `foldl` olisikin usein *huonompi:*

- Siinä operaatiot $\oplus$ viivästetään.
- Se siis kopioisi koko listan a *viivästetyksi lausekkeeksi*, ennen kuin se alkaisi kokoamaan lopputulosta.

- Haskell-ohjelmoija siis kirjoittaa mieluummin `foldr` kuin `foldl`

  koska se sopii paremmin yhteen laiskuuden kanssa.

- `foldl` on kuitenkin hyödyllinen silloin kun $\oplus$ (ei ole liitännäinen vaan) toimii "luonnostaan vasemmalta":

  Esimerkiksi operaatio "muunna numeromerkkien jono vastaavaksi luvuksi" on luonnostaan

```
1   import Char
2   arvo :: String -> Int
3   arvo =
4       foldl (\ ds d -> ds*10+d) 0 . map Char.digitToInt
```

- Haskell-kääntäjän ahkeruusanalysoija (strictness analyser) voi jopa huomata, ettei näin yksinkertaista apufunktiota kannata viivästää.

3.4.2 Takarekursio

- Perinteisissä ohjelmointikielissä on erilliset silmukkarakenteet (`for`, `while`) rekursion lisäksi.
- Silmukkarakenteita suositellaan käytettäväksi rekursion sijasta, koska ”rekursio kasvattaa pinoa”.
- Pelkästään rekursiota käyttävissä kielissä (kuten Haskell) ollaan sen sijaan kiinnostuneita *häntä- eli takarekursiosta* (tail recursion), koska sellainen voidaan toteuttaa *kasvattamatta pinoa*.
- Suuria syötteitä käsittelevät funktiot kannattaa valita takarekursiiviksi, jos mahdollista.

- Ideana on löytää *viimeinen kutsu jonka tämä funktio tekee* ennen paluutaan.

- Siitä kutsusta voidaankin palata suoraan tämän funktion kutsujaan käymättä välillä turhaan tässä funktiossa "kääntymässä".

- Tämän "oikopolun" vuoksi ei pinoa tarvitse kasvattaa.

- Tätä kääntäjän tekemää optimointia kutsutaan *takarekursion poistoksi* (tail recursion/call elimination).

- Tavallisissa ohjelmointikielissä se on toteutuskohtainen lisäpiirre; rekursioon nojaavissa välttämättömyys — ja usein kielistandardin toteutuksiltaan vaatima.

- Esimerkiksi funktion `foldr` (☞ 3.4) rekursiivisen säännön viimeinen teko on kutsua parametrina saatua funktiota `f`.

- Funktiossa `foldl` (☞ 2.2.6) reursiivisen säännön viimeinen teko onkin kutsua *itseään*.

- Tällaista rekursiokutsua nimitetään *häntä- tai takarekursiiviseksi kutsuksi* (tail(-recursive) call).

- Koko funktiota kutsutaan takarekursiiviseksi, jos kaikki sen rekursiiviset kutsut ovat takarekursiivisia.

- Koko `foldl` on siis takarekursiivinen funktio, koska sen ainoa rekursiivinen kutsu on.

- *Ahkerassa* funktionaalisessa ohjelmoinnissa takarekursio ja funktioiden kirjoittaminen takarekursiivisessa muodossa on ohjelmoijan keskeinen työväline aputilan tarpeen hallinnassa.

- Edellä (☞ 3.4.1) kuitenkin havaittiin, että *laiskassa* funktion aputilan käyttö riippuukin paitsi rekursiopinosta myös viivästyksistä.

3.4.3 Luonnolliset muunnokset

- Funktion `foldr` (☞ 3.4) voi ajatella tekevän syötelistastaan sellaisen lausekkeen, jossa
  - jokaisen operaattorin `':'` tilalla on operaattori $\otimes$
  - päättävän tyhjän listan `'[]'` tilalla on arvo q

  ja palauttavan tämän lausekkeen arvon (laiskana).

- `foldr` kuvaa siis "luontaisen listarekursion":
  - Listalla $z{:}zs$ tehdään $z \otimes$ rekursiivisesti zs.
  - Rekursion lopuksi palautetaan q.

- Siten monille muille listojen funktioille saadaan yksinkertainen `foldr`-toteutus:

  esimerkiksi `map` (☞ 2.2.6) voitaisiin ilmaista

```
map f = foldr ((:) . f) []
```

- Jokaiseen induktiivisesti määriteltyyn tietotyyppiin liittyy sen oma tällainen "luontainen läpikäyntifunktio" jolla on

  – jokaiselle perustapaukselle parametri, jolle läpikäynti antaa sen tietokentät (tässä `[]` ja q, ei tietokenttiä)

  – jokaiselle induktiiviselle tapaukselle parametri, jolle läpikäynti antaa sen tietokentät siten, että induktiivisissa kentissä on rekursiivinen läpikäyntitulos (tässä `':'` ja $\otimes$).

- Hyvin abstrakti matematiikan ala nimeltään *kategoriateoria* (category theory) tutkii (mm.) *luonnollisia muunnoksia* (natural transformations).

- Esimerkiksi tällaiset luontaiset läpikäyntifunktiot ovat luonnollisia muunnoksia.

- Niinpä kategoriateorian ideoita ja tuloksia voidaan soveltaa tietojenkäsittelyssäkin funktionaalisen ohjelmoinnin toimiessa välittäjänä.

3.5 Äärettömät listat

- Haskellin laiskuus mahdollistaa *periaatteessa äärettömän pitkän* listan käytön ohjelmassa. . .
- . . . kunhan ohjelma lukee siitä vain *äärellisen monta ensimmäistä* alkiota.
- Tämä mahdollistaa sellaisen tuottaja-kuluttaja-ohjelmoinnin (☞ 1.3.6) jossa listan kuluttajan ei tarvitsekaan ilmoittaa tuottajalle montako alkiota tarvitaan.
- Kuluttajan riittää ilmoitella vain ”anna vielä yksi lisää” kunnes se on saanut tarpeekseen.

- Tarkastellaan esimerkkinä *alkulukuja* (primes):
  - Kokonaisluku $p > 1$ on alkuluku, jos sen jakavat tasan vain 1 (triviaalisti) ja p itse.
  - Alkulukujen lista alkaa `[2,3,5,7,11,...]`.

- Erilaisia kuluttajia:

  1. "Mitkä ovat n ensimmäistä alkulukua?" Silloin kuluttaja tietää lukumäärän n.

  2. "Mikä on pienin alkuluku $\geq m$?"

     Silloin kuluttaja voi(si) vain arvioida tarvittavaa lukumäärää n saamansa luvun m pohjalta.

  Alkulukujen *tuottajan* ei tavitse tietää lukuja n ja m.

- Tuotetaan kaikkien alkulukujen ääretön lista *Eratostheneen seula*lla.

  Sift the twos and sift the threes,
  the Sieve of Eratosthenes.
  When the composites sublime
  the numbers that remain are Prime.

  (anon.)

  – Ei nopein menetelmä...

  – ... mutta havainnollinen, kun voidaan käyttää äärettömiä listoja.

- Periaate:

  1. Aloitetaan äärettömällä listalla $s_1 = [2, 3, 4, \ldots]$.
  2. Jaetaan nykyinen lista osiin $s_i = p_i : t_i$.
  3. Ensimmäinen alkio p_i on i. alkuluku.
  4. Seuraavan kierroksen lista e_{i+1} saadaan poistamalla loppuosasta t_i kaikki luvut q jotka p_i jakaa tasan.

- Esimerkki:

  1. $p_1 = 2$ ja $e_2 = [3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \ldots]$
  2. $p_2 = 3$ ja $e_3 = [5, 7, \cancel{9}, 11, 13, \cancel{15}, 17, 19, \cancel{21}, 23, 25 \ldots]$

```
1   module Eratosthenes where
2   alkuluvut :: [Int]
3   alkuluvut =
4       seulo (iterate (+1) 2)
5       where seulo (p:t) =
6                 p : seulo (filter (not . tasan) t)
7                 where tasan q =
8                           q ‘mod‘ p == 0
```

- Siis kaikki `alkuluvut` on äärettömän pitkä *listavakio*.
- Laiskuus näkyy jo siinä, ettei Haskell-tulkki jääkään ikuiseen silmukkaan jo sitä määriteltäessä.
- Rivi 1 tekee tästä tiedostosta `Eratosthenes.hs` moduulin (☞ 6).

```
1   import Eratosthenes
2
3   kuluttaja1 n =
4       head (drop (n-1) alkuluvut)
5
6   kuluttaja2 m =
7       head (dropWhile (< m) alkuluvut)
```

- Nyt molemmat `kuluttajat` voitiin ohjelmoida samalla periaatteella

  "ohita edelliset `alkuluvut` ja palauta seuraava"

  eikä tuottajalle tarvinnut ilmoittaa montako alkulukua sen pitäisi tuottaa.

- Edellä määrittelimme äärettömän listan rekursiivisen apufunktion `seulo` avulla.

- Laiskuus salli myös äärettömän listan määrittelyn *suoraan viittaamalla siihen itseensä*.

- Tarkastellaan esimerkkinä *Denningin lukujonoa:*

  Se koostuu kaikista luvuista muotoa

  $$2^p \cdot 3^q \cdot 5^r$$

  missä $p, q, r \in \mathbb{N}$ ja se

  – on suuruusjärjestyksessä

  – ei toista mitään lukua montaa kertaa.

- Olkoon tämän (vielä tuntemattoman) listan nimi Δ.
  - Δ alkaa $1 : \ldots$ koska $2^0 \cdot 3^0 \cdot 5^0 = 1$.
  - Kaikki listan $\Delta_2 = \text{map}\ (2\cdot)\ \Delta$ luvut kuuluvat siihen.
  - Δ_2 on suuruusjärjestyksessä ja toistoton, koska Δ täyttää nämä ehdot.
  - Δ_2 ei kuitenkaan sisällä lukua 1.
  - Samoin Δ_3 ja Δ_5.
  - Δ koostuu näiden listojen $\Delta_2, \Delta_3, \Delta_5$ luvuista.
  - Siis pätee yhtälö $\Delta = 1 : \text{mrg}\ \Delta_2\ (\text{mrg}\ \Delta_3\ \Delta_5)$ missä mrg lomittaa 2 järjestettyä toistotonta listaa yhdeksi.

- Tämä yhtälö kelpaa sellaisenaan Haskell-vakiolistan määrittelyksi:

```
denning =
    1 : mrg (map (2 *) denning)
            (mrg (map (3 *) denning)
                 (map (5 *) denning))
    where mrg (x'@(x:xs)) (y'@(y:ys))
              | x < y =
                  x : mrg xs y'
              | x > y =
                  y : mrg x' ys
              | otherwise =
                  y : mrg xs ys
```

- Tällainen määrittely voidaan esittää kuvana:

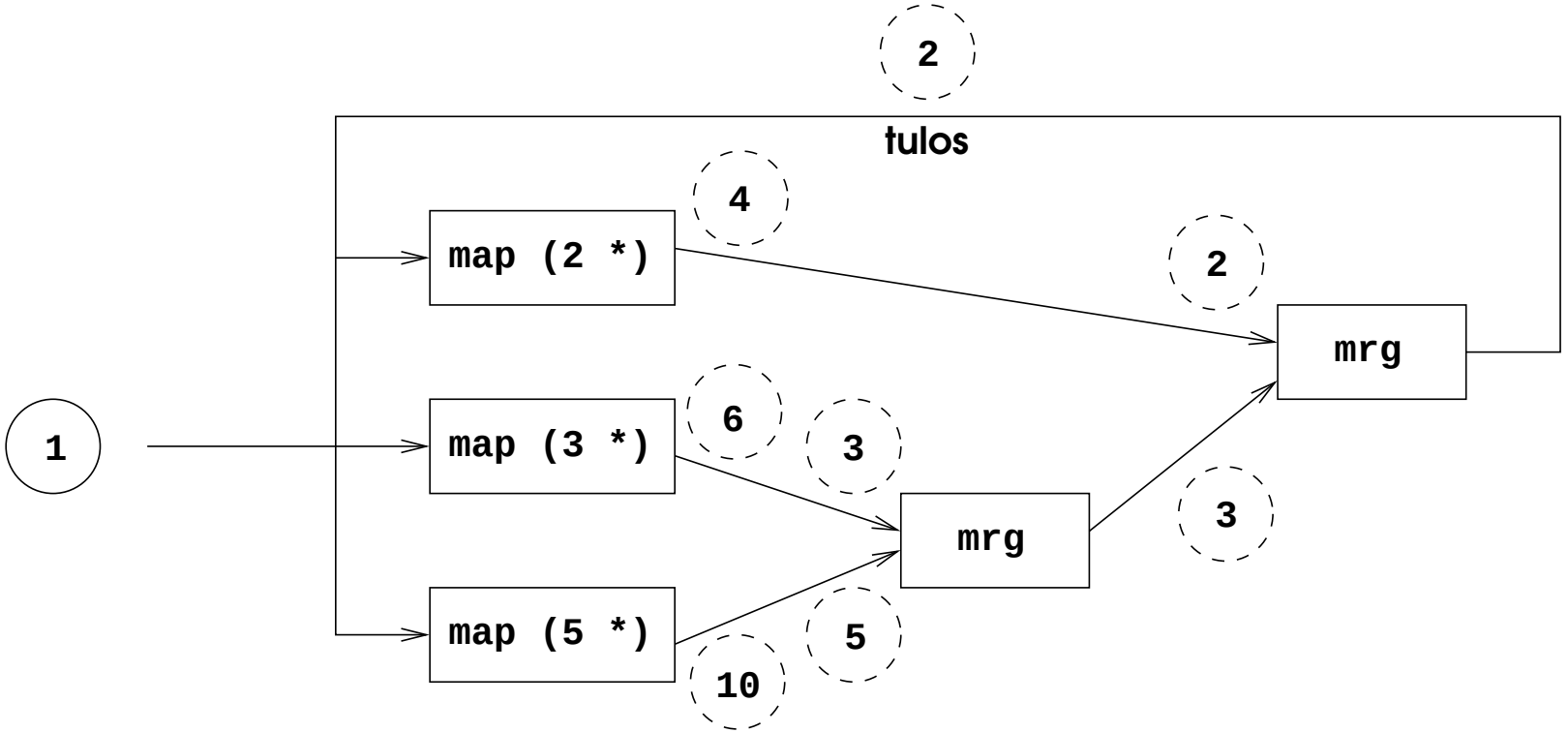
  Laatikoina ovat ne apufunktiot, jotka ottavat sisään listan ja antavat sen perusteella toisen listan — kuin ”tehtaan” koneet.

  Nuolilla yhdistellään yhden laatikon tuloslista toisen laatikon syötelistaksi — kuin tehtaan liukuhihnat.

  Alkiot matkaavat nuolia pitkin laatikosta toiseen, ja jokaisessa laatikossa sisään tulleesta alkiosta tuotetaan uusi alkio — kuin tehtaan tuotteet.

- Silloin itseensä viittaaminen on *kehä*.

- Tehdas saa ulkopuoleltaan jotakin ”raaka-ainetta”, jolla se aloittaa.


2
tulos
4
map (2 *)
2
mrg
1
6
map (3 *)
3
mrg
3
map (5 *)
5
10

3.6 Epäonnistuminen listana onnistumisia

- Joskus ohjelmoidaan *etsintäalgoritmeja*
  - jotka saattavat päätyä *umpikujaan*
  - jolloin täytyy *peruuttaa* (backtrack) ja kokeilla jotakin toista vaihtoehtoa.

- Tavallisesti ne esitetään rekursiivisesti ”2-vaihteisina”:

  Etenemisvaihteella nykyinen osaratkaisu näyttää vielä mahdolliselta, joten pyritään laajentamaan sitä.

  Peruutusvaihteella laajentaminen päätyi umpikujaan, joten palataan edelliseen valintatilanteeseen.

- Laiskat listat tarjoavat toisen esitystavan:
  - Etsintä palauttaakin *onnistuneiden (osa)ratkaisujen listan*.

    (Failure as a list of successes.)
  - Umpikuja on silloin `[]` eikä enää erillinen ”vaihde”.
  - Laiskuuden nojalla muistintarve pysyy pienenä:

    Osaratkaisuja tuotetaan listoihin yksi kerrallaan kysynnän mukaan.

- Tarkastellaan esimerkkinä kontekstittoman kielen jäsentämistä:
  - Jäsennysalgoritmi annetulle kieliopille G on tyyppiä `String -> Bool`:

    "Kuuluuko annettu merkkijono s kieliopin G määrittelemään kieleen?"
  - Lähestymistavassamme se on korkeamman kertaluvun funktio `parses` g jossa g on kielioppia G vastaava *jäsennysfunktio*.
  - Jäsennysfunktion g tyyppi on `String->[String]`:

    $g\ u$ koostuu niistä merkkijonon u loppuosista w joita vastaava alkuosa v (eli $u = v +\!\!+ w$) jäsentyi onnistuneesti.

- Määritellään rivinä 2 lyhennysmerkintä (☞ 4.1) tälle tyypille ”jäsennin”.

- Yksinkertaisin tämän tyyppinen jäsennin on **`token`** v:

  ”lue annettu (vakio)merkkijono v jonon u alusta tai epäonnistu jos se puuttuu”.

- Määritellään pieni *sovellusaluekohtainen ”ohjelmointikieli”* (Domain Specific Language) mutkikkaampien jäsentimien määrittelyyn:

  – p_1 `.-.` p_2 on ”ensin täytyy tulla p_1(:n jäsentämä merkkijono) ja sen perässä p_2”

  – p_1 `.|.` p_2 on ”joko p_1 tai p_2”.

```
module Parser where
type Parser = String -> [String]

parses :: Parser -> String -> Bool
parses =
    ((elem "") .)

token :: String -> Parser
token "" u =
    [u]
token (v:vs) (u:us)
    | v == u =
        token vs us
token _ _ =
    []

infixr 2 .|.
(.|.) :: Parser -> Parser -> Parser
p1 .|. p2 =
    \ s -> (p1 s) ++ (p2 s)

infixr 9 .-.
(.-.) :: Parser -> Parser -> Parser
p1 .-. p2 =
    concat . map p2 . p1
```

- Näillä kahdella operaattorilla (☞ 2.3) sekä Haskellin tarjoamilla itseensä ja toisiinsa viittaavilla määrittelyillä voidaan lausua mikä tahansa kontekstiton kielioppi G sellaisena lausekkeena, jonka arvo on sitä vastaava jäsennin g.

- Esimerkkinä olkoon yksinkertaisten lausekkeiden kielioppi:

$$
\begin{aligned}
E &\rightarrow F + E \\
&\quad | F \\
F &\rightarrow N * F \\
&\quad | (E) \\
&\quad | N \\
N &\rightarrow DN \\
&\quad | D \\
D &\rightarrow 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

```
1   import Parser
2
3   expr =
4       let e = f .-. token "+" .-. e
5               .|. f
6           f = n .-. token "*" .-. f
7               .|. token "(" .-. e .-. token ")"
8               .|. n
9           n = d .-. n
10              .|. d
11          d = token "0" .|. token "1" .|. token "2" .|.
12              token "3" .|. token "4" .|. token "5" .|.
13              token "6" .|. token "7" .|. token "8" .|.
14              token "9"
15      in parses e
```

- Operaattorimme '.-.' ja '.|.' ovat *jäsenninyhdistelijöitä* (parser combinators):

  "laskutoimituksia" jotka yhdistävät kaksi pienempää jäsennintä yhdeksi uudeksi.

- Saimme näin *algebrallisen* lähestymistavan jäsentämiseen:

  – Kielioppi on *itseensä viittaavasti määritelty yhtälöryhmä* jonka *muuttujat eli välikkeet* (esimerkissämme E, F, N ja D) ovat tyyppiä "jäsennin".

  – Teoriassa välikkeen tyyppi on "siitä tuotettavissa olevien merkkijonojen joukko", mutta esitimmekin tämän joukon implisiittisesti jäsentimillämme.

- On valmiskirjasto `Text.ParserCombinators`:
  - Niihin voi liittää attribuuttiarvojen laskentaa: esimerkiksi funktion, joka laskee välikkeestä N tuotetun numerovakion arvon, jne.
  - Niissä on tuki jäsennysvirheiden käsittelylle, jne.
  - Niissä on kontrollioperaattoreita, joilla voi tehostaa jäsennystä karsimalla turhia vaihtoehtoja, jne.

- On myös metakääntäjän `yacc(1)` Haskell-vastine `Happy`:
  - Siihen kannattaa siirtyä jos kielioppi G on mutkikas tai jäsennettävä tiedosto on hyvin pitkä.
  - Esimerkiksi GHC jäsentää lähdekoodit sillä.