1 Johdanto

- Kurssilla tutustutaan perusteisiin

  funktionaalisesta ohjelmoinnista — erityisesti miten se eroaa tutusta tilaperustaisesta ohjelmoinnista

  ohjelmointikielestä Haskell jolla on pakko ohjelmoida funktionaalisesti.

Tietojenkäsittelyteoriasta kiinnostuneelle kurssi esittelee erilaista laskentamallia.

(Yksityiskohdat sivuutetaan.)

Ohjelmoinnista kiinnostuneelle se esittelee erilaista ohjelmointi- ja ohjelmansuunnittelutapaa.

(Sen *omaksuminen* on tietenkin päättymätön taival. . .)

Ohjelmointikielistä ja/tai logiikasta kiinnostuneelle se mainitsee niiden ja logiikan välisiä yhteyksiä.

(Yksityiskohdat sivuutetaan.)

- Kurssilla ei ole varsinaista oppikirjaa. Esimerkiksi seuraavia kirjoja voi käyttää luentokalvojen oheislukemistona:
  - P. Hudak: *The Haskell School of Expression.* Cambridge University Press, 2000.
  - S. Thompson: *Haskell: The Craft of Functional Programming, Second Edition.* Addison-Wesley, 1999.
- Lisäksi Haskell-kielen kotisivulta `http://www.haskell.org` löytyy tutoriaaleja yms.

1.1 Historiaa

- Vuonna 1936 (siis ennen tietokoneita) julkaistiin kaksi alamme perustavaa matemaattista työtä:

Alan Turing julkaisi nimeään kantavat koneet:

- Tavoite: antaa määritelmä intuitiiviselle termille "tehtävä T voidaan ratkaista mekaanisesti".

- Ohjelman suoritus etenee *muuttamalla koneen tilaa käskyjen mukaan* $\Longrightarrow$ tietokoneen abstrakti malli.

Alonzo Church julkaisi λ-*laskennan:*

- Tavoite: antaa määritelmä intuitiiviselle käsitykselle "funktio f on sääntö, jota seuraamalla sen arvo selviää".

- Arvon laskenta etenee *sieventämällä laseketta yksinkertaisemmaksi annetuilla säännöillä* $\Longrightarrow$ ohjelmointikielen abstrakti malli.

- Turingin ja Churchin lähestymistavat osoittautuivat ekvivalenteiksi:

  Turingin koneet ja λ-ohjelmat voidaan kääntää toisikseen.

- Vuonna 1958 John McCarthy keksi (muun muassa) kielen *LISP (List Processing)*.
  - Kielen innoittajana oli λ-laskenta.
  - Alun perin se oli *spesifikaatio*kieli McCarthyn suunnittelemille *tekoäly*ohjelmille.
  - Sitten eräs hänen kollegansa implementoi LISP-sieventäjän tietokoneella — siitä tuli ohjelmointikieli.
  - Lisp kehottaa mutta ei pakota funktionaaliseen ohjelmointiin.

- LISP on yhä käytössä:

  Common Lisp on kielen vakiostandardi, jota käytetään yritysmaailmassa.

  Scheme on kielen kehittyvä murre, joka on akateemisen tutkimuksen ja kehityksen tulos ja väline.

  (Yhtä vanhoista kielistä käytössä on sen lisäksi vain *Fortran (Formula Translator)*, sekin paljon alkuajoistaan muuttuneena.)

- 1970-luvulla tekoälytutkimuksessa kehitettiin teoreemantodistajia, eli ohjelmia jotka keksivät loogisia todistuksia. Niistä haluttiin yleiskäyttöisiä:
  - Käyttäjä syöttää
    * sen väitteen, jolle hän haluaa todistuksen
    * sen logiikan säännöt, jossa hän todistuksensa haluaa.
  - Jokaisessa logiikassa on omat sääntöyhdistelmänsä, joiden on havaittu usein edistävän todistuksen kulkua $\Longrightarrow$ käyttäjien sallittiin kirjoittaa sääntöjen lisäksi myöskin "makroja".
  - Näin teoreemantodistajat alkoivat lähestyä ohjelmointikieliä.

- Tätä kautta funktionaalisessa ohjelmoinnissa päädyttiin *tyyppiteoriaan (Type Theory):*

  – LISP on *piilevästi* tyypitetty kieli:

    Jos suorituksen kuluessa päädytään laskemaan yhteen liukulukua ja merkkijonoa, niin suoritus keskeytyy virheilmoitukseen.

  – Useimmissa ohjelmointikielissä ilmaistaan koodissa *muistipaikkojen tyypit* ja kääntäjä tarkistaa (jos pystyy) ettei suoritusaikana yritetä sijoittaa ”nelikulmaista arvoa pyöreään koloon”.

  – Formaali tyyppiteoria on eräs matemaattisen logiikan haara, johon useat teoreemantodistaja nykyään pohjautuvat.

- Funktionaalisessa ohjelmoinnissa alettiin suunnitella tyypit *päättteleviä* kieliä:
  - Kääntäjä katsoo millaisen tyyppisiä tietoalkioita koodi käsittelee.
  - Ohjelmoijan ei tavitse kirjoittaa (kaikkia) tyyppejä koodiinsa, vaan kääntäjä konstruoi uusia tyyppejä yhdistelemällä vanhoja sopivasti.
  - Näin ohjelmointikielet alkoivat lähestyä teoreemantodistajia:

    kääntäjiin ilmestyi kyseisen tyyppiteorian todistusalgoritmi.

- Tämän kehityskulun tunnetuin edustaja on ohjelmointikieli *Standard ML (Meta Language)*:
  - Ensimmäinen standardi valmistui 1990, nykyinen (viimeinen?) 1997.
  - Standardit esittävät *loogisina sääntökokoelmina* SML-ohjelman

    käännösaikaisen tyyppianalyysin (”Millainen tyyppialgoritmi on?”)

    suoritusaikaisen käyttäytymisen

    eli kieli on spesifioitu paljon täsmällisemmin kuin yleensä!
  - Ranskalaisilla on oma murre *O'Caml*.

- SML-kielellä on ollut suuri vaikutus ohjelmointikielten teoriaan:

  analysoidaan olemassaolevia

  kehitetään uusia

  kieliä tällaisina sääntökokoelmina.

- Silloin voidaan lainata *todistusteorian* (joka on eräs formaalin logiikan keskeinen osa-alue) menetelmiä ohjelmointikielten teorian tutkimukseen.

Suorituksen eteneminen *(progress of execution):* koskaan ei jouduta tilanteeseen, jota ei olisikaan kielen spesifikaatiossa.

"No core dumps!"

Tyyppien säilyminen *(preservation of types):* käännöksen aikana ohjelman käsittelemille tietoalkioille päätellyt tyypit pysyvät samoina myös suorituksen ajan.

"No type casts!"

1.2 Curryn ja Howardin vastaavuus

- ”Matemaattis-loogis-filosofisesti” tyyppiteorian ja ohjelmoinnin suhteen kuvaa *Curryn ja Howardin vastaavuus* (the Curry-Howard isomorphism).

- Vastaavuuden havaitsi jo 1969 loogikko Alan Howard.

- Hän julkaisi sen kuitenkin vasta 1980 hyvin kuuluisan loogikon *Haskell B. Curry*n 80-vuotissyntymäpäivän kunniaksi kootussa juhlakirjassa.

- Alkuperäinen havainto koski erityistä yksinkertaista logiikkaa, mutta sitä on myöhemmin laajennettu muihinkin.

- *Konstruktiivinen* eli *intuitionistinen* logiikka ei hyväksy ns. *kolmannen poissuljetun lakia*.

  – *Klassisessa* logiikassa esimerkiksi väite

  ”Marsissa on tai ei ole elämää”

  on triviaalisti totta tämän lain nojalla — ei ole kolmatta mahdollisuutta.

  – Konstruktiivinen loogikko haluaakin sen sijaan todistuksen jommalle kummalle väitteistä

  ”Marsissa on elämää” tai ”Marsissa ei ole elämää”

  ja se onkin tiukempi vaatimus!

- Tarkastellaan sellaista logiikkaa, jossa on vain implikaatio "jos ϕ niin ψ".

- Sen konstruktiivinen lukutapa on "metodi M, jolla väitteen ϕ todistuksesta saisi muokattua todistuksen väitteelle ψ".

- Howardin havainto:

  - $\phi \Rightarrow \psi$ on konstruktiivisesti todistuva *jos ja vain jos*

  - on λ-termi τ jonka parametri on tyyppiä ϕ ja arvo tyyppiä ψ — eli τ on tyyppiä $\phi \rightarrow \psi$

  - *ja silloin* τ on metodin M ohjelmakoodi.

- Siten tässä yksinkertaisessa tapauksessa

  todistukset ovat ohjelmia (proofs as programs)

  väitteet ovat tyyppejä (propositions as types)

  ohjelman suoritus on todistuksen sieventämistä mutkikkaammasta yksinkertaisemmaksi.

- Tietojenkäsittelijän lukutapa:

  – Tyyppijärjestelmät ovat logiikoita, joilla voidaan ilmaista ja todistaa ohjelmien ominaisuuksia.

  – Ojelmointikielen tyyppijärjestelmän kehittäminen ja tutkiminen on vastaavan logiikan kehittämistä ja tutkimista.

- Kurssilla esitellään funktionaalista ohjelmointikieltä nimeltään *Haskell:*
  - Nykyinen standardi *Haskell 98* valmistui 1998 (ensimmäinen jo 1990).
  - Seuraavaa standardia kehitellään työnimellä *Haskell'*.
  - Tyyppijärjestelmä on pääosin samanlainen kuin kielessä SML.
  - Suoritusmekanismit ovat varsin erilaiset:

    SML sallii myös sellaisen ohjelmointitavan, jossa suoritus etenee muuttamalla koneen tilaa.

    Haskell taas on *puhdas (pure)* funktionaalinen kieli: suoritus on sievennystä, käsitettä ”tila” ei ole.

- SML ja Haskell pohjautuvat *Hindleyn ja Milnerin ns. let-polymorfiseen* tyyppiteoriaan (erilaisin laajennuksin).

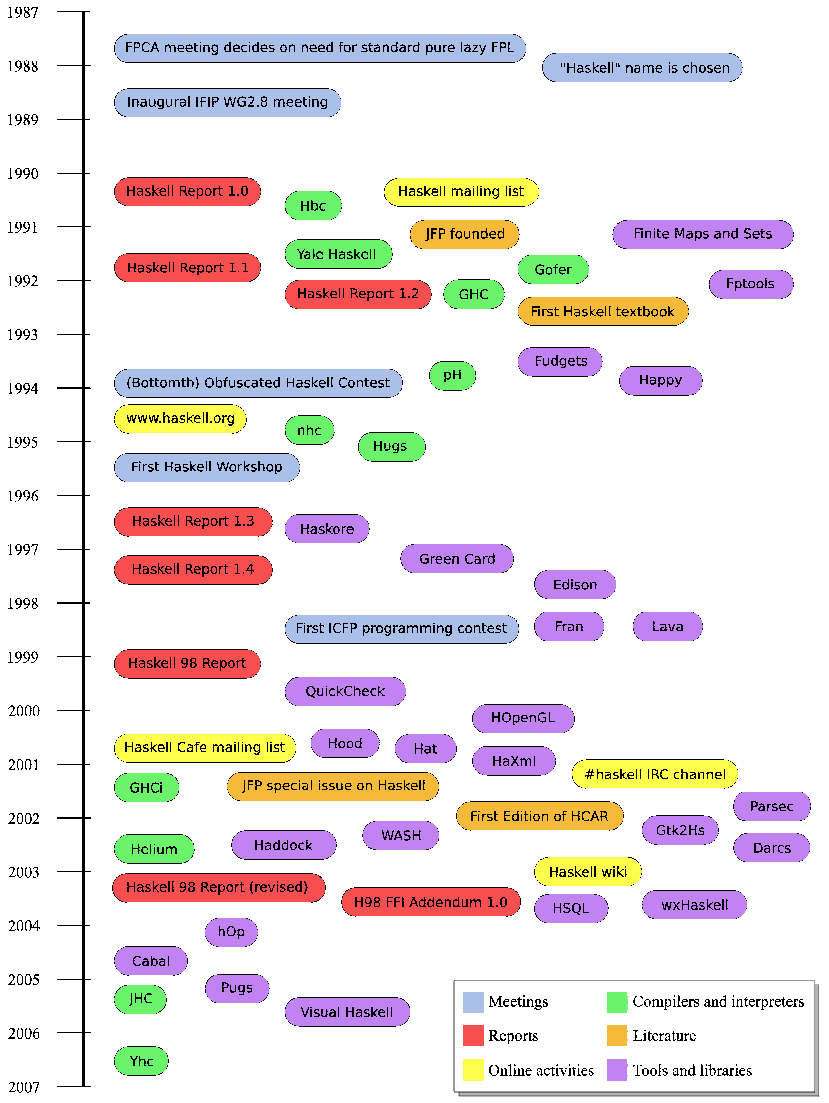
- Curryn-Howardin vastaavuuden näkökulmasta

  implikaation '$\rightarrow$' vastine on *funktio* kuten edellä

  konjunktion '$\wedge$' vastine on *monikko* (☞ 2.1.4)

  konstruktiivisen disjunktion '$\vee$' vastine on tyyppi, jonka arvot ovat "joko tällaisia tai tuollaisia" (☞ 4.2)

  universaalikvantifioinnin '$\forall$' vastine on *geneerisyys:* sama funktio soveltuu monille eri tyypeille (☞ 4.4).


1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
FPCA meeting decides on need for standard pure lazy FPL
"Haskell" name is chosen
Inaugural IFIP WG2.8 meeting
Haskell Report 1.0
Hbc
Haskell mailing list
JFP founded
Finite Maps and Sets
Yale Haskell
Haskell Report 1.1
Gofer
Fptools
Haskell Report 1.2
GHC
First Haskell textbook
Fudgets
pH
(Bottomth) Obfuscated Haskell Contest
Happy
www.haskell.org
nhc
Hugs
First Haskell Workshop
Haskell Report 1.3
Haskore
Green Card
Haskell Report 1.4
Edison
First ICFP programming contest
Fran
Lava
Haskell 98 Report
QuickCheck
HOpenGL
Haskell Cafe mailing list
Hood
Hat
HaXml
#haskell IRC channel
GHCi
JFP special issue on Haskell
Parsec
First Edition of HCAR
Gtk2Hs
Hclium
Haddock
WASH
Darcs
Haskell wiki
Haskell 98 Report (revised)
H98 FFI Addendum 1.0
HSQL
wxHaskell
hOp
Cabal
Pugs
JHC
Visual Haskell
Yhc
Meetings
Compilers and interpreters
Reports
Literature
Online activities
Tools and libraries

1.3 Tilaton ohjelmointi

- Eräs muotoilu *Leibnizin laille* on: "Ilmaukset α ja β tarkoittavat samaa

  jos ja vain jos

  kaikilla ilmauksilla γ pätee seuraava: jos vaihdamme ilmauksen γ sisällä ilmauksen α esiintymän ilmauksen β esiintymäksi, niin näin saadun ilmauksen δ merkitys on yhä sama kuin alkuperäisen ilmauksen γ."

- Otetaan ohjelmointikielen suunnittelun ohjenuoraksi, että Leibnizin lain soveltaminen sen ohjelmiin olisi mahdollisimman yksinkertaista.

- Kun halutaan kirjoittaa "kerralla oikein toimivia" ohjelmia, niin eräs keskeinen työväline on *korvata samaa tarkoittavia ohjelmanpätkiä toisillaan*.

- Esimerkiksi potenssiinkorotusfunktio x^n voidaan määritellä induktiolla:

$$\begin{aligned} \mathrm{pow}(x, 0) &= 1 \\ \mathrm{pow}(x, n) &= x \cdot \mathrm{pow}(x, n-1) \quad \text{kun } n > 0. \end{aligned}$$

  (Oletetaan että potenssi $n \in \mathbb{N}$.)

- Ryhdytään nopeuttamaan sitä jakamalla tapaus $n > 0$ osiin:

  $\mathrm{pow}(x, n) = x \cdot \mathrm{pow}(x, n-1)$ kun $n > 0$ on pariton
  $\mathrm{pow}(x, n) = \ldots$ kun $n > 0$ on parillinen.

- Jäljellä on siis tapaus $n = 2 \cdot m$ missä $m > 0$:

$$\begin{aligned} \mathrm{pow}(x, 2 \cdot m) &= x^{2 \cdot m} \\ &= (x^2)^m \\ &= x^2 \cdot (x^2)^{m-1} \end{aligned}$$

  josta induktiolla suureen m suhteen

$$= x^2 \cdot \mathrm{pow}(x^2, m-1)$$

  joka on alkuperäinen sääntö *takaperin* kun $m > 0$:

$$= \mathrm{pow}(x^2, m).$$

- Näin saimme suoralla laskulla nopeamman funktion

  $$\begin{aligned}
  \mathrm{pow}(x, 0) &= 1 \\
  \mathrm{pow}(x, n) &= x \cdot \mathrm{pow}(x, n-1) \quad \text{kun } n > 0 \text{ on pariton} \\
  \mathrm{pow}(x, n) &= \mathrm{pow}(x \cdot x, n/2) \quad \text{kun } n > 0 \text{ on parillinen}
  \end{aligned}$$

  joka toimii varmasti oikein jo konstruktionsa nojalla.

- Jokainen $=$-päättelyaskel oli Leibnizin lain sovellus!

- Oikeiden ohjelmien kehittämistä siis helpottaisi, jos ohjelmointikieli olisi suunniteltu niin, että Leibnizin lain soveltaminen sillä kirjoitettuihin ohjelmiin olisi helppoa.

- Leibnizin laki ohjelmoinnissa: "Ohjelmakoodinpätkät α ja β tarkoittavat samaa

  jos ja vain jos

  kokonaiset ohjelmat γ ja δ *käyttäytyvät ajoaikana täsmälleen samoin*."

  - Pätkä α tarkoittaa *koneen tilan muutosta* sitä edeltävästä tilasta p sen jälkeiseen tilaan q.
  - Pätkä β tarkoittaa samaa jos ja vain jos β kuvaa jokaisen tilan p samalle tilalle q kuin α.
  - Mutta tiloja p on vaikea nähdä koodista γ!
  - Silmukoissa jopa seuraava p_i riippuu edellisestä q_{i-1}.

- Voidaanko ohjelmanpätkässä

```
a[j] = a[i] + 1;
if (a[j] == a[i])
   K
else
   E
```

  tuo `if`-ehto korvata arvolla `false`?

- Eli voidaanko koko `if`-lause optimoida pelkäksi `else`-haarakseen E?

- Eli voidaanko muistiviittauksen `a[j]` tilalle kirjoittaa arvo `a[i] + 1`?

- Tilaperustaisessa ohjelmointikäsityksessä on siis mutkikasta osoittaa ohjelma oikeaksi (mm.) koska päättelyn askeleet ovat hankalia.

- Ratkaisuna ovat *invariantit:*
  - Väitteet ohjelman tiloista p ja q ennen ja jälkeen koodinpätkän α suorituksen.
  - Esimerkkejä oli kurssilla Tietorakenteet.

- Tällä kurssilla lähdetäänkin toiseen suuntaan:
  - Millaista on ohjelmoida ilman tiloja?
  - Millainen ohjelmointikieli tukee sitä?

1.3.1 Viittausten läpikuultavuus

- Toinen tapa ajatella Leibnizin lakia (☞ 1.3) ohjelmoinnissa on *viittausten läpikuultavuus* (referential transparency):
  - Jos kahdella eri lausekkeella α ja β on sama arvo v, niin niitä itseäänkin pidettäköön keskenään samoina.
  - Kääntäen, kahden arvon v ja w samuus pitää voida selvittää vain "katsomalla" niitä itseään.
  - Silloinhan α ja β voidaan vaihtaa aina keskenään, nehän ovat eri viittaustapoja samaan otukseen v.
  - Siis "se todellinen" v "kuultaa" viittausten α ja β alta.

- Seuraava esimerkkiohjelma osoittaa, ettei Java täytä tätä vaatimusta:
  - Riveillä 6 ja 7 saadaan merkkijono `bar` lisäämällä merkkijonon `foo` perään tyhjä merkkijono.
  - Viittausten läpikuultavuus vaatisi, ettei lausekkeita `foo` ja `bar` voisi erottaa toisistaan:

    Rivien 9–12 tulostuksissa ei ole mitään "silmin havaittavaa" eroa.
  - Silti rivillä 15 vertailun tulos onkin `FALSE`!
  - Siis vaikka koodissa ei sitä näy, niin rivillä 15 erotellaankin toisistaan `foo` ja `bar` sillä perusteella, että edellinen syntyi rivillä 6 ja jälkimmäinen rivillä 7.

```
 1   import java.io.*;
 2   class RefOpaq{
 3
 4       public static void main(String args[]){
 5
 6           String foo = "aina sama juttu";
 7           String bar = foo + "";
 8
 9           System.out.format("foo=\"%s\"\n",
10                             foo);
11           System.out.format("bar=\"%s\"\n",
12                             bar);
13
14           System.out.format("foo==bar? %B\n",
15                             foo==bar);
16           System.out.format("foo equals bar? %B\n",
17                             foo.equals(bar));
18       }
19   }
```

- Javassa on siis erilliset *samuuden* ja *yhtäsuuruuden* käsitteet.

- Ohjelmoijalle viittausten läpikuultavuus on sitä, että
  - kielessä on vain *yksi samuuden eli yhtäsuuruuden käsite*
  - ja se on *sisällöllinen* samuus kuten rivin 17 `equals`

    eikä ohjelman suorituksesta seuraava samuus kuten rivin 15 ”==”.

1.3.2 Matemaattiset muuttujat

- Tärkein yksittäinen askel tilaperustaisesta tilattomaan ohjelmointiin on, että

  1. muuttujaa esiteltäessä sille annetaan alkuarvo

  2. jota *ei enää muuteta* ohjelman edetessä!

- Muuttujaa ajatellaan

  ohjelmoinnissa muistipaikan

  matematiikassa arvon

  nimenä.

- Ohjelmoinnissahan on kahdenlaisia koneen tilan muutoksia:
  - siirrytään suorittamaan ohjelman seuraavaa riviä — sen vastine säilyy myös funktionaalisessa ohjelmoinnissa
  - muokataan työmuistin sisältöä — se jääkin nyt pois.

- Niinpä (puhtaassa) funktionaalisessa ohjelmoinnissa ja Haskell-kielessä *ei ole sijoituslausetta*

  vaan pelkästään *nimentöjä*

  $$\text{nimi} = \text{arvo}$$

  jonka tulkinta on "tämä nimi on lyhennysmerkintä tuolle arvolle".

1.3.3 Matemaattiset funktiot

- Eräs seuraus matemaattisista muuttujista (☞ 1.3.2) on että myös *funktiotkin ovat matemaattisia:*
  - Saman funktion f kaikki eri kutsut $f(v)$ samalla parametrin arvolla v antavat aina *saman* tuloksen: mikään muuttuja josta f riippuu ei ole voinut vaihtaa arvoaan kutsujen välillä.
  - Funktiolla f ei myöskään voi olla minkäänlaisia *sivuvaikutuksia:* se voi vain palauttaa parametriarvonsa v määräämän tuloksen.
- Funktion f koko rajapinta ulkomaailmaan näkyy siis sen omassa koodissa.

1.3.4 Ongelmallinen I/O

- Kun kielessä on vain matemaattiset funktiot, niin *syötteen lukeminen ja tulosteen kirjoittaminen* täytyy tehdä toisin kuin normaalisti.

- Esimerkiksi Java-kielen merkinlukumetodi `int read()` ei käy:
  - Sen pitäisi palauttaa *sama* merkki jokaisella kutsukerralla!
  - Se onkin *aliohjelma* (subroutine) eikä funktio!

- Ongelma onkin perustavaa laatua:
  - ympäröivällä maailmalla *on tila* (avointen syötetiedostojen nykyiset lukukohdat, yms.)...
  - jota juuri pyrimme eliminoimaan ohjelmointikielestämme!

- Tilallisen ulkomaailman kanssa kommunikointia esiintyy myös esimerkiksi
  - käyttöliittymissä,
  - tietokantarajapinnoissa,...

- Haskell 98 -standardin vastaus on *monadinen I/O* (☞ 5):
  - Standardoitiin tietty ohjelmointikuri, jossa syötettä voi lukea ja tulosta kirjoittaa vain tietyssä osassa ohjelmaa.
  - Siinä (ja vain siinä) osassa ohjelmaa saa muokata ulkomaailman tilaa.
  - Tämän osan tyyppi on tietty ”monadi”.
  - Ohjelmointikielen tyyppitarkastaja voi silloin valvoa, että kuria noudatetaan.
- Monadit soveltuvat muuhunkin kuin tilan käsittelyyn.

1.3.5 Rekursio

- Ilman sijoituslausetta (☞ 1.3.2) *toistorakenteet* kuten `for` ja `while` tulevat mahdottomiksi.

- Funktionaalisessa ohjelmoinnissa ja Haskell-kielessä toisto ilmaistaankin aina *rekursiolla:*

  – Rekursiokutsussa parametri(e)n nimet korvataan kutsuttavassa lausekkeessa kutsuhetken arvo(i)llaan ja jatketaan.

  – Se korvaa tarpeen päivittää silmukkamuuttujia.

- Esimerkkinä nopeutetulla potenssiinkorotusalgoritmillamme (☞ 1.3)

$$
\begin{aligned}
\boxed{\mathrm{pow}(3,5)} &\Longrightarrow 3 \cdot \mathrm{pow}(3, \boxed{5-1}) \\
&\Longrightarrow 3 \cdot \boxed{\mathrm{pow}(3,4)} \\
&\Longrightarrow 3 \cdot \mathrm{pow}(3 \cdot 3, \boxed{4/2}) \\
&\Longrightarrow 3 \cdot \boxed{\mathrm{pow}(3 \cdot 3, 2)} \\
&\Longrightarrow 3 \cdot \mathrm{pow}(3 \cdot 3, \boxed{2/2}) \\
&\Longrightarrow 3 \cdot \boxed{\mathrm{pow}((3 \cdot 3) \cdot (3 \cdot 3), 1)} \\
&\Longrightarrow 3 \cdot ((3 \cdot 3) \cdot (3 \cdot 3)) \cdot \mathrm{pow}((3 \cdot 3) \cdot (3 \cdot 3), \boxed{1-1}) \\
&\Longrightarrow 3 \cdot ((3 \cdot 3) \cdot (3 \cdot 3)) \cdot \boxed{\mathrm{pow}((3 \cdot 3) \cdot (3 \cdot 3), 0)} \\
&\Longrightarrow 3 \cdot ((3 \cdot 3) \cdot (3 \cdot 3)) \cdot 1 \\
&\;\;\vdots \\
&\Longrightarrow 243
\end{aligned}
$$

missä ” $\Longrightarrow$ ” on ”korvaa $\boxed{\text{laatikoitu}}$ lauseke arvollaan”.

- Tämän funktionaalisen ohjelman merkitys ja suoritus ovat siis intuitiivisesti:
  - Ohjelma koostuu säännöistä muotoa

    ”Jos rekursiokutsu on tämän muotoinen = niin se voidaan korvata tuon muotoisella lausekkeella

    (kun mahdolliset lisäehdotkin ovat totta).”
  - Arvon laskenta etenee liikuttelemalla korvaus- eli sievennyskohtaa muokkautuvassa lausekkeessa.

- *Deklaratiivisen eli kuvailevan* ohjelmoinnin idea:

  ohjelmoija ilmoittaa *mitä* hän haluaa laskettavan, ei yksityiskohtia siitä *miten* sen pitäisi tapahtua.

- Siis Leibnizin laki (☞ 1.3) on "mekanisoitu" ohjelmointimalliksi valitsemalla jokin tietty sievennysjärjestys.

1.3.6 Vapaampi suoritusjärjestys

- Potenssiinkorotusesimerkissä (☞ 1.3.5) sievensimme aina eksponenttia n, koska sen perusteella valittiin käytettävä sääntö.

- Sen sijaan jätimme kantaluvun x neliöinnit $x \cdot x$ odottamaan, kunnes niitä tarvittiin lopputulokseen 243.

- Tilaperustainen kieli ei voi viivästää suoritusta: aliohjelmakutsun kaikille parametreille pitää ensin laskea arvot, koska laskennan sivuvaikutukset voivat vaikuttaa myös kutsuttavaan aliohjelmaan.

- Haskell-kieli käyttää juuri tätä *laiskaa suoritusta* (lazy/non-strict evaluation) (☞ 2.2.2):
  - Sääntöä kokeiltaessa parametreja sievennetään vain sen verran, että nähdään soveltuuko tämä sääntö vaiko ei.
  - Käytännössä nämä sievennyksen tulokset *muistetaan* jotta niitä ei tarvitse tehdä uudelleen (☞ 7.1).

- Tarkastellaan esimerkkinä *tuottaja-kuluttaja-parien (producer-consumer pairs)* ohjelmointia:
  - Olkoon A aliohjelma, joka *tuottaa* erilaisia tietueita t.
  - Olkoon B aliohjelma, joka *kuluttaa* tällaisia tietoalkioita t.
  - Esimerkiksi ohjelmoitaessa lautapeliä:
    - A tuottaa kaikki mahdolliset siirtovaihtoehdot — pohjautuen pelin sääntöihin.
    - B arvioi eri vaihtoehtojen hyvyyttä — pohjautuen erilaisiin heuristiikkoihin.
  - Olisi kehnoa ohjelmistosuunnittelua yhdistää A ja B samaksi aliohjelmaksi.

Tilaperustaisessa ohjelmoinnissa synkronoidaan käsin:

$t_{\text{best}} = A$:n ensimmäinen vaihtoehto;
$c_{\text{best}} = B(t_{\text{best}})$;
while A:lla on lisää vaihtoehtoja **do**
  $t_{\text{next}} = A$:n seuraava vaihtoehto;
  $c_{\text{next}} = B(t_{\text{next}})$;
  if $c_{\text{next}} < c_{\text{best}}$ **then**
    $t_{\text{best}} = t_{\text{next}}; c_{\text{best}} = c_{\text{next}}$
  end if
end while

Funktionaalisessa ohjelmoinnissa annetaan määrittelyt

$x = A$:n *kaikkien* vaihtoehtojen lista;
$y =$ se lista joka saadaan kun listan x jokaiseen alkioon sovelletaan kuvausta $f(t) = (t, B(t))$;
pari $(t_{\text{best}}, c_{\text{best}}) =$ se listan y alkioista jonka jälkimmäinen komponentti on pienin

Laiskassa suorituksessa menetelmät ovat oleellisesti samat:

1. Halutaan laskea pari $(t_{\text{best}}, c_{\text{best}})$.
2. Sen määrittely sanoo, että lista y on käytävä läpi alkio alkiolta.
3. Listan y seuraava alkio saadaan soveltamalla kuvausta f listan x seuraavaan alkioon.
4. Listan x seuraava alkio saadaan kysymällä A:n seuraava vaihtoehto.

Hyödyt:

- A voidaan ohjelmoida funktioksi, joka palauttaa yhdellä kertaa laiskan listan (☞ 3) kaikista tulosvaihtoehdoistaan — alleviivattuja apuoperaatioita ei tarvitse ohjelmoida itse.

- Apumääritelmien x ja y oikeellisuus on helppo nähdä.

1.3.7 Korkeamman kertaluvun funktiot

- Tuottaja-kuluttaja-esimerkissä (☞ 1.3.6) määrittelimme listan y operaatiolla jonka parametrit olivat
  - lista x
  - ja *funktio* f.
- Tämä operaatio voidaan ilmaista yleisenä sääntöinä

$$\begin{aligned} \mathsf{map}(f, [\,]) &= [\,] \\ \mathsf{map}(f, z : zs) &= f(z) : \mathsf{map}(f, zs) \end{aligned}$$

  missä [] on tyhjä lista ja $z : zs$ on epätyhjä lista, jonka ensimmäinen alkio on z ja loppulista zs (☞ 3).

- Tällä apufunktiolla voidaan esimerkiksi ilmaista

$$y = \text{map}(f, A\text{:n kaikkien vaihtoehtojen lista})$$
$$f(t) = (t, B(t)).$$

- Lopputulos voidaan puolestaan ilmaista

$$(t_{\text{best}}, c_{\text{best}}) = \text{foldl}(g, (+\infty, +\infty), y)$$

missä apufunktio g palauttaa saamistaan kahdesta parista sen, jolla on pienempi jälkimmäinen komponentti:

$$g((t_1, c_1), (t_2, c_2)) = (t_1, c_1) \quad \text{kun } c_1 \leq c_2$$
$$g((t_1, c_1), (t_2, c_2)) = (t_2, c_2) \quad \text{muuten.}$$

- Apufunktio foldl(h, a, l) ilmaisee hyvin tavallisen tavan käydä läpi lista l alusta loppuun:
  - Annetaan "laskurille" a alkuarvo.
  - Jokaisen alkion z kohdalla päivitetään sen seuraavaksi arvoksi $h(a, z)$.
  - Lopuksi palautetaan näin kertynyt lopullinen arvo.

- Idea kirjoitetaan suoraan säännöiksi:

$$\text{foldl}(h, a, [\,]) = a$$
$$\text{foldl}(h, a, z : zs) = \text{foldl}(h, h(a, z), zs)$$

- Nämä apufunktiot map ja foldl ovat hyvin monikäyttöisiä, koska niitä voi muokata eri käyttötarpeisiin sopivilla f ja h.

- Esimerkiksi operaatio "laske annetun lukulistan l lukujen neliöiden summa" voidaan ilmaista

$$\begin{aligned} \text{sos}(l) &= \text{foldl}(\text{sum}, 0, \text{map}(\text{square}, l)) \\ \text{sum}(x, y) &= x + y \\ \text{square}(z) &= z \cdot z. \end{aligned}$$

- Ne ovat esimerkkejä *korkeamman kertaluvun* (higher-order) funktioista, eli sellaisista jotka saavat parametreikseen toisia funktioita.

- Funktionaalisessa ohjelmoinnissa käytetään paljon korkeamman kertaluvun funktioita:
  - määritellään jokin yleinen laskentaperiaate (kuten foldl) vain kerran
  - käytetään samaa määritelmää eri tilanteissa sopivin parametrifunktioin (kuten h).
- Tavoitteena on säästää paitsi kirjoitus- myös ajatustyötä:
  - Periaatteen määritelmän oikeellisuus järkeillään vain kerran — sitä kirjoitettaessa.
  - Sitä käytettäessä ei järkeilyä tarvitse toistaa — parametrit eivät pysty sotkemaan itse periaatetta.

1.3.8 Funktiot arvoina

- Funktionaalisessa ohjelmoinnissa myös *funktiot ovat täysivaltaisia arvoja* kuten muutkin tietoalkiot.

- Tämä tarkoittaa, että myös funktiolla voi tehdä rajoituksetta kaikkia niille luontaisia toimituksia:
  - Niiden palautusarvoja voi laskea.
  - Niitä voi antaa toisilleen parametreina (☞ 1.3.7).
  - Niitä voi myös saada toisiltaan *palautusarvoina*.
  - Niitä voi *yhdistellä* korkeamman kertaluvun funktioilla (☞ 1.3.7).

- Voimme esimerkiksi määritellä funktion

$$\text{foo}(x) = y$$

missä apufunktion

$$y(z) = x \cdot z + 1$$

muuttuja x viittaa pääfunktion foo parametriin.

- Silloin kutsu foo(2) palauttaa arvonaan *funktion*

$$y_2(z) = 2 \cdot z + 1$$

- . . . ja toinen kutsu foo(3) *toisen* funktion

$$y_3(z) = 3 \cdot z + 1.$$

1.3.9 Automaattinen muistinhallinta

- Java on tuonut automaattisen muistinhallinnan (= roskankeruun yms.) ohjelmoinnin valtavirtaan.

- Tämä on melko uutta: aiemmissa valtakielissä kuten C ja C++ *dynaaminen* muisti oli ohjelmoijan vastuulla.

- Voidaan sanoa, että juuri funktionaalinen ohjelmointi osoitti automaattisen muistinhallinnan hyödyllisyyden.
  - Javan suunnittelijoilla oli syvällinen LISP-tausta.
  - ”Funktiot arvoina” -ajattelu (☞ 1.3.8) vaatii käytännössä automaattisen muistinhallinnan.

1.3.10 Lausekkeita komentojen sijaan

- Edellä olemme puhuneet aina *lausekkeista* (expression) ja niiden *arvoista* (value).

- Tilaperustaisissa ohjelmointikielissä on niiden lisäksi myös *lauseita* (statement), esimerkiksi Javassa

  expression;

  – ohjelma koostuu pohjimmiltaan tällaisista lauseista

  – jotka ovat *komentoja* siirtyä tilasta toiseen

  – joiden joukossa on *pääohjelma* — se, jonka komennot käyttäjä haluaa suoritettavan.

- Funktionaalinen ohjelma taas koostuu pohjimmiltaan nimennöistä (☞ 1.3.2)

  $$\text{nimi} = \text{lauseke}$$

  sekä *päälausekkeesta* — siitä, jonka arvon käyttäjä haluaa tietää.

- Funktionaaliset kielet soveltuvatkin hyvin *interaktiiviseen* käyttöön:

  – Kielen toteutus onkin *tulkki* eikä kääntäjä.

  – Käyttäjä voi kirjoittaa nimentöjä suoraan tulkin komentoriville (tai ladata niitä tiedostoista).

  – Käyttäjä voi kirjoittaa myös lausekkeita, joiden arvot tulkki laskee välittömästi nykyisin nimennöin.

1.4 GHC

- Kurssilla käytetään Haskell-toteutusta *the Glasgow Haskell Compiler (GHC)*.
  - Laitoksen Linux-koneisiin on asennettu tulkki `ghci` (GHC interactive).
  - Pakettiin kuuluu myös mm. perinteinen kääntäjä `ghc` sellaisia ohjelmia varten, joita halutaan ajaa ilman tulkkia.
  - GHC toteuttaa nykyisen standardin Haskell 98.
  - GHC sisältää myös hyvin monia seuraavaan standardiin Haskell' ehdotetuista laajennuksista.

- Paketin (= `ghci`, `ghc`, dokumentaatiot, kirjastot,. . .) voi hakea (ilmaiseksi) kotikoneelle kielen verkkosivulta `http://www.haskell.org`.

2 Haskell-kielen ydin

- Aloitetaan Haskell-kieleen tutustuminen sen peruslausekkeista ja -tyypeistä.

- Myöhemmin sitä rikastetaan lisäpiirteillä:
  - käyttäjän määrittelemillä uusilla tyypeillä (☞ 4.3)
  - tyyppien välisillä sukulaisuussuhteilla (☞ 4.6)
  - syötteellä ja tulostuksella (☞ 5)
  - moduuleilla (☞ 6).

- Ensimmäinen Haskell-esimerkkiohjelmamme olkoon nopeutettu potenssiinkorotus (☞ 1.3):

```
1   pow :: (Integer,Integer) -> Integer
2   pow (_,0) =
3       1
4   pow (x,n)
5       | odd n =
6           x * pow (x,n-1)
7       | even n =
8           pow (x * x,div n 2)
```

(Rivinumerot eivät kuulu kieleen, ne ovat koodilistauksissamme havainnollisuuden vuoksi.)

- Lähdetään selvittämään sitä osa osalta.

2.1 Muuttujan esittely ja tyypitys

- Esimerkkiohjelmamme (☞ 2) rivinä 1 on

  muuttujan nimi :: *sen tyyppi*.

- Muuttujien nimet alkavat pienellä kirjaimella...
  - ... ja jatkuvat pienillä tai suurilla kirjaimilla, numeroilla ja *heittomerkillä* '.
  - Sivuutetaan erikoissäännöt, joiden nojalla myös operaattorit kuten '+' ovat muuttujien nimiä.

- Tyyppien nimet alkavat SUURELLA kirjaimella.

- Haskell on *vahvasti tyypitetty* kieli, eli jokaisella sen
  - arvolla
  - muuttujalla joka nimeää arvon

  on kiinteä tyyppi.

- Haskell päättelee tyypit itse, joten tätä riviä ei tarvita.

- Ne kuitenkin antavat informaatiota

  ohjelmoijalle joka tarvitsee tietoa funktion parametrien ja tuloksen tyypeistä.

  kääntäjälle joka ilmoittaa virheen, jos sen päättelemä ja ohjelmoijan kirjoittama tyyppi poikkeavat.

- Esimerkkirivimme 1 ilmoittaa, että muuttujan nimeltä `pow` arvo on *funktio*, koska tyyppi on muotoa

  parametrin tyyppi -> tuloksen tyyppi.

  missä

  tulos on kokonaislukutyyppiä `Integer` (☞ 2.1.2).

  parametri on tyyppiä "*pari* (pair) jonka molemmat osat ovat vuorostaan tyyppiä `Integer`" (☞ 2.1.4).

- Funktiollamme `pow` ei olekaan 2 parametria vaan *1 parametri jossa onkin 2 osaa*.

  Myöhemmin palataan siihen, mitä *oikeasti* moniparametrinen funktio on (☞ 2.2.6).

2.1.1 Hahmot

- *Hahmot* (patterns) ovat keskeinen ohjelmointiväline moderneissa funktionaalisissa kielissä.

- Perusidea on *sovittaa parametrina saatua dataa hahmoon*, ja jos se sopii, niin silloin ohjelma etenee hahmon mukaiseen haaraan. . .

- . . . eli hahmoa vastaavan merkin '=' jälkeiseen lausekkeeseen.

- Tämä *hahmonsovitus* (pattern matching) on formaali vastine aikaisemmalle intuitiollemme ”korvaa tämän muotoinen rekursiokutsu tuon muotoisella tuloksella” (☞ 1.3.5).

- Käydään tässä ensin läpi hahmojen yleispiirteet, ja täydennetään niitä tyyppikohtaisilla lisäpiirteillä eri tyyppejä tarkastellessamme.

- Esimerkkiohjelmamme (☞ 2) hahmot ovat

  rivin 2 `(_,0)` joka sopii kaikkiin sellaisiin pareihin, joiden jälkimmäisenä osana on kokonaisluku 0.

  – Tämä on lyhyt ja deklaratiivinen tapa ilmaista millaisille datoille tämä haara on tarkoitettu.

  – Hahmona alaviiva '_' tarkoittaa "en nyt välitä mitä dataa tässä kohdassa on".

  rivin 4 `(x,n)` joka sopii kaikkiin pareihin.

  – Samalla se *nimeää* parin ensimmäisen osan nimellä `x` ja toisen `n`.

  – Näitä osille annettuja nimiä voi sitten käyttää tämän hahmon haarassa eli riveillä 5–8.

- Hahmonsovitus on puhtaan *syntaktinen* toimitus:
  - Sama muuttuja saa esiintyä hahmossa vain kerran.

    Siis hahmo `(y,y)` on laiton: ehto ”ne parit, joiden molemmt osat ovat samat” on ilmaistava muulla tavalla.
  - Alaviiva voi silti toistua: `(_,5,_)` on laillinen hahmo niille kolmikoille, joiden keskimmäinen komponentti on kokonaisluku 5.

- Tällaisten *semanttisten* lisäehtojen ilmaisemiseen on Haskell-kielessä *hahmovahdit* (pattern guards)

  hahmo | *sen vahti* = *sen haara*

  joka on totuusarvotyyppinen lauseke.

- Hahmovahdin merkkinä olevan pystyviivan '|' voi siis lukea "kun" (☞ 1.3).

- Hahmon `(y,y)` voi siis kirjoittaa laillisesti

```
(y,y') | y == y'
```

  missä "==" on Haskell-kielen samuusvertailu. (☞ 1.3.1).

- Esimerkkiohjelmamme vahdit ovat

  rivin 5 `odd n`

  rivin 7 `even n`

  jotka tutkivat onko kokonaisluku n pariton tai parillinen.

- Nämä apufunktiot `odd` ja `even` ovat tavallisia Haskell-funktioita

  – jotka kuuluvat osana vakiokirjastoon `Prelude`

  – joka ladataan automaattisesti tulkin käynnistyessä.

- Molemmat esimerkkiohjelmamme vahdit liittyvät *samaan* rivin 4 hahmoon `(x,n)`.
  - Silloin rivi 5 tarkoittaa "käytä riviä 6 silloin

    kun data sopi rivin 4 hahmoon

    ja lisäksi sen toinen osa `n` on pariton"
  - Vastaavasti rivi 7 tarkoittaa "käytä riviä 8 silloin

    kun data sopi rivin 4 hahmoon

    ja lisäksi sen toinen osa `n` on parillinen"

- Haskell-kielessä sääntöjä kokeillaan niiden *kirjoitusjärjestyksessä*.

- Intuitiivisesti ohjelmamme koostuu siis 3 säännöstä:

  rivit 2–3 käsittelee tapauksen "korota potenssiin 0"

  rivit 4–6 tapauksen "korota parittomaan potenssiin $n \neq 0$"

  rivit 4,7–8 tapauksen "korota parilliseen potenssiin $n \neq 0$".

- Joskus haluamme nimetä samalla kertaa hahmoon sopivasta datasta sekä osia että koko datan. Silloin käytetään `@`-*hahmoa:*

```
p@(10,_,q)
```

  - sopii kaikkiin kolmikoihin, joiden ensimmäinen osa on kokonaisluku 10
  - nimeää sopivan kolmikon kolmannen osan nimellä `q`
  - nimeää koko sopivan kolmikon nimellä `p`.

- Haskell-kielessä on myös *viivästetty* hahmo:

  1. esimerkiksi hahmo `~(100,b)` sopii kaikkiin pareihin, joiden ensimmäinen osa on kokonaisluku 100

  2. mutta se tarkastetaan *vasta* kun sen toista osaa `b` tarvitaan

  3. jolloin väärä luku (kuten 101) johtaakin *ajonaikaiseen virheeseen*.

- Sitä tarvitaan harvoin, yleensä vain joissakin erityistilanteissa (☞ 7.2).

2.1.2 Numeeriset tyypit

Konekohtainen kokonaislukutyyppi `Int`, jolla on *äärellinen toteutuskohtainen* arvoalue, esimerkiksi GHC:ssa 32 bittiä eli

$$-2147483648 \ldots 2147483647.$$

- Niillä operointi on tehokasta
- mutta arvoalueen ylittyessä luvut ”pyörähtävät ympäri” auton matkamittarin tavoin.

Matemaattinen kokonaislukutyyppi `Integer`, jolla on *aito ääretön* arvoalue

$$\ldots, -2, -1, 0, 1, 2, \ldots$$

- Niillä operointi on selvästi hitaampaa
- mutta ei pyörähdä ympäri

- Lukuvakioita voi käyttää hahmoissa (☞ 2.1.1).

  Esimerkkiohjelmassamme (☞ 2) käytimme `Integer`-vakiota rivin 2 hahmossa `(_,0)`.

- Käytimme ohjelmassamme tyypin `Integer` seuraavia operaatioita:

  kertolaskua `*` riveillä 6 ja 8

  vähennyslaskua `-` rivillä 6

  jakolaskua `div` rivillä 8

  vertailua `==` vakioon `0` implisiittisesti rivin 2 hahmossa

  parittomuustestiä `odd` rivin 5 vahdissa

  parillisuustestiä `even` rivin 7 vahdissa.

Liukuluvut ovat IEEE 754 -standardin mukaiset

$$\pm\text{mantissa} \times 10^{\text{eksponentti}}$$

missä

tyypin nimi	mantissan koko	eksponentin väli
`Float`	24 bittiä	-125...+128
`Double`	53 bittiä	-1021...+1024

(Itse asiassa IEEE-`Double`n pitää olla *vähintään* näin tarkka: esimerkiksi Intel-prosessoreissa sen mantissassa on 80 bittiä silloin kun se on rekisterissä.)

- Kokonais- ja liukulukutyypeillä on samanlaisia operaatioita kuin muissakin ohjelmointikielissä:

  aritmeettisia operaatioita kuten yhteen-, vähennys- ja kertolasku

  vertailuoperaattoreita kuten samuus == sekä sen vastakohta /=, <, >, <= ja >=.

  Nämä kaikki kirjoitetaan tuttuun tapaan *operaattoreiksi* argumenttiensa *väliin*.

- Haskell-kielessä ei ole automaattisia muunnoksia lukutyyppien välillä.

- Sen sijaan `Prelude` tarjoaa seuraavat apufunktiot.

`fromIntegral` on muunnos kummasta tahansa kokonaislukutyypistä mihin tahansa kokonais- tai liukulukutyyppiin.

`truncate` *leikkaa pois desimaaliosan* kumman tahansa tyyppisestä liukuluvusta ja palauttaa kokonaisosan kummassa tahansa kokonaislukutyypissä.

`round` on kuten `truncate` mutta *pyöristäen* lähimpään.

`floor` eli $\lfloor \ldots \rfloor$ on kuten `truncate` mutta palauttaa *edellisen* korkeintaan yhtä suuren kokonaisluvun.

`ceiling` eli $\lceil \ldots \rceil$ on kuten `truncate` mutta palauttaa *seuraavan* vähintään yhtä suuren kokonaisluvun.

2.1.3 Totuusarvot

- Haskell-kielen totuusarvotyypin nimi on `Bool`.
- Sen vakiot ovat `False` ja `True`.
  - Ne siis *alkavat SUURELLA* kirjaimella.
  - Niitäkin saa käyttää hahmoissa (☞ 2.1.1).
- Tyypillä `Bool` on tutut operaattorit `&&` ja `||` sekä apufunktio `not`.

- Nämä operaattoritkin on määritelty tavallisina funktioina

```
True  && x      =  x
False && _      =  False

True  || _      =  True
False || x      =  x
```

koska Haskell-kielen *laiska suoritusmekanismi (☞ 1.3.6) takaa, että ne oikaisevat* oman suorituksensa:

Esimerkiksi valittaessa kumpi haaroista soveltuu lausekkeelle p `&&` q sievennetään vain ensimmäinen alilauseke p; jälkimmäistä alilauseketta q tarvitaan vasta jos p on `True`.

- Kielessä on myös tuttu ehtolauseke

```
if c
   then t
   else f
```

mutta hahmojen (☞ 2.1.1) vuoksi sellaisia kirjoitetaan harvemmin kuin muissa kielissä.

2.1.4 Monikot

- Haskell-kielessä on sisäänrakennettuna erilaiset *monikot* (tuples).

- Monikkotyyppi kirjoitetaan

$$(tyyppi_1, tyyppi_2, tyyppi_3, \ldots, tyyppi_k)$$

  ja se ilmoittaa että

  – tämän tyypin arvoissa on k nimetöntä kenttää

  – järjestyksessä i. kentässä on arvo, jonka tyyppi on $tyyppi_i$.

- Tämä on ensimmäinen esimerkki *tyyppikonstruktorista* (type constructor): operaatiosta, joka rakentaa annettujen tyyppien $tyyppi_i$ pohjalta uuden tyypin.

- Esimerkiksi `(Int,Bool,(Integer,Int))` on sellaisten monikoiden tyyppi, jossa

  1. kentässä on `Int`
  2. kentässä on `Bool`
  3. kentässä on toinen 2-kenttäinen monikko, jonka
     1. kentässä on `Integer`
     2. kentässä on `Int`.

- Monikkotyyppisen arvon tuottava *lauseke* kirjoitetaan vastaavasti

$$(lauseke_1, lauseke_2, lauseke_3, \ldots, lauseke_k)$$

missä kukin $lauseke_i$ on vastaavaa tyyppiä $tyyppi_i$.

- Monikkotyypin hahmotkin (☞ 2.1.1) ovat samaa muotoa

$$(hahmo_1, hahmo_2, hahmo_3, \ldots, hahmo_k).$$

- Haskell-kielen yleisperiaate onkin, että tietynlaisen *arvon tuottava lauseke* ja sellaista arvoa *tutkiva hahmo* kirjoitetaan mahdollisimman samalla tavalla.

- Esimerkkiohjelmassamme (☞ 2)

  rivin 2 hahmo ei käyttänyt ensimmäisen kentän arvoa mutta vaati että toisen pitää olla 0

  rivin 4 hahmo antoi kenttien arvoille nimet `x` ja `n`, joilla niihin voitiin viitata riveillä 5–8.

- Sopivilla hahmoilla päästään täten käsiksi monikon nimettömiin kenttiin.

- Myös funktion tulostyyppinä voi olla monikko.

  Tätä käytetään kun funktion pitää antaa moniosainen tulos.

- Esimerkiksi vakiokirjasto `Prelude` tarjoaa funktion

  $$\texttt{quotRem}\ n\ m$$

  joka jakaa kokonaisluvun n kokonaisluvulla m ja palauttaa arvonaan parin

  $$(\mathit{osamäärä}, \mathit{jakojäännös}).$$

- Kirjoitetaan funktio, joka paloittelee saamansa kokonaisluvun sen viimeiseen bittiin ja sitä edeltävien bittien muodostamaan kokonaislukuun:

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

- Apufunktio `lastBit'` palauttaa siis arvonaan parin, jonka

  1. kenttä saadaan testaamalla, onko syötteen toinen kenttä eli jakojäännös `r` yhtä kuin `1`

  2. kenttä saadaan kopioimalla syötteen ensimmäinen kenttä `q` eli osamäärä.

- Haskell-kielessä on myös monikko ilman yhtään kenttää (eli tapaus $k = 0$).

- Sen tyyppi, ainoa hahmo ja ainoa lauseke kirjoitetaan samalla tavalla: `()`.

- 1-kenttäistä monikkoa (eli tapausta $k = 1$) ei puolestaan ole:

  - Tyyppi, hahmo tai lauseke muotoa (α) tulkitaan samaksi tyypiksi, hahmoksi tai lausekkeeksi kuin α.

  - Siten tyypeissä, hahmoissa ja lausekkeissa saa käyttää vapaasti kaarisulkuja `(...)` selventämään niiden rakennetta.

2.2 Funktiot

- Haskell-kielessä on tyyppikonstruktori (☞ 2.1.4)

  parametrin tyyppi -> arvon tyyppi

  joka siis konstruoi kyseisen *funktiotyypin*.

- Funktioidenhan todettiin (☞ 1.3.8) olevan täysivaltaisia arvoja, joten näilläkin arvoilla pitää olla tyyppi, onhan Haskell vahvasti tyypitetty kieli.

- Funktion määrittely (☞ 2.2.1) onkin kyseisen funktiotyyppisen vakioarvon nimeämistä.
  - vakiona pysyy funktion laskennassa käytettävä lauseke
  - parametrin eri arvoilla saadaan eri tulosarvoja.

- Funktiotyypeillä on kuitenkin joitakin rajoituksia:

  Samuusvertailu f == g on mahdotonta — se ei ole algoritmisesti ratkeavaa.

  Purkaminen osiin hahmonsovituksella (☞ 2.1.1) — se vaatisi samuusvertailun.

2.2.1 Funktion määrittely

- Nyt esitellään Haskell-funktion nimeltä f määrittelysyntaksi.

- pow-esimerkkimme (☞ 1.3) sisältää kaikki tässä kuvattavat piirteet.

- Syntaksia täydennetään myöhemmin pienellä lisäyksellä (☞ 2.2.5).

- Määrittelyssä käytetään formalisoitavaa käsitettä *lauseke*:
  - Muuttujan nimet ja eri tyyppien vakiot ovat pienimpiä lausekkeita.
  - Monikkoarvon laskeva lauseke (☞ 2.1.4) on esimerkki mutkikkaammasta lausekkeesta.
  - Tätä induktiivista määritelmää laajennetaan sitä mukaa kun perehdymme Haskell-kieleen syvemmin.

1. Ensin voi tulla vapaaehtoinen esittely muotoa:

   f :: *parametri*n tyyppi -> *arvo*n tyyppi

2. Sitten tulee peräkkäin yksi tai useampia sääntöjä.
   Kukin sääntö on muotoa

   f hahmo haarat

   (a) *hahmo*n (☞ 2.1.1) ja *parametri*n tyyppi ovat samat

   (b) jokainen *haara* on muotoa

   | *vahti* =
   tulos

   (c) *vahti* on mikä tahansa *lauseke* tyyppiä `Bool`
   (☞ 2.1.3)

   (d) *tulos* on mikä tahansa *lauseke* jonka tyyppi on sama
   kuin *arvo*n.

- Sääntö voi myös koostua vain yhdestä haarasta, jolla ei ole vahtia.
  - Tässä erikoistapauksessa sääntö voidaan kirjoittaa suoraan

```
f hahmo =
    tulos
```

  - Se on lyhennysmerkintä aina todelle vahdille

```
f hahmo
  | True =
    tulos
```

  - Esimerkkimme ensimmäinen sääntö oli tällainen.

- Tämän erikoistapauksen erikoistapaus on funktio g, jolla ei ole parametria.
  - Vapaaehtoinen esittely kutistuu muotoon

    g :: *arvo*n tyyppi
  - Ainoa sääntö kutistuu muotoon

    g =
    tulos
  - Nyt *hahmo* jää pois, koska ei ole *parametri*a jota sovittaa.
  - Samoin *vahti* jää pois, koska ei ole *hahmo*a jonka semantiikkaa tutkia.
  - Jäljellä jää siis *nimetyn vakioarvon* g määritelmä.

2.2.2 Funktionkutsu

- Haskell-kielessä funktionkutsu on *lauseke*

$$lauseke_{\text{funktio}}\ lauseke_{\text{argumentti}}$$

jossa

  - $lauseke_{\text{funktio}}$ on lauseke, jonka tyyppi on

    *parametri*n tyyppi -> *arvo*n tyyppi

  - $lauseke_{\text{argumentti}}$ on lauseke, jonka tyyppi on sama kuin *parametri*n

  - koko lausekkeen tyyppi on sama kuin *arvo*n.

- Siis funktion g kutsu argumentilla a ilmaistaan kirjoittamalla ne peräkkäin (välissä tyhjää):

$$g \sqcup a$$

- Kutsuttavan funktion g roolissa voi olla *lauseke*.
  - Haskell-ohjelmassa voidaan siis laskea, mitä funktiota tässä kohdassa halutaan tällä kertaa kutsua.
  - Tätä mahdollisuutta hyödynnetään esimerkiksi korkeamman kertaluvun funktioissa (☞ 1.3.7), joissa kutsutaan parametrina saatua funktiota.

- Funktiokutsun semantiikka on Haskell-kielen suoritusmekanismin keskeinen osa.

- Kutsun sieventäminen aloitetaan sieventämällä kutsuttava $\mathit{lauseke}_{\text{funktio}}$ loppuun saakka; näin selviää se funktio f, jota nyt kutsutaan.

- Kutsun argumenttiin $\mathit{lauseke}_{\text{argumentti}}$ **ei vielä kosketa, vasta kun/jos on pakko!**

  - Tässä Haskell poikkeaa selvästi useimmista muista ohjelmointikielistä.

  - Tämä onkin keskeinen osa sen *laiskaa suoritusmekanismia*.

- Aloitetaan kutsuttavan funktion f ensimmäisestä säännöstä. Se on muotoa

$$
\begin{array}{l}
f\ \mathit{hahmo}_1 \\
\quad \mid\ \mathit{vahti}_1\ = \\
\quad\quad \mathit{tulos}_1 \\
\quad \mid\ \mathit{vahti}_2\ = \\
\quad\quad \mathit{tulos}_2 \\
\quad \mid\ \mathit{vahti}_3\ = \\
\quad\quad \mathit{tulos}_3 \\
\quad \vdots \\
\quad \mid\ \mathit{vahti}_m\ = \\
\quad\quad \mathit{tulos}_m
\end{array}
$$

- Muista: haarassa on ainakin yksi *tulos* ja sillä *vahti*.

- Oikean haaran valinnassa peruskysymys on: ”Sopivatko $hahmo_1$ ja $lauseke_{\text{argumentti}}$ toisiinsa?”

- Määritellään siis sovitussäännöt joilla tämä kysymys ratkeaa.

- Kun myöhemmin laajennamme kieltämme uusilla tyypeillä ja tyyppikonstruktoreilla, niin samalla laajennamme näitä hahmoja ja sovitussääntöjä vastaavasti.

1. Jos $hahmo_1$ on muuttujan nimi x niin tietenkin sopii:

   - *Korvataan* muuttujan x jokainen esiintymä tässä säännössä sen arvolla $lauseke_{\text{argumentti}}$ Leibnizin lain (☞ 1.3) hengessä.

   - Tässä ohjelmointikielen puhtaus yksinkertaistaa sen semantiikkaa:

     $lauseke_{\text{argumentti}}$ ei sisällä muokattavia muistipaikkoja, joten ei tarvitse selittää, saavatko muuttujan x eri esiintymät siitä saman vai eri kopion.

   - Toteutustasolla ne jakavat saman kopion (☞ 7.1).

2. Jos $hahmo_1$ on alaviiva '_' niin ei tarvitse edes korvata.

3. Jos $hahmo_1$ on jokin vakio c, niin $lauseke_{\text{argumentti}}$ on sievennettävä näillä samoilla sievennyssäännöillä, jotta nähdään onko sen lopullinen arvo c.

4. Jos $hahmo_1$ on monikkohahmo

$$(hahmo'_1, hahmo'_2, hahmo'_3, \ldots, hahmo'_k):$$

   - sievennetään $lauseke_{\text{argumentti}}$ muotoon

$$(lauseke'_1, lauseke'_2, lauseke'_3, \ldots, lauseke'_k)$$

     mutta jätetään jokainen $lauseke'_i$ vielä sieventämättä

   - Yritetään sovittaa jokainen $hahmo'_i$ ja $lauseke'_i$ näillä samoilla sovitussäännöillä.

- Jos $hahmo_1$ ja $lauseke_{\text{argumentti}}$ eivät sovi yhteen, niin sievennyksessä siirrytään funktion f seuraavaan sääntöön 2,3,4,...

  – Epäonnistuneen hahmonsovituksen aikana tehdyt korvaukset unohtuvat, nehän tehtiin vain tähän sopimattomaan sääntöön.

  – Näin hahmon muuttujien näkyvyysalue on täsmälleen kyseinen sääntö.

  – Toteutustasolla muistetaan kaikki hahmonsovituksen tekemät sievennykset (☞ 7.1).

- Jos $hahmo_1$ ja $lauseke_{\text{argumentti}}$ sopivat yhteen, niin jatketaan ensimmäiseen haaraan

$$\begin{array}{l} \mid\ vahti_1\ = \\ \quad tulos_1 \end{array}$$

1. Sievennetään $vahti_1$ kokonaan.

2. Jos tulos on `False`, niin siirrytään seuraavaan haaraan 2,3,4... jos sellainen on olemassa; muuten seuraavaan sääntöön.

3. Jos tulos on `True`, niin kutsun tulos on laskettu: se on lauseke $tulos_1$, joka palautetaan sieventämättä.

- Jos funktiolla f ei ole seuraavaa sääntöä johon siirtyä, niin sievennys eli ohjelmansuoritus päättyy *ajonaikaiseen virheeseen:*

  Funktiota f ei olekaan määritelty argumenttinsa $lauseke_{\text{argumentti}}$ arvolla.