Mikko Häyrynen

Toistorakenteet C#:ssa ja Haskellissa

Tietotekniikan kandidaatintutkielma

20. joulukuuta 2017

Jyväskylän yliopisto

Tietotekniikka

Tekijä: Mikko Häyrynen

Yhteystiedot: mianhayr@student.jyu.fi

Työn nimi: Toistorakenteet C#:ssa ja Haskellissa

Title in English: Repetition in C# and Haskell

Työ: Kandidaatintutkielma

Sivumäärä: 28+0

Tiivistelmä: Tutkielmassa vertaillaan funktionaalista ja imperatiivista ohjelmointiparadigmaa toistorakenteiden osalta. Vertailussa tarkastellaan Jyväskylän yliopiston ohjelmointikursseilla käsiteltäviä C#– ja Haskell–ohjelmointikieliä, jotka edustavat merkittävästi erilaisia suunnitteluperiaatteita ja määrittelevät ensisijaiset toistorakenteensa eri lähtökohdista käsin. Eroavaisuuksien ja aiemman tutkimustiedon perusteella jäsennetään, miksi funktionaalisen paradigman omaksuminen tuottaa vaikeuksia imperatiiviseen ohjelmointitapaan totuttautuneille ohjelmoijille.

Avainsanat: C#, Haskell, silmukka, rekursio

Abstract: Imperative and functional programming languages implement repetition from different standpoints. This thesis compares two drastically different languages explored on entry level courses in the University of Jyväskylä – C# and Haskell. It examines their differences and analyzes the underlying reasons for said differences, also seeking to answer why programmers acquainted with imperative languages struggle while adopting the functional mindset.

Keywords: C#, Haskell, loop, recursion

Sisältö

1 Johdanto

Yleinen kiinnostus funktionaalisia ohjelmointikieliä kohtaan lisääntyy jatkuvasti, ja uuden paradigman omaksuminen vaatii monelta ohjelmoijalta ajattelutavan muutosta. Tämän tutkielman tarkoitus on jäsentää funktionaalisten ja imperatiivisten ohjelmointikielten merkittävimpiä eroja, erityisesti toistorakenteiden osalta. Lisäksi tutkielma tarkastelee toiston pysäyttämisen kannalta välttämättömiä ehtorakenteita, sekä rakenteiden yhdistelmien syntaktista ja semanttista merkitystä.

Ohjelmointi aloitetaan lähes poikkeuksetta jollakin imperatiivisella kielellä. Jyväskylän yliopistossa tämä kieli on Microsoftin kehittämä C#. Useita paradigmoja toteuttavana kielenä C# ei ole yksikäsitteisesti lokeroitavissa, mutta se on ennen kaikkea olio- ja komponenttisuuntautunut, imperatiivisesti ohjelmoitava kieli (Microsoft 2010). Koska oliosuuntautuneisuus on kielen merkittävin suunnitteluperiaate, C#:lla kirjoitetut ohjelmat ovat jatkuvasti muuttuvassa tilassa. Muuttuva tila mahdollistaa myös silmukoiden kirjoittamisen, ja silmukat ovatkin ensisijainen tapa toteuttaa toistoa C#:ssa sekä muissa C-sukuisissa kielissä. Tutkielmassa esitellään C#:n eri silmukkarakenteiden syntaksi ja käyttötarkoitukset. Myös rekursio on ajoittain hyödyllinen tapa toteuttaa toistoa C#:ssa, joten rekursiivisten funktiokutsujen syntaksi ja semantiikka esitellään – pääosin siksi, että Haskell-kieliset rekursiorakenteet saavat jonkin vertailukohteen. C#:n toteutuksena tutkielmassa käytetään `csc`-kääntäjää.

Funktio-ohjelmointi aloitetaan Jyväskylän yliopistossa puhtaasti funktionaalisella Haskell-kielellä, jossa laskenta käsitetään matemaattisina kuvauksina. Funktionaalisen ohjelmoinnin algoritmit eivät koostu silmukan kaltaisista, toinen toisensa perään suoritettavista askelista, vaan ne ovat parametrit lopputulokseksi kuvaavia funktioita (Hughes 1990). Tästä syystä Haskell ei tue C#:lle tyypillistä muuttuvaa tilaa, ja klassisen silmukan toteuttaminen ei ole kielessä luontevaa. Matemaattisiin kuvauksiin perustuvassa, deklaratiivisessa ohjelmointitavassa toisto toteutetaan rekursiolla, eli esittämällä laskettava lauseke itsensä avulla (Hinsen 2009). Tutkielmassa esitellään Haskell-funktioiden yleinen syntaksi, erityisesti rekursiivisten funktioi-

den osalta. Rekursion yleisyyden vuoksi kieli sisältää monia valmiita abstraktiorakenteita, jotka usein piilottavat eksplisiittisen rekursion ohjelmoijalta. Myös näistä rakenteista tärkeimmät käydään läpi. Haskellin toteutuksena tutkielmassa käytetään `ghc`-kääntäjää.

Koska ikuisesti itseään toistavat rakenteet eivät useinkaan ole mielekkäitä, toistorakenteen tulee sisältää jonkinlainen lopetusehto. C#:n kaikki silmukkarakenteet ovat lähtökohtaisesti ehdollisia, mutta kieli tukee myös omavaraisia ehtorakenteita. Näistä tavallisimmat esitellään rekursiivisten funktiokutsujen yhteydessä, molempien kielien osalta. Tutkielmassa vertaillaan toistorakenteille välttämättömän ehdollisuuden toteuttamista eri kielissä; erityisesti if- ja case-rakenteisiin kiinnitetään huomioita, sillä ne esiintyvät kielissä samoilla nimillä, mutta eri semantiikalla.

Ohjelmoinnin opetusta sekä oppimista on tutkittu kattavasti niin imperatiivisten (Lahtinen ym. 2005), kuin funktionaalistenkin kielten osalta (Tirronen ym. 2015). Toistorakenteiden soveltaminen on todettu merkittäväksi ongelmakohdaksi kielestä riippumatta. Tutkielmassa perehdytään kielten eroavaisuuksiin myös tästä näkökulmasta käsin ja pohditaan, mitä hyötyjä ja haittoja kummankin paradigman soveltamisesta on aloittelevalle ohjelmoijalle.

Lukijalta ei odoteta asiantuntemusta kummastakaan käsiteltävistä kielistä, mutta ohjelmoinnin peruskäsitteistön hallinta on hyödyksi. Oliosuuntautuneen ja funktionaalisen ohjelmoinnin yhtäläisyyksistä ja eroavaisuuksista kiinnostuneille suositeltakoon teosta *Object-Oriented vs. Functional Programming – Bridging the Divide Between Opposing Paradigms* (Warburton 2016).

2 Toistorakenteet C#-kielessä

Osiossa esitellään C#:n silmukkarakenteet, `goto`-lause, ehtorakenteet ja rekursiiviset funktiot. Kaikissa ohjelmakoodiesimerkeissä on otettu käyttöön `System`-nimiavaruus eksplisiittisten luokkaviitteiden kirjoittamisen vähentämiseksi.

2.1 Silmukkarakenteet

C# tukee neljää erilaista silmukkarakennetta (Microsoft 2010). Yksinkertaisin niistä on `while`-silmukka, joka toistaa sisältöään aina, kun määritelty totuusarvotyyppinen ehto on voimassa. Alla olevan silmukan ulkopuolella määritellään kokonaislukutyyppinen muuttuja `i`, jonka arvoa silmukan ehdossa tarkastellaan. Arvon ollessa alle 10 silmukan runko-osa suoritetaan; rungon sisältö tulostaa muuttujan `i` arvon ja kasvattaa sitten samaa arvoa yhdellä. Sisäisen lohkon suorituksen jälkeen kontrolli siirtyy takaisin silmukan aloitusriville ja toistoehto evaluoidaan uudestaan. Esimerkin silmukka siis tulostaa kokonaisluvut suljetulta väliltä $[0, 9]$.

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;
}
```

`do`-silmukka on kuin `while`-silmukka, mutta sen runko-osa suoritetaan vähintään kerran. Toistoehto siis evaluoidaan vasta rungon suorituksen jälkeen. Alla oleva `do`-silmukka pyytää käyttäjältä merkkijonoja, kunnes syötetty merkkijono on ”lopeta”.

```
string syote;
do
{
    syote = Console.ReadLine();
} while (syote != "lopeta");
```

Usein silmukan suorituksessa halutaan hyödyntää muuttujaa, jonka arvoa muutetaan jokaisen iteraation jälkeen. Tämä rakenne on niin yleinen, että sen ilmaisemisen avuksi kehitettiin ALGOL 60 -ohjelmointikielessä `for`-silmukka (Backus ym. 1963). Myös C# tukee `for`-silmukkaa, joka koostuu alustusosiosta, toistoehdosta ja iteraatio-operaatiosta. Alla oleva `for`-silmukka tuottaa saman lopputuloksen kuin aiemman esimerkin `while`-silmukka, mutta on syntaksiltaan huomattavasti tiiviimpi.

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
```

Lueteltavien tietorakenteiden käsittelyä varten on myös kehitetty `foreach`-silmukka, joka suorittaa runkonsa kerran jokaista tietorakenteen alkiota kohden. Alla oleva silmukka tulostaa taulukossa olevat kokonaisluvut 0–9.

```
int[] t = { 0,1,2,3,4,5,6,7,8,9 };
foreach (int i in t)
    Console.WriteLine(i);
```

`foreach`-silmukka on tyypillinen esimerkki yksinkertaistuksesta, jonka tarkoituksena on tehdä ohjelmakoodista miellyttävämpää kirjoittaa ja lukea. Tällaisia rakenteita on alettu kutsumaan syntaktiseksi sokeriksi (Landin 1964) ja ilmaisu on jäänyt käyttöön tietojenkäsittelytieteessä. Nykyisin syntaktista sokeria käytetään C#:ssa jopa enemmän kuin vastaavia sokerittomia rakenteita (Kim ym. 2013).

Minkä tahansa C#:n silmukkarakenteen voi pysäyttää `break`-lauseella, jolloin kontrolli siirtyy suoraan ulos silmukasta. Runko-osan suoritusta voidaan myös rajata `continue`-lauseella, joka siirtää kontrollin takaisin silmukan esittelyriville.

2.2 goto

`goto`-lauseella voidaan siirtää kontrolli suoraan mihin tahansa samalla näkyvyysalueella sijaitsevaan nimettyyn kenttään. Seuraavassa esimerkissä määritellään kaksi kenttää: `loop` ja `end`. Siirtymistä näiden kenttien välillä ohjataan `if`-lauseella. Esimerkki tulostaa jälleen kokonaisluvut 0–9.

```
int i = 0;
loop:
    if (i < 10)
    {
        Console.WriteLine(i);
        i++;
        goto loop;
    }
    else goto end;
end: ;
```

`goto`-lausetta voidaan käyttää `break`-lauseen sijasta sisäkkäisten silmukoiden rungosta poistumiseen (Microsoft 2010). Lause muistuttaa kone- ja välikielissä yleisiä alhaisen tason hyppykäskyjä ja sen käyttö johtaa usein heikkoon luettavuuteen moderneissa ohjelmointikielissä. Struktuaalisena kielenä C# tarjoaa monia helpommin hallittavia kontrollirakenteita, joten yleensä `goto`-lausetta pyritään käyttämään vain erityistapauksissa. Käyttöä on vastustettu kielestä riippumatta viittaamalla rakenteen liialliseen primitiivisyyteen (Dijkstra 1968). Empiirisellä tutkimuksella on myös osoitettu, että käyttö johtaa hajanaiseen rakenteeseen ohjelmakoodissa, huonoon suorituskykyyn ja ylimääräiseen testaukseen (Benander ym. 1990). Merkittävästi C#:n kaltainen Java-ohjelmointikieli ei salli `goto`-lauseiden käyttöä lainkaan (Gosling ym. 2017).

2.3 Ehtorakenteet

C#:n silmukoissa ehdollisuus on sisäänrakennettuna, mutta `goto`-lauseen ja rekursiivisten funktioiden tapauksessa ehdollisuus tulee toteuttaa erillisillä ehtorakenteilla. C#:ssa tällaisia ovat `if`-lause, `if`-lauseke ja `switch`-rakenne (Microsoft 2010).

`if`-lause kostuu ehdosta, runko-osasta ja mahdollisesta `else`-osasta. Seuraava ohjelmakoodi tulostaa tekstin ”alle”:

```
int i = 5;
if (i < 10)
    Console.WriteLine("alle");
else Console.WriteLine("yli");
```

`?:` -operaattorilla ehdollisuus voidaan sisällyttää lausekkeeseen. Ehtolausekkeet kirjoitetaan muotoon `ehto ? arvo1 : arvo2` ja ne evaluoituvat ehdon totuusarvon mukaan toiseksi `:`-merkillä erotelluista arvoista. Myös tämä koodiesimerkki tulostaa tekstin ”alle”:

```
int i = 5;
Console.WriteLine(i < 10 ? "alle" : "yli");
```

Jos ehdossa halutaan verrata samaa arvoa useisiin ennalta määrättyihin arvoihin, luettavuutta voidaan parantaa `switch`-rakenteella. `switch`-rakenteessa hahmonsovitus vertaa muuttujan arvoa `case`-avainsanoilla esiteltyihin arvoihin. Alla oleva esimerkki tulostaa numeerista arvoa vastaavan merkkijonon.

```
int i = 2;
switch (i)
{
    case 1: Console.WriteLine("yksi");
            break;
    case 2: Console.WriteLine("kaksi");
            break;
    case 3: Console.WriteLine("kolme");
            break;
    default: Console.WriteLine("muu luku");
             break;
}
```

2.4 Rekursio

Rekursiivinen funktio määritellään itsensä avulla. Hallittu rekursio jaetaan perustapaukseen ja rekursiiviseen tapaukseen; suorituksen aikana rekursiivinen funktio kutsuu itseään toisteisesti redusoidulla syötteellä, kunnes rekursio päättyy perustapaukseen. Alla oleva rekusiivinen esimerkkifunktio laskee luvun `n` kertoman.

```
static int fac(int n)
{
    if (n <= 0) return 1;
    return n * fac (n-1);
}
```

Vaikka toisto toteutetaan C#:ssa yleensä silmukalla ja rekursio on verrattain harvoin käytetty työkalu, on sen hallitseminen ajoittain hyödyllistä. Jotkin algoritmit, kuten kertoman tai Fibonaccin lukujonon laskeminen, voivat olla intuitiivisempia kirjoittaa rekursion avulla. Käsitteen hahmottaminen myös helpottaa funktionaalisten kielten periaatteiden ymmärtämistä.

Churchin-Turingin teesi väittää, että kaikki laskettavissa oleva voidaan laskea Turingin koneella (Copeland 2000). Koska iteratiiviset ja rekursiiviset rakenteet ovat laskettavissa, teesin nojalla niitä voidaan pitää yhdenvertaisina ja kaikki rekurviisiset rakenteet voitaisiin kirjoittaa myös iteratiivisessa muodossa.

3 Toistorakenteet Haskell-kielessä

Haskell on puhtaasti funktionaalinen, laiska ja staattisesti tyypitetty ohjelmointikieli. Tämä tarkoittaa, että kaikki kielellä kirjoitettavat funktiot ovat puhtaita matemaattisessa mielessä – funktiot tuottavat samoilla parametreilla aina saman lopputuloksen, eivätkä ne voi aiheuttaa tuntemattomia sivuvaikutuksia (Hughes 1990). Kielen laiskuus takaa sen, että lausekkeiden arvoja ei lasketa, ennen kuin niitä tarvitaan esimerkiksi tulostettaessa. Laiskuuden ansiosta kielessä voi muun muassa operoida loputtomilla datarakenteilla (Lipovaca 2011). Staattisesti tyypitetyssä kielessä lausekkeiden tyypit määritellään jo käännösaikana, mikä puolestaan vähentää ajonaikaisia virheitä. Funktionaalisella kielellä kirjoitettu ohjelma on myös helppo altistaa matemaattiselle analyysille (Segal 1994).

Haskell on sukua 50-luvulta lähtien kehitetylle Lisp-ohjelmointikielelle ja perustuu lambdakalkyylin loogisen laskennan malliin (Doets ym. 2004). Kielen syntyaikoina kaavaillut suunnitteluperiaatteet painottavat formaalia ja avoimesti saatavilla olevaa kielen määritelmää sekä soveltuvuutta akateemiseen käyttöön (Hudak ym. 2007), mutta kieli on muotoutunut käyttökelpoiseksi myös ohjelmistokehityksessä.

Kappaleen ohjelmakoodiesimerkeissä funktioiden tyypit on kirjoitettu auki selkeyden vuoksi, vaikka `ghc`-kääntäjä osaisi itsekin päätellä ne Hindley-Milner -tyyppijärjestelmän ansiosta (Marlow 2010).

3.1 Eksplisiittinen rekursio

Koska Haskellin puhdas funktionaalisuus ei salli muuttuvaa tilaa, toisteiset lausekkeet on määriteltävä niiden itsensä avulla. Rekursiiviset määritelmät ovat siis erittäin keskeinen osa kielen semantiikkaa.

Klassinen esimerkki rekursiosta Haskellissa on funktio, joka laskee kokonaislukulistan summan. Perustapaukseksi määritellään tyhjä lista, jonka summa on 0. Rekursiivinen tapaus käsittelee epätyhjää listaa, jonka summa lasketaan lisäämällä ensim-

mäiseen alkioon summa kaikista seuraavista alkioista. Ensimmäisen alkion erotettamisessa hyödynnetään hahmonsovitusta, joka esitellään tarkemmin seuraavassa osiossa.

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

Eksplisiittisessä rekursiossa rekursiivinen funktiokutsu on selkeästi näkyvissä; esimerkin summafunktio kutsuu itseään toisteisesti redusoidulla listaparametrilla, kunnes lista on tyhjä ja rekursio päättyy perustapaukseen.

3.2 Ehtorakenteet

Ehdollisuutta saadaan Haskellissa aikaiseksi pääosin hahmonsovituksella sekä `if`- ja `case`-lausekkeilla (Marlow 2010). Näistä jälkimmäiset on helppo samastaa C#:n vastaaviin rakenteisiin sillä erotuksella, että Haskell ei sisällä lauseita, vaan puhtaasti funktionaalisia lausekkeita. Haskellin laajaa kontrollirakenteiden kirjoa voi vapaasti yhdistellä funktiomääritelmissä.

Hahmonsovituksen syntaksia sivuttiin jo aiemmin rekursiivisen summafunktion toteutuksen parissa. Tämän funktion parametrilistaus tulkitaan niin, että ensimmäinen määritelmä otetaan käyttöön parametrin ollessa tyhjä lista. Toista määritelmää käytetään silloin, kun lista koostuu ensimmäisestä alkiosta ja sitä seuraavasta listasta. Hahmoja sovitetaan määritelmiin ilmenemisjärjestyksen mukaan – ylhäältä alaspäin. Mikäli parametri ei sovi yhteenkään määritellyistä hahmoista, ohjelma kaatuu ajonaikaiseen virheeseen. `ghc` tarjoaa varoituksen puutteellisista hahmoista, jos `-W` -lippu on käytössä (GHC Team 2017).

Sidonnan sijasta hahmonsovituksessa voi hyödyntää myös vakiolausekkeita. Seuraava kertomafunktio sovittaa parametrejaan hahmoihin 0 ja `n`. Nollan kertomaksi määritellään 1 ja muiden lukujen kertomat määritellään rekursiivisesti. Huomaa, että tämän funktion kutsuminen negatiivisella arvolla johtaa ikuiseen rekursioon.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Hahmonsovitus sisältää myös muita ominaisuuksia, kuten ns. jokerihahmon _ sekä arvon ja hahmon yhtäaikaisen sidonnan `@`-merkillä. Nämä ominaisuudet sivuutetaan tässä tutkielmassa. Yleistyksenä mainittakoon, että kaikki muuttujalistaukset voidaan Haskellissa samastaa hahmonsovitukseen, vaikka sidonta suoritettaisiinkin näennäisesti hahmottomien muuttujien suhteen. Kertomafunktion hahmonsovitus ei voi epäonnistua, jolloin puhutaan *kiistattomista* hahmoista.

Ehdollisuus voidaan myös toteuttaa ns. *guardilla*, joka valitsee ehdon perusteella oikean rungon funktiolle. Ehdot kirjoitetaan `|`-merkillä edellettyinä parametrien ja funktion määritelmän väliin ja ne voivat sisältää hahmonsovituksia. Seuraava esimerkki laskee jälleen luvun kertoman, mutta ei tuota loputonta rekursiota negatiivisen parametrin tapauksessa. Avainsana `otherwise` on vain standardikirjaston tarjoama synonyymi vakiofunktiolle, joka tuottaa totuusarvon `True` (Marlow 2010).

```
fac :: Int -> Int
fac n | n <= 0    = 1
      | otherwise =  n * fac (n-1)
```

Haskell sisältää myös seuraavan esimerkin omaisia, tavanomaisia ehtolausekkeita. Avainsanojensa perusteella nämä muistuttavat hämäävästi C#:n ehtolauseita, mutta semanttisen merkityksen vastine on myös C#:ssa ehtolauseke. Lauseke siis evaluoituu ehdon perusteella toiseksi kahdesta arvosta. Esimerkin funktio kuvaa parametrina saadun luvun merkkijonoksi.

```
size :: Int -> String
size n = if n < 10 then "small" else "big"
```

Seuraava funktio kertoo `case`-lauseketta hyödyntäen sille annetun listan pituuden merkkijonomuodossa. `case`-lausekkeet ovat tapa sovittaa lausekkeita useisiin ehtoihin ja muistuttavat semantiikaltaan suuresti hahmonsovitusta. Todellisuudessa hahmonsovitus onkin vain syntaktista sokeria `case`-lausekkeille (Marlow 2010). Syntaksi on helposti samastettavissa myös C#:n `switch`-lauseeseen. Lukijan tulee

kuitenkin pitää mielessään, että *lausekkeet* evaluoituvat aina konkreettisiksi arvoiksi, eivätkä voi sisältää irrallisia suoritettavia lauseita, toisin kuin C#:n `switch`-lauseet.

```
length :: [a] -> String
length xs = case xs of
              []     -> "empty"
              [x]    -> "singleton"
              (x:xs) -> "longer"
```

3.3 Abstraktiorakenteet

Rekursiivisten määritelmien yleisyyden ja yleistettävyyden vuoksi Haskelliin on kehitetty lukemattomia abstraktiorakenteita. Kielen standardikirjasto sisältää yleiskäyttöisiä funktioita, jotka piilottavat eksplisiittisen rekursion ohjelmoijalta. Tässä osiossa esitellään muutama näistä funktioista.

Modulaarisuus on eräs tärkeimpiä menestyksekkään ohjelmointikielen piirteitä (Hughes 1990). Haskellin modulaarisuus ja polymorfismi kumpuavat funktionaalisten kielten perusominaisuuksien lisäksi *tyyppiluokista*. Tyyppiluokkien ansiosta funktioita voidaan yleistää toimimaan kaikille saman tyyppiluokan instansseille. Esimerkki yleiskäyttöisyydestä nähtiin jo aiemmin: `length`-funktion toiminta on täysin riippumaton listan alkioiden tyypistä. Tyyppiluokista puhutaan tässä tutkielmassa vain käsitteellisellä tasolla.

`map` on standardikirjastoon sisältyvä funktio, joka suorittaa annetun funktion jokaiselle listan alkiolle. Tämä muistuttaa C#:n `foreach`-rakennetta. `map` on sisäisesti määritelty rekursiivisesti, joten se piilottaa eksplisiittisen rekursion. Seuraavan esimerkin mukaisesti `map`-funktion avulla voidaan esimerkiksi laskea merkkijonolistan alkioiden pituuksien summa tai lisätä merkkejä listan jokaiseen sanaan. Listoilla toimiva `map`-funktio on myös itsessään yleistettävissä: `fmap` kykenee suorittamaan saman operaation mille tahansa *funktorille*. Funktori on tyyppiluokka kontekstissa olevalle datalle – tällaisia tyyppejä voivat olla esimerkiksi lista, taulukko, vektori, hajautustaulu tai binääripuu.

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

lengthsum :: [String] -> Int
lengthsum xs = sum (map length xs)

shoutwords :: [String] -> [String]
shoutwords xs = map (++ "!") xs
```

`foldr` on funktio, joka laskostaa `Foldable`-tyyppiluokan instanssin uudeksi arvoksi jonkin funktion avulla. Yksinkertaistuksena voidaan ajatella, että `foldr` yhdistää listan alkiot yhdeksi arvoksi halutulla tavalla, mutta todellisuudessa funktion toimintaa ei ole rajattu pelkkiin listoihin. `foldr` ottaa parametrinaan binäärifunktion, akkumulaattorin eli lopputuloksen aloitusarvon sekä laskostettavan datarakenteen. Alla olevassa esimerkissä on esitelty, miten listan summa lasketaan `foldr`-funktion avulla: binäärifunktiona käytetään summaoperaattoria ja summan aloitusarvoksi määritellään 0. Toinen esimerkkifunktio `cons` esittää, kuinka `foldr`-funktion avulla voidaan muodostaa uusia listoja. Myös aiemmin esitellyn `map`-funktion voi toteuttaa `foldr`:n avulla (Hutton 1990).

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

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

cons :: [Int] -> [Int]
cons xs = foldr (:) [] xs
```

`foldr` juontaa juurensa 30-luvulla syntyneeseen rekursioteoriaan (Kleene 1952). Se on erittäin yleiskäyttöinen funktio ja esiintyy myös muissa funktionaalisissa kielissä (Hughes 1990). Funktionaalisille kielille on tyypillistä todistaa toimintaansa matemaattisella induktiolla; `foldr` on hyödyllinen apuväline myös näissä todistuksissa (Hutton 1990).

Haskell sisältää lukemattomasti muita abstraktioita, joihin ei tässä katsauksessa perehdytä. Erityismaininnan arvoinen funktio on kuitenkin `fix`, joka yleistää koko rekursion käsitteen. Alla esitetään tämän funktion määritelmä ja kuinka sen avulla voidaan laskea Fibonaccin lukujono loputtomaan listaan. Esimerkin tulkinta jätetään lukijan tehtäväksi – tosin huomautettakoon, että `fibs` ei sisällä rekursiivista funktiokutsua.

```
fix :: (a -> a) -> a
fix f = f (fix f)

fibs = fix f [0,1]
    where f rec (x:y:z) = y:rec (y:x+y:z)
```

4 Idiomeista, suorituskyvystä ja optimoinnista

Olemme huomanneet, että C# ja Haskell lähestyvät toiston toteuttamista eri suunnista. Molemmilla kielillä on omat idiominsa: C# hyödyntää silmukoita ja Haskell rekursiorakenteita. Tässä osiossa pureudutaan kielten suunnitteluperiaatteisiin sekä selvitetään, miksi kielet on toteutettu eri tavoin ja kuinka tämä vaikuttaa ohjelmoijan ajattelutapaan. Eroavaisuuksia löytyy syntaksin ja semantiikan lisäksi myös tehokkuudesta. Erityisesti Haskellin tehokkuus yllättää monet ohjelmoijat. Ohjelmointikielen tehokkuus riippuu merkittävästi kääntäjän sisäisestä toteutuksesta, joten myös käännösprosessiin tutustutaan pinnallisella tasolla.

4.1 Suunnitteluperiaatteet ja kielikohtaiset intuitiot

C# on suunniteltu kehittyneeksi osaksi C:n ja C++:n jatkumoa. Ohjelmointikielten hierarkiassa se sijaitsee edeltajiänsä korkeammalla tasolla ja sisältää monia korkean tason kielille tyypillisiä ominaisuuksia, kuten poikkeuksia, automaattista roskienkeruuta, tietorakenteiden rajaamista ja alustamattomien muuttujien käytön estoa (Microsoft 2010). Kielen on tarkoitus olla helposti lähestyttävä C- tai C++ -ohjelmoijalle (Microsoft 2010), joten suuri osa näiden kielten ominaisuuksista on peritty myös C#:iin. Virhealttiit konseptit kielestä on kuitenkin joko jätetty pois tai korvattu turvallisemmilla vaihtoehdoilla – esimerkiksi osoitintyyppien sijasta C#:ssa suositaan delegaatteja ja `ref`-avainsanaa (Microsoft 2010).

Haskellin suunnittelussa on painotettu ennen kaikkea puhtautta, laiskuutta ja vahvaa tyypitystä (Hudak ym. 2007). Nämä periaatteet rajaavat jo itsessään kielen toteutusta merkittävästi ja karsivat pois suuren osan rakenteista, joita esimerkiksi C# hyödyntää. Rajoitukset ovat kuitenkin perusteltuja ja tuovat mukanaan hyötyjä: puhtaalla koodilla ei voi olla sivuvaikutuksia, laiskuus mahdollistaa matemaattisesti elegantit rakenteet ja vahva tyyppijärjestelmä vähentää virheitä sekä parantaa modulaarisuutta. *Viitteellinen läpinäkyvyys* myös takaa, että mikä tahansa lauseke voidaan korvata sen arvolla ilman, että ohjelman toiminta muuttuu (Hughes 1990).

Yleisesti ottaen funktionaaliset kielet kykenevät käsittelemään funktioita korkeammalla tasolla kuin imperatiiviset kielet (Hughes 1990): ensiluokkaisia funktioita voidaan käsitellä arvoina, osittaiset funktioapplikaatiot voivat palauttaa uusia funktioita ja funktioita voidaan yhdistää matemaattisella yhdistetyn funktion notaatiolla. Monet näistä ominaisuuksista ovat viime vuosina periytyneet tavalla tai toisella myös C#:iin. Ensiluokkaiset funktiot ja lambdalausekkeet ovat esimerkkejä funktionaalisille kielille ominaisista ominaisuuksista, jotka ovat nykyään osa monia imperatiivisia kieliä. Kielten eri suunnitteluperiaatteet johtavat erilaisiin ajattelutapoihin. Siinä, missä C#-ohjelmoija näkee silmukan ja apumuuttujia, voi Haskell-ohjelmoija nähdä `map`- ja `foldr`-funktiot.

4.2 Tietorakenteet, kääntäjät ja yleinen optimointi

C#:n tyypillisin tietorakenne on staattinen taulukko, joka tukee vakioaikaista elementtien indeksointia. Dynaamisena tietorakenteena C#:ssa käytetään lähtökohtaisesti listaa, joka on todellisuudessa toteutettu staattisella taulukolla ja kasvaa dynaamisesti tarpeen vaatiessa (Microsoft 2010). Tällaiset tietorakenteet ovat tyypillisiä imperatiivisissa ohjelmointikielissä ja niiden läpikäyminen on intuitiivista `for`- ja `foreach`-silmukoilla.

Haskellin tyypillisin tietorakenne on yhteen suuntaan linkitetty lista, joka ei tue vakioaikaista alkioiden indeksointia (Marlow 2010). Linkitettyä listaa pitkin on edettävä alkio kerrallaan alusta loppuun. Aikavaativuus ei kuitenkaan yleensä muodostu ongelmaksi, koska listoja käytetään pikemminkin kontrollirakenteena tietorakenteen sijasta (Coutts ym. 2007). Optimoiva kääntäjä voikin hyödyntää *fuusiota*, jonka avulla väliaikaiset tietorakenteet voidaan muuttaa konkreettisiksi kontrollirakenteiksi (Coutts ym. 2007). Esimerkiksi lista voi siis muuttua käännöksen aikana silmukaksi. Tehokkaiksi tietorakenteiksi Haskell tarjoaa esimerkiksi `Array`- ja `Vector`-kirjastojen toteutukset.

C# käännetään Microsoftin CIL-välikieleksi, jota JIT-tulkki kääntää konekieleksi ajon aikana (Microsoft 2010). Käännösprosessin aikana koodista poistetaan syntaktinen

sokeri ja generoitua välikieltä optimoidaan, esimerkiksi poistamalla vakioehtoja sisältävät ehtolauseet tai siirtämällä iteraation aikana muuttumattomat lausekkeet laskettaviksi silmukan ulkopuolella. C#:n ja muiden imperatiivisten kielten kääntäminen on melko suoraviivaista, koska kielten rakenne on kehittynyt von Neumannin arkkitehtuurin mukaan. Silmukat ja ehtorakenteet on helppo kääntää hyppykäskyiksi.

Funktionaaliset kielet tukevat rakenteita, joille ei ole selkeää vastinetta konekielisessä koodissa. Kielet sisältävät paljon syntaktista sokeria, kuten hahmonsovitusta, lambdalausekkeita ja listakomprehensiota, jotka täytyy yksinkertaistaa muiksi rakenteiksi (Marlow ym. 2012). Sokerinpoisto tuottaa yksinkertaista ydinkieltä, joka vastaa ominaisuuksiltaan suppeata lambdakalkyylia. Sitä optimoidaan iteratiivisesti poistamalla turhia koodilohkoja sekä ylikirjoittamalla turhia funktiomääritelmiä (Marlow ym. 2012). Koska Haskellilla kirjoitettu ohjelmakoodi on viitteellisesti läpinäkyvää, kääntäjä voi tehdä siitä merkittäviä oletuksia ja suorittaa optimointia raskaammalla kädellä kuin imperatiivisen ohjelmakoodin tapauksessa.

Yksittäisten prosessoriytimien kehitys on viime vuosina hidastunut, ja nykyään prosessorien tehokkuutta parannetaan lähinnä kasvattamalla ytimien määrää (Hinsen 2009). Tämän myötä myös rinnakkaislaskennan merkitys on kasvanut ja laskentaa haluttaisiin suorittaa usealla ytimellä yhdenaikaisesti. Ohjelman sisäinen muuttuva tila monimutkaistaa rinnakkaislaskennan toteuttamista, sillä koordinoimattomat muutokset dataan voivat aiheuttaa konflikteja. Haskellin puhdas funktionaalisuus ja viitteellinen läpinäkyvyys auttavat rinnakkaisuuden toteuttamisessa: muuttujien puute poistaa konfliktien mahdollisuuden ja ohjelman rakenne voidaan transformoida tukemaan rinnakkaislaskentaa ilman, että sen lopputulos muuttuu (Hinsen 2009). Kielten tehokkuus vaihtelee merkittävästi sovelluksen mukaan, mutta Haskellin on ajoittain todettu olevan jopa oliosuuntautuneita kieliä tehokkaampi, vaikka sillä kirjoitettu koodi on usein kertaluokan verran lyhyempää (Alic ym. 2016).

4.3 Rekursio ja häntäkutsun poisto

Eräs mielenkiintoinen optimointitekniikka on rekursiivisen häntäkutsun poisto. Funktiota sanotaan *häntärekursiiviseksi* silloin, kun funktion määritelmä loppuu rekursiiviseen kutsuun (Jones 1992). Alla on esitelty häntärekursiivinen C#-aliohjelma, jonka on tarkoitus tulostaa loputtomasti kokonaislukuja. Kutsuttaessa tämä aliohjelma kaatuu ennen pitkää `StackOverflow`-poikkeukseen, koska loputtomat rekursiiviset kutsut aiheuttavat ohjelman pinokehyksen ylityksen.

```
void Rec(int n)
{
    Console.WriteLine(n);
    Rec(n+1);
}
```

Vastaava tuloste saadaan aikaiseksi myös alla olevalla Haskell-funktiolla. Tämä funktio ei kuitenkaan kaadu, vaan tulostaa loputtomasti kokonaislukuja. Miksi?

```
rec :: Int -> IO ()
rec n = print n >> rec (n+1)
```

Ratkaisu piilee rekursiivisen häntäkutsun optimoinnissa. Koska funktion määritelmä päättyy rekursiiviseen kutsuun, ei tämän kutsun suorittamisen jälkeen ole mitään syytä palata takaisin edelliseen funktioaktivaatioon – siispä aiemman aktivaation pinokehys voidaan tuhota häntäkutsun yhteydessä (Jones 1992). Häntäkutsu siis rinnastetaan funktion sisäiseen hyppykäskyyn ja rekursio voidaan muuttaa iteraatioksi (Jones 1992). Optimoitu häntärekursiivinen funktio vaatii vain vakiomäärän pinomuistia toimiakseen. Aikavaativuus tosin paranee vain, jos roskienkeruu on toteutettu kielessä nopeammaksi kuin muistinvaraus pinosta (Clinger 1998).

CIL–välikieli mahdollistaa tämän tekniikan (Microsoft 2010) ja Microsoftin funktionaalinen F#-kieli myös hyödyntää sitä. C#:n `csc`-kääntäjä ei sisällä vastaavaa optimointia, koska häntärekursiiviset funktiot ovat kielessä melko harvinaisia.

5 Kompastuskivet kielten oppimisessa

Aloittelijoille optimaalisesta ohjelmointikielestä on käyty kiivasta keskustelua, ja mielipiteet ovat myös muuttuneet vuosien saatossa. Toiset pitävät oliosuuntautuneisuutta parhaana tapana aloittaa ohjelmointi (Kölling ym. 1999), mutta myös funktionaalisia kieliä on opetettu ensimmäisen vuoden opiskelijoille (Joosten ym. 1993). Tutkielmassa esitellyt toistorakenteet – silmukat ja rekursio – tuottavat ongelmia kielestä riippumatta (Dale 2006). Ohjelmoinnin alkeiskursseilla opiskelijoiden taitotaso vaihtelee suuresti, joten kaikille sopivaa oppimateriaalia on vaikea kehittää (Lahtinen ym. 2005).

5.1 Aloittelijoiden ongelmat C#:n kanssa

Vaikka C# on ennen kaikkea olio-ohjelmointikieli, Jyväskylän yliopiston Ohjelmointi 1 -kurssilla sitä käytetään ohjelmoinnin peruskonseptien opetuksessa pääosin rakenteellisena kielenä. Rakenteellisessa ohjelmoinnissa korostuu erityisesti ratkaistavan ongelman jakaminen pienempiin, hallittaviin kokonaisuuksiin. Kielen oliosuuntautuneisuus voi häiritä kurssilla käsiteltävän ajattelutavan sisäistämistä, sillä olio-ohjelmointikielellä tulisi ohjelmoida oliosuuntautuneesti (Kölling ym. 1999). Olio-ohjelmoinnin erityiskäsitteet, kuten luokat, perintä ja staattisuus, eivät ole olennaisia yksinkertaisen rakenteellisen ohjelmoinnin kannalta, mutta ovat aiheuttaneet ongelmia alkeiskursseilla (Dale 2006). C# on kuitenkin edeltäjiänsä aloittelijaystävällisempi, ja esimerkiksi merkittävästi C#:n kaltainen Java-kieli on koettu helpommaksi oppia kuin C++, jossa mm. osoittimien käyttö on yleistä (Lahtinen ym. 2005).

Silmukan rakenne voi olla haasteellista hahmottaa, sillä se koostuu useasta erillisestä osasta – `for`-silmukassa määritellään monta suoritettavaa lausetta yhdellä rivillä, mikä ei ole C#:n muissa rakenteissa tavanomaista. Toisaalta silmukkarakenteiden oppimisen on osoitettu tuottavan ongelmia lähinnä opiskelijoille, joille ohjelmoinnin opiskelu on muutenkin haastavaa (Lahtinen ym. 2005). Yleinen ongelma on myös algoritmin muuttaminen ohjelmakoodiksi. Aloittelevat ohjelmoijat yrittä-

vät usein käyttää rakenteita, jotka eivät sovi käytössä olevaan ohjelmointikieleen, tai yleensäkään mihinkään ohjelmointikieleen (Someren 1990). Aloittelijat myös käyttävät tavallista enemmän kyseenalaisia rakenteita, kuten rakenteellisen ohjelmoinnin parissa kiivasta keskustelua herättäviä `goto`- ja `break`-lauseita (Sorva ym. 2016). Tämä viittaa siihen, että kielelle ominainen ajattelutapa sisäistetään vasta kokemuksen karttuessa.

5.2 Haskell ja paradigmanvaihdos

Rekursion konsepti on opiskelijoiden mielestä eräs vaikeimmin hahmotettavia asioita ohjelmointikursseilla – osittain siksi, että sitä on vaikea yhdistää reaalimaailmaan (Lahtinen ym. 2005). Myös opettajat kokevat rekursion haastavana käsitteenä opettaa (Dale 2006). Tämä muodostuu ongelmaksi erityisesti funktionaalisissa kielissä, joissa rekursiiviset funktiomääritelmät ovat olennainen osa kielen semantiikkaa (Segal 1994). Rekursio tuottaa aloittelijoilla usein ajonaikaisia virheitä, koska määritelmiin on unohtunut liittää rekursion lopettava perustapaus (Tirronen ym. 2015). Myös abstraktiorakenteiden laaja kirjo vaikeuttaa ohjelmien tulkitsemista. Esimerkiksi `foldr`-funktion avulla kirjoitetut funktiot voisivat olla yksinkertaisempia lukea, jos niissä olisi käytetty eksplisiittistä rekursiota (Hutton 1990).

`ghc`-kääntäjän tuottamat virheviestit ovat aloittelijoille haastavaa luettavaa. Haskellin tyyppitarkastus perustuu Hindley-Milner -tyyppijärjestelmään (Damas ym. 1982), joka tuottaa virheen ristiriitaisten tyyppimääritelmien tapauksessa. Valtaosa aloittelijoiden virheistä Haskellin parissa onkin tyyppivirheitä (Tirronen ym. 2015). Tyyppivirhe voi tapahtua esimerkiksi rekursiivisen funktiokutsun yhteydessä, jos funktiolle ei ole annettu oikeaa määrää oikean tyyppisiä parametreja. Yleisiä ovat myös syntaksivirheet (Tirronen ym. 2015). Tämä saattaa johtua toisen ohjelmointikielen syntaksin sopimattomasta soveltamisesta Haskellissa (Someren 1990).

Haskell sisältää lukuisia ominaisuuksia, joille C# ei tarjoa suoraa vastinetta, kuten matemaattisen yhdistetyn funktion notaation, funktion osittaisapplikaation ja eksplisiittisen applikaation `$`-operaattorilla. Nämä ominaisuudet ovat osoittautuneet

haastaviksi aloittelijoille (Tirronen ym. 2015). Myös Haskell-kielisten ohjelmien debuggaus on haastavaa (Lipovaca 2011). Tämä johtunee osittain siitä, että funktionaalisella kielellä kirjoitetut ohjelmat muuttuvat käännösprosessin aikana huomattavasti enemmän kuin niiden imperatiiviset vastineet, joten konekielistä koodia on vaikea yhdistää lähdekoodiin rivi kerrallaan askellettavaksi. Vaikka muuttuvan tilan puutetta pidetään eräänä Haskellin suurimmista vahvuuksista, mielekkäitä ohjelmia on lähes mahdotonta kirjoittaa ilman jonkilaista simuloitua muuttuvaa tilaa. Puhtaasti funktionaalisia kieliä onkin tästä syystä kritisoitu, vetoamalla muuttuvan tilan intuitiiviseen ja keskeiseen asemaan ohjelmoinnissa sekä sen monimutkaiseen toteutukseen funktiokielissä (Spinnelis ym. 2009).

5.3 Sovellukset oppimistehtävissä

Ohjelmointikielten syntaksi ja semantiikka eivät itsessään ole suuria kompastuskiviä aloitteleville ohjelmoille, vaan ongelmia tuottaa pikemminkin eri rakenteiden sovittaminen yhteen (Soloway 1986). Esimerkiksi silmukka- tai rekursiorakenteet eivät siis välttämättä ole itsessään haastavia hahmottaa, vaan muodostuvat ongelmiksi vasta sovellettaessa muiden rakenteiden kanssa. Tärkeää olisikin tunnistaa yksittäisistä, yksinkertaisista tehtävistä niiden yhteiset teemat, joiden avulla konkreettisen tehtävän ratkaisun voi abstrahoida yleispäteväksi algoritmiksi (Soloway 1986). Ideaalitilanteessa näitä yleistettyjä algoritmeja tulisi pystyä hyödyntämään jatkotehtävissä, eli oppilaan tulisi osata hajottaa uusi ongelma aiemmin opittujen ratkaisujen yhdistelmäksi (Soloway 1986).

Suuri osa imperatiivisen ohjelmoinnin alkeiskurssien soveltavista tehtävistä vaatii jonkinlaisen silmukkarakenteen hyödyntämistä. Eräs tunnetuimpia tehtäviä on jo 80-luvulla kehitetty ns. *rainfall problem*, jossa tarkoituksena on laskea sademäärien keskiarvo. Yleisimmässä variantissa huomioidaan ainoastaan positiiviset arvot ja laskeminen tulee lopettaa jonkin erityisarvoisen alkion kohdalla. Tehtävä on periaatteiltaan yksinkertainen, mutta tutkimusten mukaan vain murto-osa alkeiskurssien oppilaista onnistuu ratkaisemaan sen kurssin lopuksi (Seppälä ym. 2015). Yleisimmät virheet johtuvat nollalla jakamisesta ja kelvottomien alkioiden huomioi-

misesta, tosin valtaosa vastauksista sisältää myös toimivia osuuksia (Seppälä ym. 2015). Tämä viittaa siihen, että vaikka oppilas hallitsisi yksittäiset rakenteet, niiden yhdistäminen tuottaa usein ongelmia.

Sademäärätehtävää on tutkittu myös funktionaalisissa kielissä, joissa ongelmaa lähestytään hieman eri suunnasta: syötteiden suora lukeminen on harvinaista, tietorakenteet ovat erilaisia ja toisto toteutetaan rekursiolla (Fisler 2014). Funktionaaliset kielet tukevat monia valmiita operaatioita (kuten `map` ja `filter`) tietorakenteille, joten koko syötteen läpikäyminen tuottaa harvoin ongelmia. Funktionaalisten kielten alkeiskursseilla syötteiden lukeminen ohjelman ulkopuolelta on kuitenkin vähäistä, ja osa ongelmista imperatiivisissa kielissä johtuu juuri tästä (Fisler 2014). Useilla funktionaalisilla ohjelmointikielillä toteutettu tutkimus (Fisler 2014) on tuottanut merkittävästi parempia tuloksia kuin vastaava tutkimus C#:lla (Simon 2013).

6 Yhteenveto

Olemme huomanneet, että toiston toteuttamista lähestytään C#:ssa ja Haskellissa eri tavalla. Tämä huomio on yleistettävissä myös muihin imperatiivisiin ja puhtaasti funktionaalisiin ohjelmointikieliin; jos kieli ei salli muuttuvaa tilaa, klassiset silmukat ja hyppylauseet on korvattava rekursiivisilla määritelmillä. Molemmilla lähestymistavoilla on omat hyötynsä ja haittansa, jotka juontavat juurensa syvälle kielten suunnitteluperiaatteisiin. Lähestymistavasta riippumatta toistorakenteiden soveltaminen tuottaa ongelmia aloittelijoille.

Imperatiiviset ja funktionaaliset kielet eroavat toisistaan merkittävästi yleiseltä semantiikaltaan, mikä vaikuttaa myös kielten käännösprosessiin ja tehokkuuteen. Funktionaalinen lähdekoodi on kaukana prosessorin ymmärtämästä konekielestä ja hyödyntää mielenkiintoisia optimointitekniikoita muuttuakseen järkeväksi ajettavaksi ohjelmaksi. Paradigmojen välinen kuilu kuitenkin kapenee jatkuvasti, ja monia funktiokielten ominaisuuksia on peritty C#:iin ja muihin imperatiivisiin sekä oliosuuntautuneisiin kieliin.

Ohjelmoinnin opetusta on tutkittu paljon, ja funktionaalisen ohjelmoinnin yleistymisen myötä myös tätä paradigmaa on alettu hyödyntämään opetuksessa. Vertailevaa tutkimusta on tehty tunnettujen ohjelmointitehtävien avulla, ja tulokset ovat olleet funktionaalisten kielten kannalta lupaavia. Funktiokieliä sovelletaan paikoittain jopa ensimmäisillä ohjelmointikursseilla. Kielen valinnan vaikutusta aloittelevan ohjelmoijan ajattelumaailman kehitykseen tulisi tutkia lisää. Aloittelijalle optimaalisen kielen valinta on avoin kysymys, johon tämän tutkielman kirjoittaja ei odota vastausta lähivuosina.

Kirjallisuutta

Alic, D., Omanovic, S., & Giedrimas, V. 2016. *Comparative analysis of functional and object-oriented programming*. In Proceedings of MIPRO 2016.

Backus, J., Bauer, F., Green, J., Katz, J., McCarthy, J., Naur, P., Perlis, A., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J., van Wijngaarden, A. & Woodger, M. 1963. *Revised report on the algorithmic language ALGOL 60*.

Benander, B., Benander, C. & Gorla, N. 1990. *An empirical study of the use of the GOTO statement*. Journal of Systems and Software, Volume 11, Issue 3, s. 217–223.

Clinger, E. 1998. *Proper tail recursion and space efficiency*. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, s. 174–185.

Copeland, J. 2010. *The Church-Turing Thesis*. Saatavilla WWW-muodossa <URL: `http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html`>. Viitattu 16.11.2017.

Coutts, D., Leshchinskiy, R. & Stewart, D. 2007. *From Lists to Streams to Nothing at All*. In Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, s. 315–326.

Dale, N. 2006. *Most difficult topics in CS1: results of an online survey of educators*. ACM SIGCSE Bulletin, Volume 38, Issue 2, s. 49–53.

Damas, L. & Milner, R. 1982. *Principal type-schemes for functional programs*. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, s. 207–212.

Dijkstra, E. 1968. *A Case against the GO TO Statement*. Communications of the ACM, Volume 11, Issue 3, s. 147–148.

Doets, K & van Eijck, J. 2004. *The Haskell Road to Logic, Maths and Programming*. Toimittanut Mackie, I. London: College Publications.

Fisler, K. 2014. *The recurring rainfall problem*. Proceedings of the tenth annual conference on International computing education research, s. 35–42.

GHC Team. 2017. *GHC User's Guide Documentation — Release 8.2.1*.

Gosling, J., Joy, B., Steele, G. Bracha, G., Buckley, A. & Smith, D. 2017. *The Java Lan-*

guage Specification – Java SE 9 Edition.

Hinsen, K. 2009. *The Promises of Functional Programming*. Computing in Science & Engineering, Volume 11, Issue 4, s. 86–90.

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. 2007. *A history of Haskell: being lazy with class*. Proceedings of the third ACM SIGPLAN conference on History of programming languages, s. 1–55.

Hughes, J. 1990. "Why Functional Programming Matters". Teoksessa *Research Topics in Functional Programming*, toimittanut David A. Turner, s.17–42. Boston: Addison-Wesley Longman Publishing Co., Inc.

Hutton, G. 1999. *A tutorial on the universality and expressiveness of fold*. Journal of Functional Programming, Volume 9, Issue 4, s. 355–372.

Jones, R. 1992. *Tail recursion without space leaks*. Journal of Functional Programming, Volume 2, Issue 1, s. 73–79.

Joosten, S., Berg, K., & Hoeven, G. V. D. 1993. *Teaching functional programming to first-year students*. Journal of Functional Programming, Volume 3, Issue 11, s. 49–65.

Kim, D. & Yi, G. 2013. "Measuring Syntactic Sugar Usage in Programming Languages: An Empirical Study of C# and Java Projects". Teoksessa *Advances in Computer Science and its Applications*, toimittanut Jeong, H., Obaidat, M., Yen, N. & Park, J., s. 279-–284. Berlin: Springer Publishing.

Kleene, S. 1952. *Introduction to Metamathematics*. New York, NY: Ishi Press.

Kölling, M. 1999. *The Problem of Teaching Object-Oriented Programming, Part 1: Languages*. Journal of Object-Oriented Programming, Volume 11, Issue 8, s. 8–15.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H-M. 2005. *A study of the difficulties of novice programmers*. In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, s. 14–18.

Landin, P. 1963. *The mechanical evaluation of expressions*. The Computer Journal, Volume 6, Issue 4, s. 308–320.

Lipovaca, M. 2011. *Learn You a Haskell for Great Good!* San Francisco, CA: No Starch Press, Inc.

Marlow, Simon. 2010. *Haskell 2010 Language Report*. Saatavilla WWW-muodossa <URL: `https://www.haskell.org/onlinereport/haskell2010/`>.

Viitattu 15.10.2017.

Marlow, S. & Peyton Jones, S. 2012. "The Glasgow Haskell Compiler". Teoksessa *The Architecture of Open Source Applications International, Volume 2*, toimittanut Brown, A. & Wilson, G., s. 279–284. Morrisville, NC: Lulu Press, Inc.

Microsoft. 2010. *C# Reference*. Saatavilla WWW-muodossa <URL: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/`>. Viitattu 16.11.2017.

Microsoft. 2010. *CIL Tailcall*. Saatavilla WWW-muodossa <URL: `https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.tailcall`>. Viitattu 16.11.2017.

Peyton Jones, S. 1987. *The Implementation of Functional Programming Languages*. Hertfordshire, UK: Prentice Hall International.

Segal, J. 1994. *Empirical studies of functional programming learners evaluating recursive functions*. Instructional science, Volume 22, Issue 5, s. 385–411.

Seppälä, O., Ihantola, P., Isohanni, E., Sorva, J. & Vihavainen, A. 2015. *Do we know how difficult the rainfall problem is?* In Proceedings of the 15th Koli Calling Conference on Computing Education Research, s. 87–96.

Simon. 2013. *Soloway's Rainfall Problem Has Become Harder*. Proceedings of the 2013 Learning and Teaching in Computing and Engineering, s. 130–135.

Soloway, E. 1986. *Learning to program = learning to construct mechanisms and explanations*. Communications of the ACM, Volume 29, Issue 9, s. 850–859.

Someren, M. W. 1994. *What's wrong? Understanding beginners' problems with Prolog*. Instructional science, Volume 19, Issue 4, s. 357–282.

Sorva, J & Vihavainen, A. 2016. *Break statement considered*. ACM Inroads, Volume 7, Issue 3, s. 36–41.

Spinnelis, D. & Gousios, G. 2009. *Beautiful Architecture*. Sebastopol, CA: O'Reilly Media, Inc, s. 315–347.

Tirronen, V., Uusi-Mäkelä, S. & Isomöttönen, V. 2015. *Understanding beginners' mistakes with Haskell*. Journal of Functional Programming, Volume 25, Issue 11.

Warburton, R. 2016. *Object-Oriented vs. Functional Programming – Bridging the Divide Between Opposing Paradigms*. Sebastopol, CA: O'Reilly Media, Inc.