
Arttu Pennanen

# Pragmaattisen funktionaalisen ohjelmoinnin arviointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

18.11.2024

# Tiivistelmä


| | |
|---|---|
| Tekijä: | Arttu Pennanen |
| Otsikko: | Pragmaattisen funktionaalisen ohjelmoinnin arviointi |
| Sivumäärä: | 57 sivua |
| Aika: | 18.11.2024 |
| Tutkinto: | Insinööri (AMK) |
| Tutkinto-ohjelma: | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine: | Ohjelmistotuotanto |
| Ohjaajat: | FM Simo Silander |


---


Funktionaalista ohjelmointia tukee vahva teoreettinen tausta, ja sen käytännön toteutus eroaa olennaisesti muista ohjelmointiparadigmoista. Tässä insinöörityössä tarkastellaan, miten funktionaalisen ohjelmoinnin periaatteita voidaan pragmaattisesti sisällyttää ohjelmistoprojekteihin.

Käydään läpi, miten ohjelmistoympäristöön voidaan tuoda funktionaalisen ohjelmoinnin kieliagnostisia piirteitä, jotka eivät ole sidoksissa käytettävään ohjelmointikieleen. Koodiesimerkit, näkemykset ja perustelut pohjautuvat kokemukseen ohjelmistoyrityksessä sekä julkisiin lähteisiin, joissa käsitellään funktionaalisen ohjelmoinnin, kategoriateorian, ongelmien mallintamisen ja ohjelmoinnin periaatteita.

Tuloksena perustellaan funktionaalisen ohjelmoinnin käyttöä ohjelmien oikeellisuuden ja ylläpidettävyyden parantamiseksi. Näytetään, miten ohjelmat voidaan estää valehtelemasta ja miten operaatioita voidaan ketjuttaa funktioiden ja monadien avulla. Funktionaalista ohjelmointia voidaan liittää muihin ohjelmointiparadigmoihin, kunhan huolehditaan ohjelmiston monimutkaisuuden hallinnasta. Ohjelmoijat lähtökohtaisesti ajattelevat ongelmia ja ratkaisuja sen ohjelmointikielen näkökulmasta, jota he käyttävät.



Avainsanat: funktionaalinen ohjelmointi, JavaScript, TypeScript, koodin luettavuus, koodin ylläpidettävyys, koodin suorituskyky, kehittäjäkokemus


---

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

## Abstract


| | |
|---|---|
| Author: | Arttu Pennanen |
| Title: | Evaluating Pragmatic Application of Functional Programming |
| Number of Pages: | 57 pages |
| Date: | 18 November 2024 |
| Degree: | Bachelor of Engineering |
| Degree Programme: | Information and Communication Technology |
| Professional Major: | Software Engineering |
| Supervisors: | Simo Silander, M.Sc. |


---


Functional programming is underpinned by a robust theoretical foundation in mathematics, distinguishing it significantly from other programming paradigms. This thesis explores how the principles of functional programming can be pragmatically applied into software projects.

The center of discussion is introducing language-agnostic features of functional programming into the software environment. Code examples, insights, and justifications are drawn from experiences in a software company, and from public sources focusing on the principles of functional programming, category theory, problem modeling, and programming in general.

As a result, an argument is drawn in favor of functional programming to enhance the correctness and maintainability of programs. It is demonstrated how programs can be prevented from lying, and how operations can be chained using functions and monads. Functional programming can, and should, be integrated into other programming paradigms, provided that software complexity is taken care of. Programmers typically assume they are coding in the language they are using.



Keywords: functional programming, JavaScript, TypeScript, code readability, code maintainability, code performance, developer experience

## Lisenssit



## PDF-tiedoston linkit

Navigoinnin helpottamiseksi PDF-tiedostossa on käytössä linkkejä. PDF-katseluohjelmasta riippuen linkeissä näkyy ääriviivat:

- Punaiset ääriviivat: tiedoston sisäiset linkit sanastoon
- Vihreät ääriviivat: tiedoston sisäiset linkit lähdeluetteloon
- Siniset ääriviivat: ulkoiset linkit verkkosivuille.

Sisäisistä linkeistä voi palata takaisin näppäinyhdistelmällä `Alt` + `Vasen nuoli` (tai `Cmd` + `Vasen nuoli` ).

## Sisällys

## Sanasto

**Deklaratiivinen ohjelmointi:** Ohjelmointiparadigma, jossa ohjelmoija määrittelee, mitä lopputuloksen tulisi olla, mutta ei yksityiskohtaisesti sitä, miten tämä tulos saavutetaan.

**Funktionaalinen ohjelmointi:** Ohjelmointiparadigma, joka korostaa laskennan mallintamista funktioiden avulla.

**Imperatiivinen ohjelmointi:** Ohjelmointiparadigma, jossa ohjelmoija antaa tarkat ohjeet siitä, kuinka tehtävä suoritetaan vaihe vaiheelta. Tämä lähestymistapa keskittyy ohjelman tilan hallintaan ja muuttamiseen käskyjen avulla, kuten muuttujien asettaminen ja silmukoiden käyttö.

**JavaScript:** ohjelmointikieli (js).

**Joukko-oppi:** engl. *set theory*. Joukko-oppi on joukkojen ominaisuuksiin perehtynyt matematiikan osa-alue. Joukko-oppi on perustavanlaatuisessa merkityksessä tietorakenteissa. Joukko-opissa olennaisimpia laskutoimituksia ovat esimerkiksi unioni, leikkaus ja joukkoerotus.

**Kategoriateoria:** engl. *category theory*. Kategoriateoria on matematiikan osa-alue, joka tutkii erittäin yleistävällä tasolla rakenteiden välisiä suhteita ja yhteyksiä. Kategoriateoriaa käytetään funktionaalisen ohjelmoinnin apukeinona.

**Kieliagnostinen:** engl. *language agnostic*. Kieliagnostisuudella tarkoitetaan lähestymistapaa, jossa ei keskitytä tiettyyn ohjelmointikieleen, vaan tarkastellaan peruskäsitteitä ja -periaatteita yleisellä tasolla, sovellettavissa eri kieliin riippumatta niiden syntaksista tai erityispiirteistä.

**Koodin oikeellisuus:** engl. *correctness*. Koodin oikeellisuus on ohjelman kyky täyttää sille asetetut vaatimukset virheettömästi. Toisaalta ilman vaatimuksia koodi ei voi olla oikeellista.

**Korkeamman asteen funktio:** engl. *higher order function*. Funktiot, jotka voivat ottaa toisia funktioita syötteenään tai palauttavat funktioita tuloksena.

**Monadi:** engl. *monad*. Monadi on abstrakti tietorakenne, joka mahdollistaa ohjelman kontrollin virtaamisen läpinäkyvämmin ilman, että sivuvaikutuksia tarvitsee käsitellä suoraan. Monadi on alun perin kategoriateorian käsite, mutta sitä käytetään myös funktionaalisessa ohjelmoinnissa.

**Muuttumaton data:** engl. *immutable data*. Data jota ei voi muuttaa sen luomisen jälkeen. Muuttumattomuutta käytetään funktionaalisessa ohjelmoinnissa ohjelman luotettavuuden vuoksi.

**Ohjausrakenne:** engl. *control structures*. Ohjelmoinnissa käytettäviä rakenteita, jotka määrittävät ohjelman suorituksen kulun ja järjestyksen. Keskeisiä ovat ehtorakenteet (esim. `if`, `else`), silmukat (esim. `for`, `while`) ja haaraumat (esim. `goto`, `break`).

**Olio-ohjelmointi:** Ohjelmointiparadigma, joka perustuu olioiden, eli ohjelmakomponenttien, ympärille, jotka yhdistävät sekä dataa että siihen liittyviä toimintoja. Olio-ohjelmointi korostaa perinnän, kapseloinnin ja polymorfismin kaltaisia periaatteita.

**Puhdas funktio:** engl. *pure function*. Funktiot, jotka eivät vaikuta ohjelman tilaan ja palauttavat aina saman tuloksen samoilla syötteillä.

**Sivuvaikutus:** engl. *side effect*. Puhtaiden funktioiden ulkopuolella tapahtuvaa toimintaa, eli esimerkiksi tiedostojen lukeminen, http-pyyntöjen lähettäminen tai muuttujan mutatointi.

**TypeScript:** ohjelmointikieli (ts). JavaScript mutta mukaan on lisätty staattinen tyypitys.

**Yhdistetty funktio:** engl. *function composition*. Useammasta funktiosta koostettu funktio, jossa funktiot suoritetaan toinen toisensa jälkeen niin, että edellisen paluuarvo annetaan seuraavan syötteeksi.

# 1 Johdanto

Insinöörityössä tutkitaan funktionaalisen ohjelmoinnin käytänteitä sellaisissa ohjelmointikielissä, joissa funktionaalinen ohjelmointi ei ole ollut kielen luonnin peruste. Etsitään pragmaattista, tavoitelähtöistä puolta funktionaalisesta ohjelmoinnista, jossa teoriaa ei painoteta ilman selkeitä ja perusteltuja syitä. Funktionaalinen ohjelmointi on erittäin teoreettista ja akateemista lähtökohdiltaan.

Kokemusten ja median tarkkailun perusteella on huomattu funktionaalisten periaatteiden yhdistyvän yleiseen näkemykseen siitä, mitä ohjelmoinnin tulisi olla. Funktionaalinen ohjelmointi ei ole kuitenkaan noussut valtavirran suosioon. Insinöörityön tavoitteena on etsiä toimintaperiaatteita funktionaalisen ohjelmoinnin tuontiin mukaan arkiseen ohjelmointiin ilman pakonomaista teoriaa. Insinöörityössä kuitenkin etsitään funktionaalisen ohjelmoinnin teoriasta niitä osia, joilla voitaisiin kasvattaa funktionaalisen ohjelmoinnin pragmaattista mukaanottoa.

Insinöörityössä tutkitaan funktionaalisen ohjelmoinnin teoriaa hieman tavanomaisesta poikkeavasta näkökulmasta, jossa ohjelmat ajatellaan joukko-opin näkökulmasta olevan siirtymiä joukoista toisiin. Ei paneuduta kaikkiin käsitteisiin, joita tavanomaisesti tarkastellaan funktionaalisen ohjelmoinnin arvioinnin yhteydessä. Etsitään empiirisiä anekdootteja tunnettujen ohjelmistokehittäjien töistä, ja kokemuksista työympäristössä. Haastatellaan lyhyesti entistä Google-kehittäjää, joka puoltaa ohjelmointikielen omien ominaisuuksien käyttämistä.

Insinöörityön koodiesimerkeissä käytetään TypeScriptiä ja JavaScriptiä. Vaikka kaikki validi JavaScript-koodi on validia TypeScript-koodia [1], niin kielten nimiä käytetään työssä tilanteeseen sopien. Kirjoitetaan TypeScriptistä, kun sen tarjoamat tyypit JavaScriptin päälle ovat tilanteeseen nähden merkittäviä. Toisaalta kirjoitetaan JavaScriptistä, kun tyypit eivät ole esimerkille merkittäviä. Muuten pyritään pitämään käsitellyt asiat sellaisina, että niitä voisi hyödyntää myös muissa ei-funktionaalisissa ohjelmointikielissä.

## 2 Motivaatio

Ei ole yksiselitteistä, mitä funktionaalisen ohjelmoinnin tulisi olla ja miten sitä tulisi käyttää. Matemaatikot vannovat funktionaalisen ohjelmoinnin oikeellisuuden nimeen, ja pragmaatikot kauhistuvat. Kuuluisan ristiriitaiset ja mutkikkaat keskustelut hämärtävät pohjaa, minkä päälle funktionaalinen ohjelmointi on rakennettu. [2–5.]

Funktionaalisen ohjelmoinnin hyötyjä on usein sanottu löytyvän suurelta osin vain matemaattisissa tai datapainotteisissa ohjelmissa [6, s. 10]. Ajatus on haastettavissa.

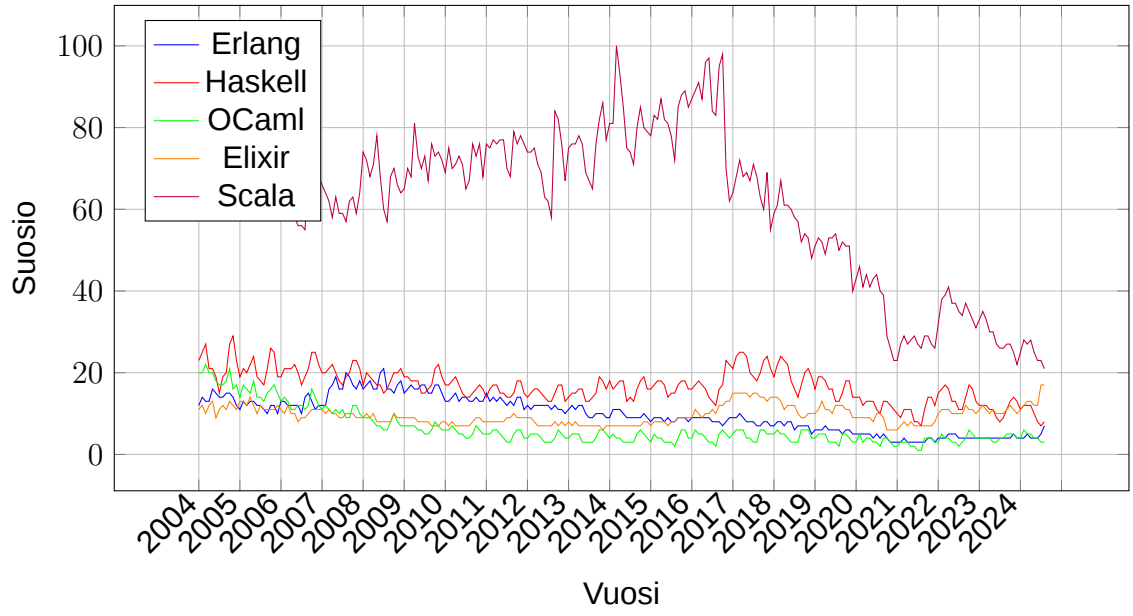
Ensimmäisenä huomiona on se, että suuri osa ohjelmista on juuri datapainoitteisia [7]. Tietokannassa on dataa, jota näytetään käyttöliittymässä. Samaan aikaan kerätään lisää dataa, jota näytetään toisissa käyttöliittymissä; toisin sanoen varsin datapainoitteista.

Toisena huomiona on se, että kaikki ongelmat voidaan mallintaa matemaattisesti. Funktionaalisen ohjelmoinnin Turing-vahva lambda-kalkyyli-pohja määritelmällisesti mahdollistaa sen, että funktionaalisessa ohjelmoinnissa voi toteuttaa kaiken saman kuin olio-ohjelmoinnissa [8; 9].

Ongelmien mallintaminen matemaattisesti ei kuitenkaan ole helppoa. Mallintaminen tuntuu vaativan vahvoja tyyppijärjestelmiä mallinnuksen järkevästi todistamiseksi.

Kuitenkin pikkuhiljaa funktionaalista ohjelmointia aletaan kutsumaan ilmiselväksi reitiksi ohjelmoida, ja esimerkiksi olio-ohjelmointia ylivertaisemmaksi [10, s. 1]. Jopa Robert C. Martin, tunnettu ohjelmointikäytäntöjen asiantuntija ja olio-ohjelmoinnin puolestapuhuja, on siirtynyt merkittävästi funktionaalisen ohjelmoinnin suuntaan [11; 12]. Lienee tarpeen selvittää, miten funktionaalisuutta saataisiin enemmän osaksi nykyohjelmoijan työkalupakkia.

Tiukasti funktionaaliseen ohjelmointiin tarkoitetut kielet eivät tunnu saavan nostetta päästäkseen yleiseen suosioon (kuva 1). Ero siirtyessä olio-ohjelmoinnista tai imperatiivisesta ohjelmoinnista funktionaaliseen ohjelmointiin voi olla kuin yöllä ja päivällä. Siksi funktionaalisen ohjelmoinnin keinoja ripotellaan osaksi suosittuja ohjelmointikieliä, että saataisiin vietyä todettuja parhaita funktionaalisen ohjelmoinnin paloja myös muihin paradigmoihin. Kenties ajatellaan niin, että varpaat on ensin kasteltava, ennen kuin voi hypätä syvään päätyyn.




Kuva 1: Funktionaalisten ohjelmointikielien Erlangin, Haskelin, OCamlin, Elixirin ja Scalan suosiokehitys vuosina 2004–2024. Kielet eivät ole olleet suuressa nousussa. [13.]

Ripottelun voi havaita esimerkiksi, kun vahvasti olio-ohjelmointiin sovellettua Javaa on tuotu kohti funktionaalista pelikenttää erilaisin kielen työkalujen, kuten lambdafunktioiden ja Stream-API:n metodien yhdistämisen avulla [14; 15]. Javaa voikin nykyään jo kutsua hybridiksi ohjelmointikieleksi funktionaalisen ja olio-ohjelmoinnin välillä [16, s. 50].

Myös JavaScriptissä Array-metodit, kuten `Array.prototype.map` [17], `Array.prototype.filter` [18] ja `Array.prototype.reduce` [19], ovat vakiintuneet keskeisiksi työkaluiksi [20; 21]. Kyseiset funktiot ovat laajalti käytössä funktionaalisessa ohjelmoinnissa. Myös JavaScript Set -tietorakenne (joukko) on juuri saamassa joukko-opille ja funktionaaliselle ohjelmoinnille olennaisia metodeja kuten `Set.p`

`rototype.union` [22] ja `Set.prototype.difference` [23], jotka palauttavat uusia Set-tietorakenteita (joukkoja) aiempien muokkaamisen sijaan funktionaaliselle ohjelmoinnille ominaisesti [24].

Myös kirjoitushetkellä tason 3 TC39-ominaisuusesitys JavaScript iteraattorien apumetodeille vahvistaa funktionaalista tapaa ohjelmoida [25]. Näiden metodien lisääminen mahdollistaa vielä johdonmukaisemman funktionaalisen koodin kirjoittamista.

## 2.1 Kokemukset funktionaalisesta ohjelmoinnista

Tehtäessä työtä osana Hoxhunt Oy:n tuotekehitystiimiä on saatu kokemusta funktionaalisesta ohjelmoinnista. Kokemuspohjaista tietoa on kerätty tiimiläisiltä sekä muuten omalta osalta. On nähty, kuinka funktionaalisen ohjelmoinnin käytänteitä käytetään, siten, että ensin niiden käyttöön on kannustettu, ja toisin pyritty estämään. Koodikatselmuksissa ja muissa keskusteluissa on usein kuultu ajatuksia kärkkäästi niin funktionaalisen ohjelmoinnin puolesta kuin vastaankin.

Funktionaalisen ohjelmoinnin kulmakivi on sen matemaattinen perusta [8; 9]. Toisinaan funktionaalista ohjelmointia on mielletty vain älykkyyden todistamiseksi, ja jota humoristisesti ajatellaan vain matematiikan maisterin tutkinnon suorittaneiden taitavan. Voi olla vaikea nähdä funktionaalisen ohjelmoinnin hyötyjä, jos kokee sen vain ylimääräisenä monimutkaisuutena.

Funktionaalisen koodin kääntöpuoli on sen selittävyys: deklaratiivisuus. Jos on nähty funktionaalisen ohjelmoinnin tuntemattomuuden tai havaitun monimutkaisuuden läpi, niin on huomattu, että funktionaalista ohjelmakoodia voi parhaimmillaan lukea kuin selkeää englannin kieltä. Tämä johtuu siitä, että funktionaalinen ohjelmakoodi on perusteiltaan (deklaratiivista ohjelmakoodia) suorien käskyjen sijaan (imperatiivinen ohjelmakoodi) [26].

Yhdistetyt funktiot, joissa funktioita suoritetaan toinen toisensa jälkeen, ja annetaan aina edellisen paluuarvo seuraavan syötteeksi, kertoo suoraan, mitä siinä

tapahtuu. Koodi dokumentoi itse itseään, kun kaikki ohjelman vaiheet ovat selkeästi nimetyissä palikoissa; erillisiä kommentteja ei koodiin tarvitse edes lisätä (koodiesimerkki 1).

```
1     const congratulateUsers = pipe(
2         filter(isActiveUser),
3         filter(isBirthdayToday),
4         map(createCongratulationMessage),
5         sendCongratulations
6     )
7     // Onnitellaan kaikkia asiakkaita, joilla on tänään
          syntymäpäivä!
8     congratulateUsers(users)
```

Koodiesimerkki 1: JavaScript-esimerkki yhdistetystä funktiosta. Funktioiden yksityiskohtia ei tarvitse tietää, että voi ymmärtää ohjelman kokonaiskuvan.

Toisaalta koodin deklaratiivisuuden ylistämisellä voi yrittää peittää muita ongelmia. Jos sokeana seuraa sääntöjä ja käytänteitä, voi jäädä muuten tärkeitä osa-alueita huomioimatta [27]. Jos pyrkii liiallisesti luettavuuteen deklaratiivisella koodilla, voi esimerkiksi ohjelman tehokkuus kärsiä.

Luettavuus, tehokkuus ja ylläpidettävyys ovat kolme keskeistä osa-aluetta, joita on painotettu eniten työpaikalla käydyissä keskusteluissa, kun arvioidaan funktionaalisen ohjelmointitavan laajempia hyötyjä. Nämä osa-alueet näkyvät merkittävinä myös muiden ohjelmointityylien arvioinnissa. Luettavuutta pidetään yleensä kaikkein tärkeimpänä. Ohjelmakoodin ei tarvitse olla tehokasta ennen kuin optimointia todella tarvitaan [28], ja ylläpidettävyyden perusta on vahvasti sidoksissa siihen, kuinka luettavaa koodi on.

## 2.2 Lähestymistapa

Tässä insinöörityössä tutkitaan, mitä tapahtuu, kun funktionaalista ohjelmointia ripotellaan proseduraaliseen ohjelmakoodiin. Tätä tarkastellaan sellaisen ohjelmointikielen näkökulmasta, jota ei ole suunniteltu erityisesti funktionaaliseen ohjelmointiin, mutta jossa se on kuitenkin mahdollista. Täksi ohjelmointikieleksi valikoitui JavaScript (ja TypeScript). Ohjelmointikieli valittiin aiemman osaamisen ja kokemusten perusteella, ja toisaalta myös siksi, että kieli on yksi markkinoi-

den käytetyimpiä, ja siten myös yksi olennaisimmista [29]. JavaScript on myös kielenä siitä mielenkiintoinen, että se mahdollistaa koodiabstraktioiden luomisen ilman suurta vaivaa ohjelmoijalta. Hätiköidyt abstraktiot johtavat epäjohdonmukaisuuteen ja lopulta uudelleenkäyttämättömyyteen [30]. Tämä tekee JavaScriptista haastavan, mutta antoisan kielen tutkia pragmaattista funktionaalisen ohjelmoinnin ripottelua.

Insinöörityössä käsiteltäviä asioita pyritään perkaamaan kieliagnostisesti, jolloin saadut tiedot ja taidot olisivat oleellisia myös kehittäjille, jotka eivät käytä TypeScriptiä tai JavaScriptiä.

Tarkoituksena on löytää tasapaino sille, miten funktionaalinen ohjelmointi kannattaa ottaa mukaan ohjelmistoprojektiin. Pohjalla on ajatus, että paras tapa olisi vain siirtyä kokonaan funktionaalisiin ohjelmointikieliin hybridikielten sijasta. Tätä ajatusta pyritään kumoamaan, sillä toivotaan, että funktionaalisuus voisi toimia myös kevyemmin osana ohjelmoijan arkea.

Kun jotain ei vaadita, se jää usein tekemättä tai ainakin puutteelliseksi. Siksi voi olla haastavaa ohjelmoida pragmaattista funktionaalista koodia JavaScriptissä. Kielessä, jossa juuri mitään sääntöjä ei pohjimmiltaan ole.

# 3 Teoreettinen tausta

Jotta voidaan löytää pragmaattinen näkökulma funktionaaliseen ohjelmointiin, on käsiteltävä funktionaalisen ohjelmoinnin teoreettisia perusteita ja keskeisiä käsitteitä.

Tarkastellaan joukko-opin sovelluksia ohjelmoinnissa käytännön tasolla. Sivutaan kategoriateorian osuutta funktionaalisessa ohjelmoinnissa ilman syvällistä paneutumista yksityiskohtiin. Sivutaan myös kategoriateoriaa, joka on funktionaalisen ohjelmoinnin teoreettisen taustan kulmakivi. Tarkemmin käydään läpi kategoriateorian monadi-rakennetta, ja sen toimintaa.

Tutkitaan, miten nämä käsitteet integroituvat moderniin ohjelmistokehitykseen,

kuten TypeScriptin käyttöön.

On tärkeää huomata, että tämän insinöörityön tavoitteena ei ole tarjota kattavaa opasta kaikkeen funktionaaliseen ohjelmointiin, vaan keskittyä valittuihin aihealueisiin, jotka tukevat työn pääteemoja ja osoittavat, miten teoreettiset käsitteet voidaan soveltaa käytännön ohjelmointihaasteisiin.

Valitut aihealueet pyritään avaamaan kieliagnostisesti, mutta kuitenkin JavaScript- sekä TypeScript-ekosysteemien näkökulmasta.

Lopulta näillä tiedoilla voidaan näyttää, miten funktionaalisen ohjelmoinnin käytänteet näkyvät hyödyllisesti (tai haitallisesti) ei-funktionaalisissa ohjelmistoprojekteissa.

Teoriaosuudessa pyritään pitämään sisältö mielenkiintoisena liimaamalla käsitteet yhdeksi kokonaisuudeksi. Teoriaan pyritään tuomaan konkretiaa erilaisilla esimerkeillä ja anekdooteilla.

## 3.1 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on ohjelmointiparadigma, jonka juuret ulottuvat 1930-luvulle ja lambda-kalkyylin kehitykseen. Lähestymistapa korostaa funktioiden käyttöä perusyksikköinä ja pyrkii välttämään muuttuvia tiloja (mutability) sekä sivuvaikutuksia (side-effects). Funktionaalisen ohjelmoinnin keskeisiä käsitteitä ovat puhtaat funktiot, korkeamman asteen funktiot, yhdistetyt funktiot sekä deklaratiivinen ohjelmointi. Funktiot ovat funktionaalisessa ohjelmoinnissa tosiaankin keskiössä. [8; 9.]

### 3.1.1 Työn nimeämiskäytänteet

Ohjelmoidessa funktionaalista ohjelmakoodia nimeämiskäytänteet ovat yhtä vaikeita kuin aina ennenkin. Tässä insinöörityössä käytetään Haskell-ohjelmoijien keskuudessa näkyviä nimeämiskäytänteitä, joissa käytetään lyhyitä ja ”intuitiivi-

sia” nimiä funktioille ja muuttujille. Esimerkiksi:

- x: Yleisesti käytetty muuttuja tai parametri, joka edustaa arvoa, jota funktiot käsittelevät. Jos näkyvyysalalla on muita muuttujia, niitä nimetään usein aakkosjärjestyksessä x:stä eteenpäin (x, y, z).
- xs: Muuttuja, joka edustaa useita arvoja. Usein sisältää listan arvoista, mutta tietorakenteen tietäminen ei ole merkittävää.
- f: Yksittäinen funktio, jota voidaan käyttää käsittelemään arvoja. Jos näkyvyysalalla on muita funktioita, niitä nimetään usein aakkosjärjestyksessä f:stä eteenpäin (f, g, h, i...).
- fs: Muuttuja, joka edustaa useita funktioita. Usein sisältää listan funktioista, mutta tietorakenteen tietäminen ei ole merkittävää.

Esimerkkejä käytänteiden käytöstä:

```
1 const map    = f => xs => xs.map(f)
2 const either = f => g => x => f(x) || g(x)
3 const max    = x => y => x > y ? x : y
4 const pipe   = fs => x => fs.reduce((acc, f) => f(acc), x)
```

Koodiesimerkki 2: Esimerkkejä insinöörityössä käytettävistä nimeämiskäytänteistä. Muun muassafunktio `map` ottaa ensin yhden funktion (f), ja tämän jälkeen monta arvoa (xs). `Pipe` ottaa ensin monta funktiota (fs), ja tämän jälkeen yhden arvon (x).

### 3.1.2 Puhtaat funktiot

Funktionaalisessa ohjelmoinnissa puhtaat funktiot ovat funktioita, jotka poikkeuksetta aina palauttavat samalle syötteelle saman arvon. Funktion puhtauden voi päätellä siitä, jos sen voisi teoreettisesti korvata hakutaulukolla. [31.]

Puhtaista funktioista hyötyy siten, että niitä voi käyttää uudelleen ja uudelleen kontekstista riippumatta. Puhtailla funktioilla voi rakentaa järjestelmäriippumattomia kirjastoja.

Puhtaiden funktioiden perusta on matematiikassa. Jos matemaattinen funktio $f(x) = x + 2$ esimerkiksi ei aina palauttaisi samalle $x$ arvolle samaa palautusarvoa, mitä hyötyä funktiosta edes olisi? Toivottavasti $2 + 2$ tulee aina olemaan $4$.

### 3.1.3 Kombinaattorit

Funktionaalinen ohjelmointi mahdollistaa koodin uudelleenkäytettävyyden. Lambda-kalkyylin perustein joka ikisien ongelman voi purkaa pieniksi funktioiksi [32]. Lambda-kalkyylistä johdetut kombinaattorit voi ottaa osaksi ohjelmoijan työkalupakkia missä tahansa ohjelmointikielessä, joka kohtelee funktioita samalla tavalla kuin muuttujia (first-class citizen) (koodiesimerkki 3).

Jos kombinaattoreille haluaa etsiä vastaavanlaista määritelmää olio-ohjelmoinnin termeistä, voisi niitä ajatella olevan perustavanlaatuisia suunnittelumalleja.

Kombinaattori `compose` on tapa tehdä yhdistettyjä funktioita, joista kerrotaan enemmän seuraavassa luvussa.

```
1  const identity      = x => x
2  const constant      = x => y => x
3  const apply         = f => x => f (x)
4  const thrush        = x => f => f (x)
5  const duplication   = f => x => f (x) (x)
6  const flip          = f => y => x => f (x) (y)
7  const compose       = f => g => x => f (g (x))
8  const substitution  = f => g => x => f (x) (g (x))
```

Koodiesimerkki 3: Yleiset kombinaattorit esitettynä JavaScriptissä [33]. Kombinaattoreilla voi esittää lambda-kalkyyliä. Kombinaattoreilla voi muokata funktioiden toimintaa, ja yhdistää funktioita yhdistetyiksi funktioiksi.

Kombinaattorit näyttävät uhkaavilta, ja raa'assa muodossaan niiden hyötyarvoa voi olla hankala havaita. Niiden käyttämistä voi kuitenkin perustella niiden perustavanlaatuisten ja kieliagnostisien ominaisuuksien takia. Ohjelmoidessa funktionaalisesti ei kombinaattoreita kuitenkaan tarvitse paljoa ajatella. Niitä tulee kirjoittamaan varmasti vähintään vahingossa, sillä niitä vaaditaan funktioita pyörittäessä [33]. Kuitenkin on hyvä tiedostaa käytössä olevat rakennuspalikat, ja millä nimellä etsiä niistä tietoa tarvittaessa.

### 3.1.4 Yhdistetyt funktiot ja niiden vahva merkitys

Usein funktionaalista ohjelmointia mainostaessa puhutaan funktioiden puhtauden ja datan muuttumattomuuden olevan paradigman oleellisinta. Kuitenkin samaan aikaan juuri datan muuttumattomuus ja tilattomuus koetaan paradigman suureksi kompastuskiveksi [6; 20; 34].

On tärkeää huomata, että kaiken ei tarvitse olla täysin puhdasta ja muuttumatonta, kunhan ne osat, joissa funktionaalista ohjelmointia hyödynnetään, pysyvät ennustettavina ja hallittavina.

Yhdistetyt funktiot ovat loistava tapa kirjoittaa selkeää ja modulaarista koodia. Parasta on, että kaiken ei tarvitse olla pelkkiä funktioita: voi hyvin kirjoittaa myös olio-ohjelmointityylillä ja silti hyödyntää yhdistettyjä funktioita logiikan selkeyttämiseksi ja yksinkertaistamiseksi.

Loppujen lopuksi funktionaalisen ohjelmoinnin perusta on funktioissa ja niiden yhdistämisessä. Datan muuttumattomuus ja sivuvaikutuksettomuus on vain esivaatimus funktioiden ennustettavuudelle.

Yhdistetty funktio tarkoittaa yksinkertaisesti kahden, tai useamman, funktion yhdistämistä siten, että yhden funktion tulos syötetään seuraavalle. Esimerkiksi koodiesimerkin 4 funktio $h$ on funktioiden $f$ ja $g$ yhdiste. Ajaessa funktio $h$, suoritetaan ensin $g$, jonka palautusarvo annetaan funktiolle $f$. Kaksi funktiota yhdistämällä on saatu yksi uusi funktio.

```
1 const f = x => 2 * x
2
3 const g = x => x + 3
4
5 const h = x => f(g(x))
```

Koodiesimerkki 4: JavaScript-esimerkki yhdistetystä funktiosta h ilman pipe- tai compose-funktiota

Funktionaalisissa ohjelmointikielissä yhdistettyjä funktioita pystyy usein kirjoittamaan käyttäen kieleen sisäänrakennettuja operaattoreja, joilla funktioiden yhdis-

täminen on helppoa ja suoraan osana ohjelmointikieltä [35; 36]. JavaScriptissä ei ole vastaavanlaista sisäänrakennettua operaattoria. (Vaikkakin sellainen on ollut kehitteillä [37].) Operaattorin voi kuitenkin kirjoittaa funktiona helposti osaksi mitä tahansa koodikantaa (koodiesimerkki 5).

```
1 const pipe    = f => g => x => g(f(x))
2 const compose = f => g => x => f(g(x))
3
4 const h = pipe(f)(g)
5 // tai vaihtoehtoisesti
6 const h = compose(g)(f)
```

Koodiesimerkki 5: JavaScript-esimerkki funktiokompositiosta pipe- ja compose-funktioilla.

Voi olla mielenkiinoista huomata, että koodiesimerkin `compose` löytyy myös koodiesimerkin 3 tunnetuista yleisistä kombinaattoreista.

Koodiesimerkissä 5 on näytettynä kaksi eri tapaa yhdistää funktioita: `pipe` ja `compose`. Käytännössä tapojen ainoa ero on se, kummasta suunnasta funktiot suoritetaan. `compose` on lähempänä yhdistettyjen funktioiden matemaattisia perusteita, kun taas `pipe` on usein helpompi lukea, sillä se suoritetaan yleisessä lukemissuunnassa, eli vasemmalta oikealle, tai ylhäältä alas. [38.]

Tulevissa koodiesimerkeissä tullaan suosimaan funktioiden yhdistämistä käyttäen `pipe`-funktiota `compose`-funktion sijasta.

Esimerkin 5 `pipe` ja `compose` toimivat vain kahdelle funktiolle kerrallaan. Tämä tapa seuraa lambda-kalkyyliä [9]. Ohjelmointikielen mukaan voi kuitenkin olla mieluisaa kirjoittaa funktiot niin, että ne tukevat suoraan mielivaltaista määrää funktioita.

JavaScriptissä näin voi tehdä esimerkiksi hyödyntämällä Array-tietorakennetta

(koodiesimerkki 6).

```
const pipe    = fs => x => fs.reduce((acc, f) => f(acc), x)
const compose = fs => x => fs.reduceRight((acc, f) => f(acc), x)

const f = x => x + 1
const g = x => x * 2
const h = x => x - 3

const i = pipe([f, g, h])
i(5) // (((5 + 1) * 2) - 3) = 9
// tai vaihtoehtoisesti
const i = compose([h, g, f])
i(5) // (((5 + 1) * 2) - 3) = 9
```

Koodiesimerkki 6: JavaScript-esimerkki yhdistettyjen funktioiden luomisesta käyttäen `reduce` ja `reduceRight` funktioita. Nämä versiot voivat ottaa mielivaltaisen määrän funktioita argumenttina.

Mikä tästä sitten tekee loistavaa? Se, että ohjelmointi on kerrankin oikeasti kuin LEGO-palikoilla leikkimistä (koodiesimerkki 7).

```
const pipe     = fs => x => fs.reduce((acc, func) => func(acc), x)
const multiply = x => y => x * y
const add      = x => y => x + y
const filter   = f => xs => xs.filter(f)
const map      = f => xs => xs.map(f)
const isEven   = x => x % 2 === 0

const rejectOdds   = filter(isEven)
const multiplyBy10 = multiply(10)

const pipeline = pipe([rejectOdds, map(multiplyBy10)])

pipeline([1, 2, 3, 4]) // [20, 40]
```

Koodiesimerkki 7: JavaScript-esimerkki yhdistettyjen funktioiden käyttämisestä laskutoimituksiin. Erilaiset operaatiot on pilkottu uudelleenkäytettäviksi funktioiksi.

Esimerkissä näytetyt funktiot ovat hyvin yksinkertaisia. Kuitenkin ne ovat täysin uudelleenkäytettäviä. Jos totuttautuu käyttämään joitain funktioita kaikissa projekteissa, voi huomata koodin kirjoittamisen tehokkuuden ja johdonmukaisuuden kasvavan.

## 3.2 Tyyppiteoria ja joukko-oppi

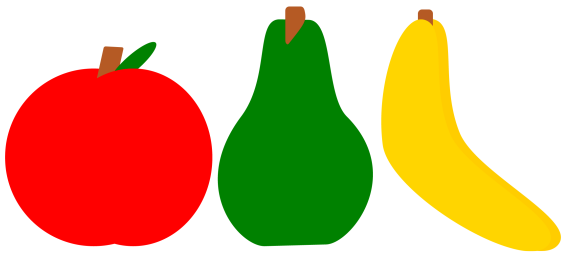
Tämä osa pyrkii kirjoittamaan matemaattisista käsitteistä käytännönläheisesti ja mahdollisesti suuresti yksinkertaistaen. Ajatuksena on, että ylemmän tason ajattelumalli välittyisi tekstissä tarkkojen määritelmien sijasta.

Joukko-oppi on matematiikan haara, joka tutkii joukkojen ominaisuuksia ja niiden välisiä suhteita. Ohjelmoinnissa joukko-oppi ilmenee yksinkertaisimmin joukkotietorakenteiden kautta, jotka mahdollistavat uniikkien alkioiden käsittelyn tehokkaasti ja ilmaisuvoimaisesti [24; 39]. On myös luotu ohjelmointikieli, joka perustuu kokonaan joukko-oppiin [40].

Tyyppiteoria puolestaan on matematiikan haara, joka tutkii arvojen luokittelua, ja miten niitä käsitellään ohjelmoinnissa ja logiikassa [41; 42].

Tyyppien ja joukkojen semanttinen ero on nyanssinen [43]. Ohjelmoinnissa puhutaan usein tyypeistä. Insinööriopintojen matematiikassa joukot ovat olleet vahvasti läsnä. Tässä luvussa pääajatuksena on miettiä, kuinka asiat muutetaan funktioilla toisiksi, asiat voi mieltää kuuluvan joukkoihin tai tyyppeihin, kumpaa nyt ymmärtää paremmin. Oikea määritelmä saattaa lähestyä kategoriateoriaa, mutta erottaminen ei liene olevan tällä tasolla relevanttia.

Ihmisillä on kuitenkin kyky hahmottaa asioita joukkoina (tai tyyppeinä). Koko maailman voi mallintaa niillä (kuva 2) [42].



Kuva 2: Mihin joukkoon tai tyyppiin nämä saattaisivat kuulua [44]?

Koska koko maailman voi mallintaa joukkoina (tai tyyppeinä), voidaan niillä mallintaa myös ohjelmia.

### 3.2.1 Joukkotietorakenne

Ohjelmoinnissa joukkotietorakennetta käytetään hyvin laajasti, ja sen merkitys on keskeinen monissa ohjelmointikielissä, kuten Pythonissa, Javassa ja JavaScriptissä. Joukkotietorakenne mahdollistaa alkioiden tallentamisen ilman duplikaatteja. Sen operaatiot, kuten unioni, leikkaus ja erotus, ovat suoraan peräisin matemaattisesta joukko-opista. Joukkotietorakenteen tehokkuus ja yksinkertaisuus tekevät siitä tehokkaan työkalun monenlaisissa sovelluksissa. [39; 45.]

Joukkotietorakenne kuvaa joukko-opin joukkoa. Joukkotietorakenteessa säilytetään kokoelma uniikkeja alkioita. Joukkotietorakenne on ohjelmoinnissa mieluinen työkalu sen tehokkuuden ansiosta. [45.]

Kokemusten mukaan ohjelmoija ei usein mieti joukkotietorakennetta matemaattisista lähtökohdista. Tästä kuitenkin voisi olla hyötyä, sillä joukko-opin periaatteiden ymmärtäminen ja soveltaminen ohjelmoinnissa ei ainoastaan tehostuskeino, vaan myös laajentaa näkemystä siitä, miten abstraktit matemaattiset konseptit voidaan muuttaa konkreettisiksi työkaluiksi. Tämä yhdistelmä tarjoaa ohjelmoijille sekä teoreettisen että käytännöllisen perustan, joka on arvokas monissa erilaisissa sovelluksissa ja ongelmanratkaisutilanteissa [46].

### 3.2.2 Funktioilla joukoista toisiin

Kun ohjelmoinnissa pohditaan joukko-opin hyödyntämistä, kehittyy kieliriippumaton kyky kirjoittaa koodia. Kun miettii, mitä ylipäätään voi laskea, pääsee ohjelmoinnissa omalle abstraktille tasolle. [8; 32.]

Matemaattisesti joukkojen välille voidaan tehdä kuvauksia, tai toisin sanoen funktioita. Merkintä funktiosta, joka ”muuttaa” joukon $A$ joukoksi $B$, on esimerkiksi yksinkertaisesti $A \rightarrow B$ [47]. Funktio muuttaa minkä tahansa joukon $A$ alkion joksi-

kin joukon $B$ alkioksi.

Funktionaalisessa ohjelmoinnissa puhtaiden funktioiden voidaan ajatella aina olevan rinnastettaessa jonkin joukon muutokseksi johonkin toiseen joukkoon (koodiesimerkki 8) [46]. (Toisaalta funktioiden argumentit ja palautusarvot ovat aina joukkoja riippumatta siitä onko funktio puhdas vai ei. Ei-puhtaat funktiot eivät kuitenkaan välttämättä luo yksikäsitteisiä muutoksia joukkojen alkioille.)

```
type SetA = 'a' | 'b' | 'c'
type SetB = 1 | 2 | 3
const aToB = (a: SetA): SetB => {
    switch (a) {
        case 'a': return 1
        case 'b': return 2
        case 'c': return 3
    }
}
```

Koodiesimerkki 8: Havainnollistava funktio, joka muuttaa joukon A, {a,b,c}, joukoksi B, {1, 2, 3}.

Tarkastellaan konkreettisempaa esimerkkiä lähtien koodiesimerkistä 9.

```
const divide = (x:number) => (y:number) => x / y
```

Koodiesimerkki 9: Funktio, joka jakaa numeron toisella.

Funktio ottaa kaksi numeroa, ja jakaa ensimmäisen toisella. Argumentit on tyypitetty JavaScript-numeroiksi. Jos kyseistä funktiota haluaa käyttää, ohjelmoijan tulisi tietää, että nollalla ei voi jakaa ja että jos niin kuitenkin tehdään, JavaScript palauttaa arvon `Infinity` (tai erityisemmässä tapauksessa `NaN`) (koodiesimerkki 10).

```
console.log(1/0)  // Infinity
console.log(1/-0) // -Infinity
console.log(0/0)  // NaN
```

Koodiesimerkki 10: Jakamisoperaattorin toiminta JavaScriptissä

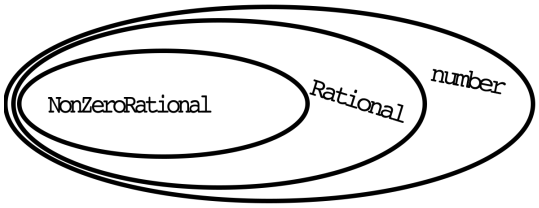
Jos jakolaskufunktion kehitykseen halutaan soveltaa joukko-oppia, voi olla mieluista määritellä argumenteille spesifimmät joukot, sillä joukkona JavaScriptin `number` sisältää epämieluisia arvoja laskujen kannalta (`[NaN, Infinity, -Infinity]`).

Voidaan luoda jakolaskua varten joukot `Rational` ja `NonZero`, ja ohjelmoida ne TypeScript-tyypeiksi (koodiesimerkki 11).

```
1  const RationalSymbol = Symbol('Rational')
2  const NonZeroSymbol = Symbol('NonZero')
3
4  type Rational = number & { [RationalSymbol]: "brand" }
5  type NonZero = Rational & { [NonZeroSymbol]: "brand" }
6
7  const isRational = (x: number): x is Rational =>
8      ![Infinity, -Infinity, NaN].includes(x)
9  const isNonZero = (x: number): x is NonZero =>
10     isRational(x) && x !== 0
```

Koodiesimerkki 11: Rational- ja NonZero-joukkojen määritelmät TypeScriptissä

Ajatuksena on, että joukko `Rational` sisältää kaikki JavaScriptin tukemat rationaaliluvut, ja että joukko `NonZero` muuten sama paitsi ilman nollaa (kuva 3).




Kuva 3: Joukot NonZero, Rational ja number esitettynä osajoukkoina

Jos jakamisesimerkkiin määritetään joukot funktion argumenttien oikeiksi rajoitteiksi, ei funktiota voi käyttää väärin. TypeScript ei anna ajaa koodia, jos arvojen ei ole varmistettu kuuluvan oikeisiin joukkoihin (koodiesimerkki 12) [1].

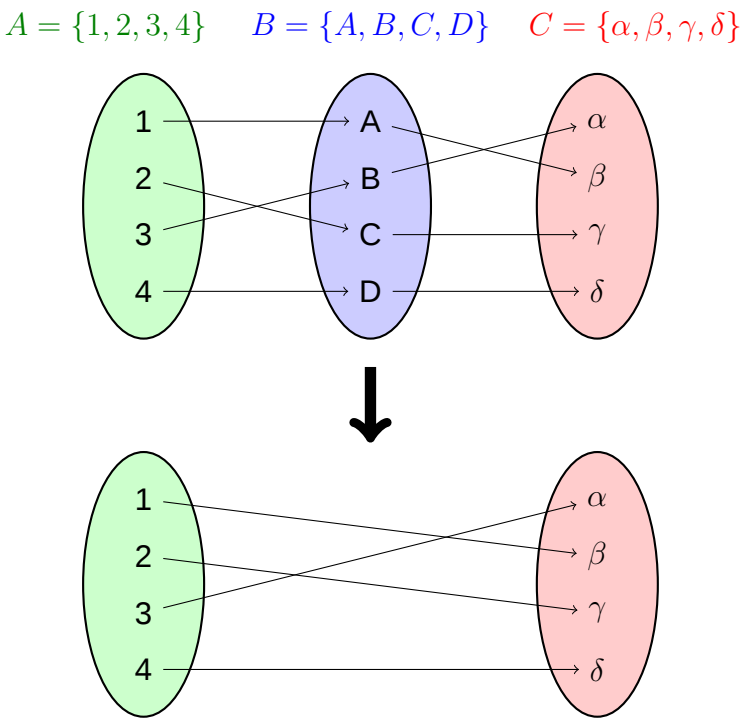
```
1  const divide = (x: Rational) => (y: NonZero) => x / y as Rational
```

Koodiesimerkki 12: Korrekti versio. Funktiolle annettavat arvot on varmistettava kuuluvan oikeisiin joukkoihin jossain vaiheessa ennen funktioon syöttöä käyttäen määriteltyjä `isRational`- ja `isNonZero`-funktioita.

Aiempaan esimerkkiin voisi päätyä, vaikkei olisi koskaan kuullut joukko-opista. Päällimmäisenä ajatuksena on kuitenkin se, että jos syötteitä ja palautusarvoja miettii joukkojen kannalta, syntyy luotettavampaa ohjelmakoodia ja vähemmän yllätyksiä. Tyypit ovat jo itsessään joukkoja, mutta niiden yksityiskohdat riippuvat

käytössä olevasta, esimerkiksi ohjelmointikielen, tyyppijärjestelmästä [41]. Siksi ajatuksen tasolla joukkojen ajattelu on hyödyllistä.

Argumentit ja palautusarvot (tyypit) voi tietysti mieltää joukkoina myös yhdistetyissä funktioissa (kuva 4). Vaikka ohjelmoinnissa joukko-oppia ei rinnasteta ainoastaan funktionaaliseen ohjelmointiin, joukko-opin ajattelu on siinä vähintäänkin tervetullutta.




Kuva 4: Funktioiden yhdistäminen joukko-opin näkökulmasta. Funktiot A $\rightarrow$ B ja B $\rightarrow$ C yhdistämällä saadaan A $\rightarrow$ C. Älykkäällä kääntäjällä voi poistaa turhia laskutoimituksia. Keskimmäisen joukon evaluointi on käytännössä turhaa, jos funktiot ovat puhtaita.

### 3.2.3 Kategoriateoria

Joukko-oppi ei ole ainoa ohjelmointiin sovellettava matemaattinen osa-alue. Funktionaaliselle ohjelmoinnille oleellisin ja syvällisin osa-alue on kategoriateoria [4; 5; 46].

Kategoriateoria tutkii matemaattisia rakenteita ja niiden välisiä suhteita erittäin abstraktilla tasolla. Se keskittyy objektien (kuten joukkojen tai tyyppien) ja niiden vä-

lillä olevien morfismien (funktioiden tai muunnosten) tutkimiseen [46]. Kategoriateorian avulla voidaan ymmärtää ja formalisoida monimutkaisia matemaattisia ja ohjelmallisia rakenteita. [4; 46; 48.] Kategoriateoriaa ohjelmoijille opettavan Bartosz Milewskin kokemusten mukaan, vaikka algebra tai kalkyyli olisi ohjelmoijalle ajatuksena hirveää, kategoriateoria on kuitenkin erityisen mieluisaa sen suuresta teoreettisuudesta huolimatta. [49, s. 9.].

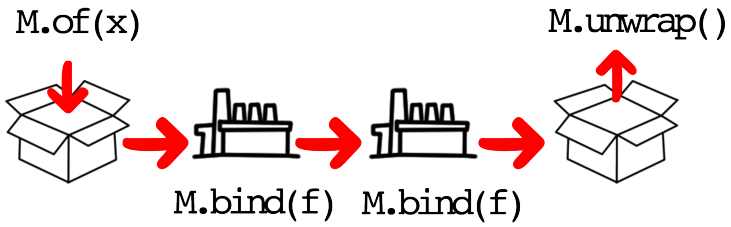
Kategoriateoriassa peruspalikat ovat kategoriat [49, s. 9]. Kategoria on rakenne, joka koostuu morfismeista. Esimerkiksi joukot ja funktiot koostavat kategorian, jossa joukot ovat kategorian objektit, ja funktiot kategorian morfismit [48]. Kategoriateoriaan ei perehdytä tarkemmin tässä työssä monadirakennetta lukuun ottamatta. Ei ole käytännönläheistä tuoda tavattoman teoreettisia käsitteitä työhön, jossa tavoitteena on etsiä jonkinkaltaista konkretiaa.

## 3.3 Monadi

Monadi tulee esille funktionaalisessa ohjelmoinnissa erittäin usein. Se on alun perin teoreettinen rakenne, joka on löytynyt 1900-luvun puolivälin jälkeen kategoriateoriaa tutkivien matemaatikkojen ansiosta [50]. Monadin hyöty ohjelmoinnissa ilmeni vasta myöhemmin 1990-luvulla [51].

Ohjelmoinnissa ohjelman sivuvaikutukset ovat miltei vaatimus. Funktionaalisessa ohjelmoinnissa sivuvaikutukset eivät sovi. Monadeilla onnistuttiin tuomaan sivuvaikutuksia funktionaaliseen ohjelmointiin säilyttäen ohjelmien ennustettavuus ja puhtaus mutta kuitenkin siten, että koodi pysyy ulkoisesti sivuvaikutuksettomalta vaikuttavana. [51.]

Ohjelmoinnissa monadeilla voidaan hallita sivuvaikutuksia, ketjuttaa operaatioita ja yhtenäistää abstraktioita [2; 51; 52]. Ohjelmoinnissa monadi on kieliagnostinen käsite. Monadin voi kuvitella olevan lähetys, jota kuljetetaan tehtaasta toiseen. Jokainen tehdas aina pitää huolen, että arvo on kääritty johonkin laatikkoon (monadiin) (kuva 5).




Kuva 5: Monadisten operaation ketjuttaminen kuvattuna laatikoin ja tehtain. Laatikko (monadi) kuljetetaan (bind) tehtaiden (function) läpi. Laatikon sisälle ei nähdä suoraan, ennen kuin se avataan.

Tässä luvussa käydään läpi, mitä ovat yleiset monadit, miten implementoida monadi, miten niitä käytetään, ja lopulta pyritään tiivistämään miten monadeja voi ajatella ohjelmoinnissa.

### 3.3.1 Yleiset monadit

`Promise`, `Maybe`, `Either`, `Result`, `State`, `IO`, ja `Lista (Array)`. Nämä ovat esimerkkejä yleisesti käytetyistä monadeista. [52; 53.]

Pintapuolisesti tarkasteltuna esitetyt käsitteet vaikuttavat hyvin erilaisilta, sillä niiden käyttötarkoitukset ja rakenteet vaihtelevat merkittävästi. Esimerkiksi `Promise` käsittelee asynkronisia operaatioita, kun taas `Maybe` kuvastaa mahdollisia arvoja, jotka voivat olla olemassa tai olla olematta. `Either` ja `Result` tarjoavat mekanismin virheiden käsittelyyn, ja `State` mahdollistaa tila-arvojen kantamisen funktioiden läpi. `IO` käsittelee sivuvaikutuksia, ja `Lista (Array)` tarjoaa toistuvien arvojen käsittelyn.

Kuitenkin näitä kaikkia yhdistää yhteinen abstraktio. Ne kaikki toteuttavat tietyt perustoiminnot `bind` (`flatMap`/`chain`) ja `of` (`unit`/`pure`). Näiden operaatioiden avulla monadit voivat kapseloida laskennan tilan tai sivuvaikutuksen, mikä tarjoaa yhdenmukaisen tavan yhdistää operaatioita ja siirtää tietoa laskentaketjujen läpi.

JavaScriptissä `Array` toteuttaa toiminnon `bind` `Array.prototype.flatMap`-metodilla, ja `of` toiminnon staattisella `Array.of`-metodilla [54; 55].

JavaScriptin `Promise` toteuttaa toiminnon `bind` `Promise.prototype.then`-metodilla, ja `of`-toiminnon staattisella `Promise.resolve` ja `Promise.reject` metodeilla [3; 4; 56].

Vaikka JavaScriptin `Promise` ja `Array` toteuttavatkin monadin `bind`- ja `of`-toiminnot, se ei tee niistä automaattisesti monadeja kategoriateorian tai funktionaalisen ohjelmoinnin sääntöjen mukaan [3; 4; 55]. Kehittäjä Jake Archibaldin mukaan akateemisten käsitteiden seuraaminen puhtaasti ei kuitenkaan usein johda käytännöllisiin ratkaisuihin [57]. Tästä syystä on kohtuullista puhua näistä rakenteista monadeina, vaikkeivat ne sitä puhtaasti olisikaan.

### 3.3.2 Monadin implementointi

Konkreettisesti monadilla tulee siis olla kaksi toimintoa: `of` ja `bind`. Toiminto `of` käärii arvoja monadiin. Toiminnon voidaan ajatella olevan monadin konstruktori. [2.]

`Bind`-toiminnolla muutetaan monadiin käärittyä arvoa antamalla toiminnolle funktio, joka palauttaa toisen saman tyypin monadin [2]. `Bind`-toiminto on käytännössä sama kuin tunnetumpi `map`-toiminto. Erona on, että `map` palauttaa muokatun arvon, joka on käärityn samaan monadiin, kun taas `bind` mahdollistaa monadin sisällä olevan arvon muuttamisen ja sen ketjuttamisen muihin monadisiin operaatioihin.

Näillä kahdella toiminnolla voidaan luoda rakenteita, jotka käyttäytyvät kuin monadi. Monadin täydet hyödyt eivät kuitenkaan ilmene, jos rakenne ei myös täytä monadien kolmea sääntöä: vasen identiteetti (left identity), oikea identiteetti (right identity), ja liitännäisyys (associativity) [58]. Jos kaikki kolme sääntöä toteutuu, voidaan luoda työkaluja, joissa voidaan käyttää mitä tahansa monadia, ja silloin monadi on määritelmän mukaisesti monadi.

Luvussa ”Promise-tietorakenne on miltei monadi” puidaan sitä, miksei JavaScriptin `Promise` ole määritelmällisesti täydellisesti monadi ja miksei ilman ulkoisia kirjastoja JavaScriptiin voida enää tuoda määritelmän täyttäviä monadeja. Kuitenkin

rakenne, joka on kuin monadi, voi myös olla erittäin hyödyllinen. Siksi ei tarkemmin käsitellä, miten monadin määritelmän voi oikeellisesti täyttää.

### 3.3.3 Monadin käyttäminen

Monadeja käytetään operaatioiden ketjuttamiseen. Monadeilla voi käytännössä tehdä yhdistettyjä funktioita, joissa on mukautettu tapa hallita, mitä yhdisteen funktiokutsujen välissä tapahtuu.

Käydään esimerkkinä JavaScriptin `Promise`-tietorakennetta (joka toimii kuin monadi) (koodiesimerkki 13).

```
1 fetch("https://some.imaginary.api.com/v1")
2     .then(result => result.json())
3     .then(data => data.users)
4     .then(users => users.map(user => user.id))
5     .catch(error => Array.of())
```

Koodiesimerkki 13: Promise-tietorakenteella bind-operaatioiden ketjutus, jossa haetaan käyttäjätietoja kuvitteellisesta ulkoisesta rajapinnasta. Jos mikään yksittäinen askel päätyy virhetilaan, palauttaa ohjelma tyhjän listan kutsumalla `Array.of`-metodia.

Koodiesimerkissä nähdään operaatioiden monadinen ketjutus. Jokainen `Promise.then`-kutsu palauttaa aina uuden Promise-monadin. Jos jokin yksittäinen Promise päätyy virhetilaan, ei enää seuraavia then-kutsuja ajeta, vaan ketju päätyy automaattisesti `catch`-osioon.

Käytännössä koko ohjelma on siis vain yhdistetty funktio, jossa on määritelty räätälöityä logiikkaa siihen, miten funktiokutsut käsitellään. Kuitenkaan `Promise.catch` ei liity suoraan monadisuuteen, eikä käytännössä oikeellisesti implementoidun monadin tarvitsisi käyttää sellaista [4].

Raja on häilyvä. Onko JavaScriptin Promise monadi vai ei? Matemaatikkojen mielestä monadi on ”yksinkertaisesti” monoidi endofunktoreiden kategoriassa [2; 52; 53]. Toisten mielestä monadi on asia, jolla on `flatMap`-metodi [54].

Pragmaattisuuden nimissä monadi saa tässä työssä olla rakenne, jolla voidaan

ketjuttaa operaatioita itsemäärittelemällä tavalla, joka mahdollisesti eroaa tavallisesta funktioiden yhdistämisestä.

## 3.4 Pragmaattisuus funktionaalisessa ohjelmoinnissa

Funktionaalisen ohjelmoinnin iskevin teoria on pitkälti käytynä, ja on ehkä selvää, että funktionaalinen ohjelmointi on pohjimmiltaan erittäin teoreettista. Voi pohtia, miten funktionaalinen ohjelmoinnin voi tuoda ohjelmistokehittäjän arsenaaliin käytännönläheisesti.

Kun funktionaalisen ohjelmoinnin kauneus on matemaattisissa kulmakivissä, joilla ei ole tapaa taipua, miten sen voi yhdistää säännöttömään JavaScriptiin, tai muuhun vastaavanlaiseen ohjelmointikieleen?

Mikä on se lähestymistapa, jota tarvitaan käytännönläheiseen, pragmaattiseen, funktionaaliseen ohjelmointiin?

Robert C. Martinin mukaan pragmaattinen funktionaalinen ohjelmointi on sitä, että käytetään esimerkiksi muuttumattomuutta ja puhtaita funktioita silloin, kun ne parantavat ohjelman luettavuutta, ylläpidettävyyttä ja suorituskykyä, ja kuitenkin niin, ettei vaadita täydellistä sitoutumista funktionaalisen ohjelmoinnin paradigmaan. [12.]

Myös Cantarella mainitsee opinnäytetyönsä loppupuolella ajatuksen, että mitä enemmän paradigmoihin liittyviin artikkeleihin perehtyy, sitä myötämielisemmäksi tulee ajatukselle, että paradigmoja tulisi yhdistellä vapaammin. [6, s. 45.]

Myös kokemusten perusteella työpaikalla on havaittu funktionaalisen ohjelmoinnin toimivan kohtuullisen hyvin ripoteltuna proseduraalisen ohjelmakoodin.

Pragmaattista funktionaalista ohjelmointia vaikuttaa siis olevan se, että otetaan vain, mitä tarvitaan [5; 6; 12]. Toisaalta se, mitä tarvitaan ei ole välittömästi selvää. Toisille pragmaattista on jättää kaikki matemaattinen teoria pois ohjelmoinnista heti kättelyssä, toisille kategoriateorian rakenteiden käyttäminen, kun ne oi-

keasti sopivat mallintamaan ongelmaa [12; 59].

Tässä insinöörityössä tarkempia pragmaattisia funktionaalisen ohjelmoinnin käytäntöjä pyritään tuomaan esille neljännessä luvussa ”Funktionaalisia käytänteitä ei-funktionaalisissa ohjelmointikielissä”, ja siitä eteenpäin.

## 3.5 Funktionaalinen ohjelmointi & TypeScript

JavaScript ei ole funktionaalinen ohjelmointikieli, vaikka se funktionaalisuutta tukeekin [34]. TypeScriptin tyyppijärjestelmällä funktionaalisesta ohjelmoinnista saa jo mieluisampaa [59].

JavaScriptia halutaan käyttää kaikkeen ohjelmointiin, vaikka usein erinäköisiin tehtäviin olisi parempiakin ohjelmointikieliä tarjolla. Voi itse päättää, onko pragmaattista käyttää kaikkeen samaa työkalua, ja säästää opiskelussa, tai käyttää aikaa uuden sopivamman työkalun löytämiseen, ja säästää toteuttamisessa.

### 3.5.1 Kielen sopivuus funktionaaliseen ohjelmointiin

Vaikka TypeScript ei ole pohjimmiltaan funktionaalinen ohjelmointikieli, on sillä ominaisuuksia, joilla funktionaalisesta ohjelmoinnista tulee verraten helppoa.

TypeScriptissä on käytännössä kolme eri tapaa määrittää funktioita (koodiesimerkki 14). Nuolifunktioilla voidaan harjoittaa funktionaaliselle ohjelmoinnille tyypillistä argumenttien yksi kerrallaan antamista.

```
1 function rnd(min: number, max: number) {
2   return Math.floor(Math.random() * max) + min;
3 }
4 const rnd2 = (min, max) => Math.floor(Math.random() * max) + min;
5 const rnd3 = min => max => Math.floor(Math.random() * max) + min;
```

Koodiesimerkki 14: Kolme eri tapaa kirjoittaa funktio JavaScriptissä [60]. Funktiomäärittely, funktioilmaus ja osittain sovellettava funktioilmaus.

Koska funktioita voi käyttää JavaScriptissä kuin muuttujia (first-class citizen), voi niitä antaa argumentteina toisille funktioille (koodiesimerkki 15). Tämä mahdol-

listaa funktionaaliselle ohjelmoinnille tyypillisten korkeamman asteen funktioiden luomisen, ja käyttämisen.

```
1 function A()  { return "moi"; }
2 function B(f) { return f(); }
3 B(A) // "moi"
```

Koodiesimerkki 15: Funktioiden ensiluokkaisuus JavaScriptissä. Funktiolle B voidaan antaa funktio argumenttina.

Funktionaalisen ohjelmoinnin piireissä TypeScriptin tyyppijärjestelmää arvostetaan myös, koska sillä voidaan rakentaa algebrallisia tietotyyppejä, jotka ovat yleisesti puhtaissa funktionaalisissa ohjelmointikielissä läsnä ja suuressa roolissa [59].

### 3.5.2 Algebralliset tietotyypit

TypeScriptin tyypit ovat mieluisia funktionaaliselle ohjelmoinnille. TypeScript mahdollistaa funktionaaliselle ohjelmoinnille olennaisten algebrallisten tietotyyppien määrittelemistä [59].

Algebralliset tietotyypit ovat osa funktionaalisen ohjelmoinnin maailmaa. Niiden käyttämisen on huomattu olevan järkevää ja luontevaa [42; 59; 61].

Yksinkertaisesti algebralliset tietotyypit ovat keino määritellä ja käsitellä monimutkaisia datastruktuureja, jotka koostuvat perustietotyypeistä yhdistämällä niitä eri tavoin. Tyypillisesti algebralliset tietotyypit jaetaan kahteen päätyyppiin: tulo- ja summatyypit.

**Tulotyyppi** (product type) kuvaa tietorakenteita, joissa yhdistetään useita arvoja yhdeksi kokonaisuudeksi, esimerkiksi luokkien tai tietueiden (records) avulla. TypeScriptissä tulotyyppejä voi määritellä käyttäen objekteja tai Record-tyyppejä (koodiesimerkki 16). [42; 59.]

```
1 // Tyypin mahdollisten erilaisten alkioiden määrä:
      värimahdollisuudet * istuinmahdollisuudet -> tulotyyppi
2 type Car = {type: "car", color: string; seats: number;}
```

Koodiesimerkki 16: Tulotyyppi-esimerkki TypeScriptissä. Tulotyyppi tulee siitä, että tyypin osien määrän voi kertoa keskenään saadakseen tyypin kokonaisen permutaatioiden määrän.

**Summatyyppi** (sum type) mahdollistaa arvon, joka voi olla yksi useista ennalta määritellyistä tyypeistä. TypeScriptissä summatyyppejä voi määritellä käyttäen diskriminoituja unioneja (discriminated union) (koodiesimerkki 17). [42; 59.]

```
1 type Car = {type: "car", color: string; seats: number;}
2 type MotorBike = {type: "motorbike", top_speed: number;
      wheelieability:number;}
3 // Tyypin mahdollisten erilaisten alkioiden määrä: autojen määrä +
      moottoripyörien määrä -> summatyyppi
4 type Vehicle = Car | MotorBike
```

Koodiesimerkki 17: Summatyyppi-esimerkki TypeScriptissä. Summatyyppi tulee siitä, että tyypin osien määrän voi summata keskenään saadakseen tyypin kokonaisen permutaatioiden määrän.

Vaikka TypeScriptin tyyppijärjestelmä on vankkarakenteinen, ei se varmista kaikista monimutkaisimpien tyyppien oikeellisuutta. Sellaisiin tarvitaan jokin pohjimmiltaan funktionaalinen ohjelmointikieli, kuten Haskell. [59.]

### 3.5.3 Kirjastot

Funktionaalista ohjelmointia on pyritty pitämään elossa erinäköisin kirjastoin, joissa hyväksikäytetään kielen ominaisuuksia ja luodaan kirjastoja, joiden käyttäminen vaatii ohjelmoijalta täysin uuden syntaksin opettelemista JavaScript syntaksin lisäksi [59; 62–65].

Kokemusten perusteella syntaksierot ovat todella hidastava tekijä. Ohjelmoijat olettavat ohjelmoivansa TypeScript-projektissa TypeScriptiä. Tai yleisemmin sanottuna: ohjelmoijat olettavat ohjelmoivansa sitä ohjelmointikieltä, mitä ohjelmoivat.

Funktionaalisen ohjelmoinnin kirjastoilla halutaan tuoda TypeScript-ekosysteemiin sitä, mitä siitä puuttuu. Kuitenkin on kirjasto kuinka hyvälaatuinen tahansa, tulee TypeScriptin tyyppijärjestelmän rajoitteet kuitenkin lopulta vastaan [59].

### 3.5.4 Promise-tietorakenne on miltei monadi

JavaScriptissä oleva Promise-tietorakenne toimii miltei kuin monadi, muttei ole sitä määritelmällisesti matematiikan kategoriateorian mukaan [3; 4]. Tässä luvussa puidaan, miten tämä pääsi tapahtumaan ja mitä merkitystä sillä edes on, että JavaScriptissä ei ole standardoitua tapaa luoda monadeja.

JavaScriptin kehitys perustuu vahvasti takaperin yhteensopivuuteen, mikä estää merkittäviä muutoksia kielen rakenteisiin ilman riskiä rikkoa olemassa olevia verkkosivuja [66]. Selaimet eivät voi myöskään tehdä omin päin rikkovia muutoksia [67; 68]. Jos vierailee lääkärin vastaanoton verkkosivulla selvittääkseen aukioloajat, kumpi selain on paras: se, joka näyttää vastaanoton aukioloajat, vai se, joka ei näytä mitään, koska on ottanut uuden JavaScript-version käyttöön?

Kuitenkin JavaScript kehittyy. Vuonna 2015 Promise-tietorakenne lisättiin JavaScriptiin ratkaisemaan callback-funktioiden hallinnan ongelmia asynkronisessa ohjelmoinnissa [69; 70]. Promise-tietorakenteen juuret ovat funktionaalisessa ohjelmoinnissa. Promise-tietorakenne on rakenteeltaan sellainen, että sen voi kuvata monadina [3; 4].

Kun Promise-tietorakennetta oltiin tuomassa osaksi JavaScriptia, keskusteltiin, tulisiko JavaScriptin Promise-tietorakenteen noudattaa monadirakennetta [4]. Keskustelu jäi tuloksettomaksi, vaikka muutokset implementointiehdotukseen olisivat olleet pienet monadin sääntöjen pitämiseksi.

Yksityiskohdat eivät liene merkittäviä, mutta lopputulos on se, ettei JavaScriptin Promise-tietorakennetta saatu täyttämään kaikkia monadien sääntöjä.

Mahdollisuus tuoda monadit osaksi JavaScript standardia on todennäköisesti menetetty [66; 68]. Ohjelmoijat joutuvat itse valitsemaan, miten luoda ja käyttää monadeja JavaScriptissä. Jos sekä Array että Promise olisivat yhtenäisesti monadeja, niiden yhteiskäyttö olisi suoraviivaisempaa, ja uusien monadien implementointi olisi johdonmukaisempaa suoraan JavaScriptissä kielen tasolla. Nykytilanne pakottaa ohjelmoijat improvisoimaan.

# 4 Funktionaalisia käytänteitä ei-funktionaalisissa ohjelmointikielissä

Tässä luvussa pyritään esittämään kaikista oleellisin, mitä funktionaalisella ohjelmoinnilla voidaan saada aikaan. Teoreettisen perustan ja kokemusten avulla tuodaan esille, mitä funktionaalinen ohjelmointi on koetusti parhaimmillaan.

Ajatuksena on, että kun ohjelma on mallinnettu oikein, ei ole suurta merkitystä, ohjelmoidaanko olio-ohjelmointia, imperatiivista ohjelmointia, vai funktionaalista ohjelmointia. Olio-ohjelmoinnissa mallinnus saatettaisiin hoitaa rajapinnoilla (interface). Funktionaalisen ohjelmoinnin tapa mallintaa on käyttää tyyppejä ja abstrakteja (algebrallisia) tietorakenteita.

Tuodaan ilmi, millaisia käytänteitä voidaan ripotella ohjelmointiin, vaikka ohjelmointikieli ei varsinaisesti olisi funktionaalinen.

Pyritään herättämään ajatuksia, milloin ja miten funktioita kannattaa käyttää. Esitetään, kuinka suuressa roolissa ongelmien mallintaminen on. Kun ongelmien mallit pureskellaan oikeisiin joukkoihin, saadaan turvallisempaa ohjelmakoodia. Silloin on myös helpompi kirjoittaa deklaratiivista koodia, joka on funktionaaliselle ohjelmoinnille ominaista.

## 4.1 Opi kerran, käytä kaikkialla

Funktionaalinen ohjelmointi kannustaa asioiden uudelleenkäytettävyyteen. Funktionaalisessa ohjelmoinnissa etsitään yleisiä teemoja ja viedään toistuvia malleja funktioiksi, jotta voidaan kirjoittaa funktioiden nimiä syntaksin sijasta.

Esimerkiksi olio-ohjelmoinnissa, tai muuten imperatiivisessa ohjelmoinnissa, for-silmukalla toteutettavat algoritmit voidaan yleistää funktioiksi (kuva 6). Onko helpompaa opetella ääretön määrä funktioita, vai yksi kielirakenne: for-silmukka?

Nimet ovat syntaksiriippumattomia. Samoja nimiä voi käyttää ohjelmointikielestä

$$\left.\begin{array}{l} filter, without, find, findIndex, findLast, findLastIndex \\ map, mapIndexed, mapIndexedRight, chain, concat, zip \\ reduce, reduceRight, scan, partition, uniq, forEach \\ slice, drop, take, dropWhile, takeWhile, zip, zipWith \\ all, any, none \end{array}\right] \text{ for loop}$$

Kuva 6: Ramda.js-kirjaston funktioita listojen käsittelyyn ohjelmoinnissa [71]. Kaikki nämä voitaisiin korvata for-silmukoilla imperatiivisessa paradigmassa.

riippumatta. Nimillä voi kuvata oikean maailman asioita. Siksi voi perustella, että kun tietyn logiikan aina laittaa tietyn nimen taakse funktioksi, ymmärtää sen aina vaikka ohjelmointikieli vaihtuisi. Jos muistaa idean, ei implementaation yksityiskohdilla ole merkitystä.

Jos asia on tarpeeksi monimutkainen ja sen piilottaa nimen taakse, hyötyarvon määrittäminen voi olla haastavaa. Mitä monimutkaisempaa asiaa yritetään enkoodata yksittäiseen nimeen, sitä vaikeampaa se on lukijan ymmärtää, ja myös sitä vaikeampaa sille on löytää uudelleenkäyttötilanteita. Pienikin nyanssiero funktion implementaatiossa totuttuun implementaatioon voi mitätöidä opitun hyödyn eri ympäristöissä.

Yksinkertaisiakin funktioita voi olla vaikea hahmottaa koodista. Mikään ei pakota ohjelmoijaa nimeämään funktiota samalla tavalla kuin miten sama funktio on nimetty jossain toisessa projektissa tai ohjelmointikielessä. Nimeämiskäytänteiden noudattaminen ei kuitenkaan ole helppoa. Käytänteitä voi olla vaikea löytää, tai niitä voi olla useita [71].

Kannattanee käyttää ohjelmointikielen sisäänrakennettuja funktioita, ja metodeja aina kun mahdollista. Näin voi maksimoida sen, että koodi ymmärretään. Jos kyse on jostakin projektin sisäisestä kirjastosta, voi olla hyödyllistä lisätä funktiolle kommentti, jos se tunnetaan muissa ohjelmointiympäristöissä jollain toisella nimellä.

Jos on kyse jostain toimintaympäristökohtaisesta funktiosta, eikä perustavanlaatuisesta yleispätevästä funktiosta, on tärkeää, että funktio nimetään mahdollisim-

man kuvaavasti. Esimerkiksi sen sijaan, että nimeää funktion nimellä ”processUsers”, voi miettiä, tarvitseeko funktion kutsupaikoissa ymmärtää enemmän siitä, mitä funktio tekee. Jos kontekstista on selvää, mitä funktio tekee, nimen voi jättää sellaiseksi kuin on. Jos konteksti ei avaa toimintaa enemmän, ”processUsers” on erittäin ympäripyöreä nimi.

## 4.2 Yhdistetyt funktiot

Kun funktioita on saatu kasattua työkalupakkiin, niitä voi yhdistellä erilaisten tarpeiden mukaan. Kerran kirjoitetut funktiot ovat monikäyttöisiä ja niitä voidaan ketjuttaa keskenään, jolloin monimutkaisemmat operaatiot rakentuvat valikoiduista palikoista.

Yhdistettyjen funktioiden avulla voidaan saavuttaa korkea taso ohjelmoinnin joustavuudessa. Ad hoc -funktioiden, eli tilanteeseen nopeasti mukautettujen funktioiden, kirjoittaminen on vaivatonta, ja niiden lisääminen osaksi olemassa olevia ketjuja onnistuu helposti. Tämä mahdollistaa nopean iteroinnin ja mukautumisen uusiin tarpeisiin ilman merkittäviä muutoksia alkuperäiseen koodiin.

Yhdistetyt funktiot eivät kuitenkaan automaattisesti takaa koodin luettavuutta. Liiallinen abstraktio voi tehdä koodista vaikeasti seurattavaa, erityisesti jos funktion yhdisteleminen tuo mukanaan monimutkaisia käsitteitä, jotka eivät ole suoraan yhteydessä ratkaistavaan ongelmaan. Turha abstraktio voi johtaa tilanteeseen, jossa koodin toiminnan ymmärtämiseksi täytyy perehtyä moniin välivaiheisiin ja ylimääräisiin konsepteihin, vaikka ongelma itsessään olisi yksinkertainen.

Koodiesimerkki 18 näyttää tilanteen, jossa Ramda.js-kirjastoa on käytetty yhdistetyn funktion rakentamiseen. Vaikka funktio toimii, sen ymmärtämiseen tarvitsee

paljon ylimääräistä tietoa.

```
const convertUsersToLeaderboardUsers = (users: Array<User>,
    anonymizeUsers: boolean) => R.pipe(
    R.pick(["userName", "score", "id"]),
    R.ifElse(
        R.always(anonymizeUsers),
        R.map(R.omit(["userName"])),
        R.identity
    ),
    R.map(R.assoc("type", "leaderboardUser")),
    R.sortWith([R.descend(R.prop("score"))])
)(users)
```

Koodiesimerkki 18: Funktio, muuttaa listan käyttäjiä sellaisiksi, että niitä voidaan käyttää tuloslistoilla. Funktio on yhdistetty monesta funktiosta käyttämällä Ramda.js-kirjaston funktioita. Funktiossa käytetään useita muita funktioita, joiden toiminnan lukijan on tiedettävä tai arvattava.

Sen sijaan esimerkkikoodissa 19 käytetään vain JavaScriptin sisäänrakennettuja työkaluja. Funktio on yleisemmin helppo lukea. Vaikka luettavuus on subjektiivista, ja perustuu pitkälti ohjelmointityylin tuttavuuteen, on todennäköisempää, että ohjelmoija ymmärtää JavaScriptin omia Array-metodeja, kuin minkäkin kirjaston funktioita, vaikka kirjaston funktiot olisikin pyritty kirjoittamaan noudattaen yleisiä käytänteitä.

```
const convertUsersToLeaderboardUsers = (users: Array<User>,
    anonymizeUsers: boolean) => users
    .map(user => ({
        type: "leaderboardUser",
        userName: anonymizeUsers
            ? undefined
            : user.userName,
        score: user.score
    }))
    .sort((userA, userB) => userB.score - userA.score)
```

Koodiesimerkki 19: Toiminnaltaan sama funktio, mutta käytettynä on vain JavaScriptin sisäänrakennettuja palikoita. Useampi lukija ymmärtää funktion, sillä se ei käytä ulkoisia kirjastoja.

On tärkeää, että abstrahoinnin taso pysyy hallittuna, eikä yhdistetty funktio tuo mukanaan lisämonimutkaisuutta. Abstraktioiden on oltava perusteltuja ja niiden tulee aidosti parantaa koodin ylläpidettävyyttä ja uudelleenkäytettävyyttä, eikä vain noudattaa jotain häilyviä teoreettisia malleja ilman konkreettisia hyötyjä.

Pitää siis pitää mielessä ohjelmointikieli, jossa toimitaan. Säännöt tulee sopeuttaa ympäristöön. Vaikka funktionaalisessa ohjelmoinnissa pyritään olla mutatoimatta dataa, tästä ei missään nimessä ole pakko pitää kiinni kynsin hampain.

## 4.3 Puhdas funktio ei ole aina paras ratkaisu

Funktionaalisessa ohjelmoinnissa pyritään välttämään mutatointia, eli ajonaikaisten muuttujien arvojen muuttamista. Tietorakenteiden muuttamisen sijasta pyritään luomaan tietorakenteista uusia kopioita ja jättämään vanhat ennalleen [72; 73].

Joissain ohjelmointikielissä, tai ohjelmointikielien kirjastoissa, muuttumattomien tietorakenteiden käyttäminen on saatu varsin tehokkaaksi [72; 73]. Näin ei kuitenkaan ole asian laita JavaScriptissä, tai ohjelmointikielissä yleisesti. Tiedon mutatointi on nopeaa ja kopiointi hidasta [74]. Näin ollen, jos kirjoitetaan funktioita, jotka palauttavat aina uusia kopioita, vanhojen mutatoimisen sijasta, saadaan usein hidasta ja tehotonta ohjelmakoodia.

Jossain vaiheessa ohjelman suoritusta tietokoneen bitit kääntyvät nollista ykkösiin, tai toisin sanoen mutatoituvat [34]. Niin tietokoneet toimivat, minkä vuoksi mutaatiota ei saa pelätä, kunhan sen hoitaa asianmukaisesti.

Jos funktio mutatoi ainoastaan muuttujia, joita se itse luo funktion ajon ajaksi, ei ole koodin turvallisuuden kannalta toiminnallista merkitystä mitä funktion sisällä tapahtuu. Kuitenkin tehokkuusvoitot voivat olla huomattavat versioihin, joissa mutatointia ei harjoiteta, kuten seuraavat esimerkit osoittavat.

Esimerkkinä on funktio, joka ottaa sisään listan avain-arvo-pareja, ja luo niistä olion, josta voi hakea arvoja avainten perusteella. Näytetään kaksi (funktionaalista) tapaa toteuttaa tämä mielivaltainen funktio: versio, joka ei mutatoi muuttujia missään vaiheessa (koodiesimerkki 20), ja versio, joka mutatoi vain itse luomiaan muuttujia (koodiesimerkki 21).

Tarkkailijan näkökulmasta molemmat funktiot ovat kuitenkin puhtaita.

```
const entries = Array.from({ length: 1000 }, (_, i) => [
    `key-${i}`,
    `value-${i}`
])

const objectFromEntries = (entries) =>
    entries.reduce((acc, entry) => {
        return { ...acc, [entry[0]]: entry[1] }
    }, {})

console.time('Immutable')
objectFromEntries(entries)
console.timeEnd('Immutable')
// Immutable: 112.718017578125 ms
```

Koodiesimerkki 20: Funktio, joka ottaa listan avain-arvo-pareja ja luo niistä olion. Olion luonnissa ei käytetä ollenkaan mutatointia Funktion suorittaminen 1000 avain-arvo-parille kesti noin 113 millisekuntia.

Täysin mutatoimattomalla versiolla kesti luoda tuhannesta avaimesta olio yli 300 kertaa kauemmin kuin versiolla, joka mutatoi käyttämiään muuttujia (113 ms ja 0,4 ms).

```
const entries = Array.from({ length: 1000 }, (_, i) => [
    `key-${i}`,
    `value-${i}`
])

const objectFromEntries = (entries) =>
    entries.reduce((acc, entry) => {
        acc[entry[0]] = entry[1]
        return acc
    }, {})

console.time('Mutable')
objectFromEntries(entries)
console.timeEnd('Mutable')
// Mutable: 0.35693359375 ms
```

Koodiesimerkki 21: Sama funktio kuin aiempi. Ainoa ero, että pareja iteroidessa luotavaa oliota mutatoidaan. Funktion suorittaminen 1000:lle avain-arvo-parille kesti vain noin 0,4 millisekuntia.

Useimmiten ohjelman tehokkuudella ei ole paljoa merkitystä, ja luotettavuus ja luettavuus on syytä pitää etusijalla. Kuitenkin tässä tilanteessa tehokkuuserot

ovat niin merkittävät, että jos 1 000 avain-arvo-parin sijasta, olisi käytetty 100 000, tai vaikka 1 000 000 avain-arvo-paria, mutatoimaton versio ei välttämättä olisi suoriutunut tehtävästä koskaan. Näitä määriä yritettiin, mutta suoritukset lopetettiin kesken, kun alkoi näyttämään, ettei tehtävä tosiaan ollut suoriutumassa koskaan.

## 4.4 Ongelmien mallintaminen lisää turvallisuutta

Funktionaalinen ohjelmointi mahdollistaa deklaratiivisen ja itsensä selittävän koodin kirjoittamisen [26]. Oikeiden nimien löytäminen funktioille tai muille ohjelmiston osille on vaikeaa, jos ongelma on mallinnettu väärin. Tässä osiossa käsitellään pragmaattista mallinnusta pala palalta kuvitteelliselle alustalle.

Käydään läpi valheellisen mallin haitat ja oikean mallin hyödyt. Staattisen tyyppijärjestelmän avulla oikea malli poistaa ohjelmavirheitä jo käännösvaiheessa ja vähentää testaustarpeita [75].

Oikeellisen mallin avulla koodi on luotettavampaa, ja sen avulla voidaan poistaa tarpeettomia if-lauseita. Jos mallista poistetaan mahdottomat tilat, niitä ei tarvitse edes testata [75]. Seuraavaksi muutetaan virheellinen malli oikeaksi.

Kuvitellaan, että ollaan rakentamassa rajapintaa verkossa toimivalle tietovisaalustalle, jossa tietovisa koostuu useasta monivalintakysymyksestä. Mallinnus voi alkaa näin.

```
type Question = {
  question: string
  answers: Array<string>
  correctAnswerIndices: Array<number>
}

type Quiz = {
  description: string
  questions: Array<Question>
}

```

Koodiesimerkki 22: Mahdollinen lähestymistapa yksinkertaiselle tietovisan mallinnukselle. Malli koostuu visasta (Quiz) ja se kysymyksistä (Question).

Yksinkertaisen tietovisan saisi toimimaan kyseisellä mallilla. Yksi kysymys tukee jopa montaa oikeaa vastausta. Mallintamisessa on kuitenkin ongelma. Se valehtelee. Sillä voidaan kuvata tiloja, joiden pitäisi olla mahdottomia.

**Listat voivat olla tyhjiä.** Malli ei pidä huolta, että tietovisassa on oltava vähintään yksi kysymys. Malli ei pidä myöskään huolta siitä, että kysymyksillä tulisi olla vähintään yksi vastaus. Tämä johtuu siitä, että `Array`-tietorakenne voi olla tyhjä.

Jos mallissa käytetään listaa, mutta se ei kuitenkaan todellisuudessa saisi olla tyhjä, niin ohjelmoijan on tarkistettava koodissa manuaalisesti, että listan sisältö on läsnä.

Vastauksena ongelmaan on luoda tyyppi listalle, joka yksiselitteisesti ei voi olla tyhjä (koodiesimerkki 23).

```
type NonEmptyArray<T> = {
    first: T
    rest: Array<T>
}
```

Koodiesimerkki 23: Mahdollinen lähestymistapa yksinkertaiselle tietovisan mallinnukselle. Tyypissä on käytetty tyyppimuuttujaa `T`. Tyyppimuuttuja tarkoittaa sitä, että sen sijalle voi laittaa minkä tahansa tyypin. Kyse on myös parametrisestä polymorfista (parametric polymorphism).

Tyypillä on kenttä `first`, jonka arvon on oltava olemassa, ja kenttä `rest`, jossa säilytetään listan loput alkiot. Tuon kentän lista taas on tavallinen `Array`. Pelkällä tyypin muutoksella voidaan pitää huolta, että listassa on 0..n sijasta 1..n alkiota.

```
type Question = {
    question: string
    answers: NonEmptyArray<string>
    correctAnswerIndices: Array<number>
}

type Quiz = {
    description: string
    questions: NonEmptyArray<Question>
}
```

Koodiesimerkki 24: Vaihtoehtoinen lähestymistapa tietovisan mallintamiselle, jossa käytetään itsemääritettyä NonEmptyArray-tyyppiä.

Nyt tietovisassa on oltava vähintään yksi kysymys (koodiesimerkki 24), ja jokaisella kysymyksellä vähintään yksi vastaus. Kuitenkin oikeita vastauksia ei ole välttämättä ainuttakaan, sillä kenttä `correctAnswerIncides` on edelleen tyypiltään `Array<number>`.

Staattisella tyyppijärjestelmällä voi pitää huolen, että mallia seurataan. Tarkastuksia tyhjyydestä ei tarvitse tehdä if-lauseilla. Malli ei kuitenkaan ole vieläkään täydellinen.

**Kenttien arvot voivat olla mitä tahansa.** Mikään ei pidä huolta, että esimerkiksi kentän `correctAnswerIncides` indeksit ovat oikeellisia. Myös kaikki, joissa tyyppinä on `string` voivat ottaa vastaan mitä tahansa merkkijonoja, myös tyhjiä merkkijonoja.

Merkkijonojen validointi TypeScriptin tyyppijärjestelmässä on kuitenkin hieman vaivalloista, eikä välttämättä tarpeellista. Kenttä `correctAnswerIncides` on kuitenkin ajatuksena erittäin huono, sillä malli ei millään tavalla määrittele, miten se liittyy kysymykseen tarkemmin. Mikään ei estä laittamasta täysin päättömiä numeroita ”oikeiksi” indekseiksi. Korjauskeinona on paras keino: kentän poistaminen.

Kentän voi poistaa kokonaan, jos mallinnusta viedään pidemmälle. Voidaan sopia, että on olemassa kahdenlaisia vastauksia: oikeita tai vääriä. Tämä on varsin helppo mallintaa (koodiesimerkki 25)

```
type CorrectAnswer = {
    type: "correct"
    answer: string
}
type IncorrectAnswer = {
    type: "incorrect"
    answer: string
}
```

Koodiesimerkki 25: Oikeille ja väärille vastauksille omat tyypit

Uusilla tyypeillä voidaan mallintaa kysymyksiä oikeellisemmin. Voidaan myös

määrittää, että kysymyksellä on oltava vähintään yksi oikea vastaus, ja nolla tai useampi väärä vastaus (koodiesimerkki 26).

```
type Question = {
    question: string
    correctAnswers: NonEmptyArray<CorrectAnswer>
    incorrectAnswers: Array<IncorrectAnswer>
}
```

Koodiesimerkki 26: Kysymykseen voi tarkentaa millaisia vastauksia hyväksytään.

Malli ei enää valehtele. Voi kuitenkin olla, että tulevaisuudessa on tarve luoda visalle enemmän ominaisuuksia. Voidaan haluta rajoittaa, että kysymyksessä voi valita vain yhden vaihtoehdon, tai usean. Voidaan esimerkiksi haluta antaa mahdollisuus antaa vastauksena vapaata tekstiä [75]. Muita tarpeita voi olla loputtomasti.

**Voidaan ottaa käyttöön summatyyppejä.** Sen sijasta, että lisättäisiin tyyppeihin uusia kenttiä kuten `isMultipleAnswerQuestion` tai `isFreform`, voidaan puolestaan luoda jokaiselle fundamentaalisesti erilaiselle asialle kokonaan oma tyyppinsä, ja yhdistää ne yhdeksi summatyypiksi. Näin pidetään huolta, että jokaisella

asialla voi olla vain kenttiä, jotka kuuluvat sille.

```
type MultipleAnswerQuestion = {
    type: "multipleAnswerQuestion"
    question: string
    correctAnswers: NonEmptyArray<CorrectAnswer>
    incorrectAnswers: Array<IncorrectAnswer>
}

type SingleAnswerQuestion = {
    type: "singleAnswerQuestion"
    question: string
    correctAnswer: CorrectAnswer
    incorrectAnswers: Array<IncorrectAnswer>
}

type FreformQuestion = {
    type: "freformQuestion"
    question: string
}

type Question = MultipleAnswerQuestion | SingleAnswerQuestion |
    FreformQuestion
```

Koodiesimerkki 27: Kysymysten mallintaminen summatyypillä. Monivalintakysymykselle, yksinkertaiselle kysymykselle ja vapaatekstikentälliselle kysymykselle on jokaiselle oma tyyppi. Tyypit on nostettu yhteen summatyyppiin (Question).

Kysymysten `type`-kentän perusteella voidaan tarkistaa, millaisesta kysymyksestä on todella kyse, jonka jälkeen voidaan suorittaa koodia, joka liittyy vain kyseisen kysymystyypin toimintaan.

Todellisuutta mallintavilla malleilla saadaan luotettavampaa koodia ja poistetaan turhia if-lauseita. Myös jos mallista poistetaan mahdottomat tilat, niitä ei tarvitse edes testata.

### 4.4.1 Virheet mukaan mallintamiseen

Ohjelmissa tapahtuu virheitä. Vaikka pyrittäisiin malleihin, joista on poistettu mahdottomat tilat, niin ohjelmia ajaessa tullaan silti törmäämään virheisiin.

Miten on hyvä mallintaa tilanteita, kun virheitä tapahtuu?

Yleisesti imperatiivisissa paradigmoissa virhetilanteissa metodit ja funktiot palauttavat `null`-arvoja, virheitä tai poikkeuksia. Funktionaalisessa ohjelmoinnissa voidaan käyttää samoja keinoja, mutta useimmin funktioiden suorituksen räjäyttämisen sijasta funktiot palauttavat summatyyppejä, tai monadeja, joissa voi olla sisällä joko oikea palautus- tai virhetyyppi.

TypeScriptissä yksinkertaisimmillaan tämän voi toteuttaa yhdellä omalla tyypillä (koodiesimerkki 28).

```
type Maybe<T> = T | null
const mightWorkMightNot = (): Maybe<string> => {
    if (Math.random() > 0.5) {
        return null
    }
    return "hello world"
}
```

Koodiesimerkki 28: Mahdollisesti puuttuvan paluuarvon malli. Maybe voi sisältää jonkin tyypin tai arvon `null`.

Deklaratiivisesti funktio kertoo, että se saattaa palauttaa merkkijonon, tai saattaa olla palauttamatta. Ehkä se on merkkijono, ehkä se ei ole. Tyypin käyttäminen on hieman tönkköä, sillä jos funktion oikea palautusarvo on null, on mahdotonta sanoa, oliko funktiokutsu onnistunut vai ei.

Tyypin voi viedä pidemmälle, jos arvot kääritään objekteihin. Vaihdetaan nimi Result-tyypiksi, että tyyppi voi kapseloida myös virheitä. (koodiesimerkki 29.)

```
type Success<T> = { value: T }
type Failure = { error: Error }
type Result<T> = Success<T> | Failure

const mightWorkMightNot = (): Result<string> => {
    if (Math.random() > 0.5) {
        return { error: new Error("Sorry, bad failure.") }
    }
    return { value: "hello world" }
}
```

Koodiesimerkki 29: Vaihtoehtoinen malli mahdollisesti epäonnistuvalle paluuarvolle. Result on joko Success, tai Failure. Molemmissa tapauksissa arvo on kuitenkin yhden kentän objekti, jonka ansiosta tyyppiin voi tallentaa `null`-arvon ilman, että tietoa menetetään.

Näin Result-tyyppi voi sisältää myös arvon `null`, sillä Result-objektin tarkan tyypin voi nyt päätellä ilman sitä. Nyt tarkan tyypin voi määrittää `value`- tai `error`-kentän läsnäolosta tai puutteesta. Näin rakennettuna tyypille on myös helppo rakentaa apufunktioita, joilla selvitetään, onko paluuarvo onnistunut, vai ei. Voi myös luoda apufunktiot, joilla luoda Result-objekteja (koodiesimerkki 30).

```
const isFailure = <T>(x: Result<T>): x is Failure =>
    "error" in x
const isSuccess = <T>(x: Result<T>): x is Success<T> =>
    !isFailure(x)
const failureOf = (error: string): Failure =>
    ({ error: new Error(error) })
const successOf = <T>(value: T): Success<T> =>
    ({ value })
```

Koodiesimerkki 30: Apufunktioita Result-tyypin tarkastamiseen ja luomiseen TypeScriptissä.

Näillä voidaan rakentaa yhdistettyjä funktioita, joissa paluuarvon luonti ja palauttaminen on eksplisiittistä ja selkeää. Koodiesimerkissä 31 käsitellään Result-objekteja yhdistetyssä funktiossa. Jokaisessa yhdisteen funktiossa tarkistetaan, onko aiempi tulos epäonnistunut. Epäonnistumisen sattuessa virhearvo pääste-

tään suoraan putken läpi.

```
const helloMaybe = (): Result<string> => {
    if (Math.random() > 0.5) {
        return failureOf("Sorry, bad failure.")
    }
    return successOf("hello world")
}

const addExclamation = (x: Result<string>): Result<string> => {
    if (isFailure(x)) return x
    return successOf(x.value + "!")
}

const toUpperCase = (x: Result<string>): Result<string> => {
    if (isFailure(x)) return x
    return successOf(x.value.toUpperCase())
}

pipe([
    helloMaybe,
    addExclamation,
    toUpperCase
])()
// Palauttaa { value : "HELLO WORLD!}
// tai { error: Error("Sorry, bad failure.") }
```

Koodiesimerkki 31: Esimerkki yhdistetystä funktiosta Result-tyypin kanssa. Jokaisen välifunktion on tarkastettava, onko edellinen kutsu onnistunut vai ei.

Result-tyypin käyttö on selkeää, ja TypeScriptillä on pakko tarkistaa, onko arvo `Success` vai `Failure`, ennen kuin sen käärittyä arvoa voi tarkastella tai muokata.

Jos yksikin funktio palauttaa Result-objektin, jossa on virhe, objekti kulkee muuttumattomana loppujen funktioiden lävitse.

Kuitenkin ongelma on, että tämänkaltaisena Result-tyypin käyttäminen on loppujen lopuksi vain syntaksisokeroitu null-arvon tarkistus. Ne if-lauseet, joita aiemmin poistettiin todenmukaisella tyyppimallinnuksella, ovat tulleet takaisin aina pakollisen virhetarkistamisen muodossa.

Result-tyypillä on kuitenkin jotain, jolla sen voi viedä vielä askelta pidemmälle. Result-tyypin voi toteuttaa monadina.

### 4.4.2 Result-monadi

Monadi oli suuressa osassa teoriaosuudessa. Jatketaan mahdollisesti epäonnistuvan ohjelman toteuttamista siten, että siinä on mukana monadisia periaatteita. Monadia käyttämällä voidaan siirtää virheentarkastuksen if-lauseet yhteen paikkaan keskitetysti.

Luodaan `Result`-monadi (koodiesimerkki 32).

```
class Result {
    constructor(x) {
        this.value = x
    }
    static of(x) {
        return new Result(x)
    }
    bind(f) {
        if (this.value instanceof Error) {
            return this
        }

        const result = f(this.value)

        if (!(result instanceof Result)) {
            return Result.of(result)
        }

        return x
    }
}
```

Koodiesimerkki 32: Result implementoituna monadiksi. Result monadeja voi luoda staattisella `Result.of`-metodilla, ja monadiin voi ketjuttaa operaatioita `Result.prototype.bind`-metodilla. Jos ketjutuksessa monadin arvo on Error, niin ketjutus lopetetaan, ja saatu virhearvo kuljetetaan suoraan lävitse.

Jos `Result`-monadi otetaan käyttöön aiemman osan koodiesimerkkiin, huoma-

taan luettavan koodin määrän pienentyneen (koodiesimerkki 33).

```
const helloMaybe = (): string | Error =>
    Math.random() > 0.5
        ? "hello world"
        : new Error("Sorry, bad failure.")

const addExclamation = (x: string) => x + "!"
const toUpperCase = (x: string) => x.toUpperCase()
const reverse = (x: string) => x.split("").reverse().join("")

Result.of(helloMaybe())
 .bind(addExclamation)
 .bind(toUpperCase)
 .bind(reverse)
```

Koodiesimerkki 33: Monadisia operaatioita ketjutettuna Result-monadiin. Annettavat funktiot voivat palauttaa käärimättömiä arvoja johtuen Result-monadin implementaatiosta.

Jos yksikin monadin funktioketjusta epäonnistuu, suoritus lakkaa ja virhe palautuu loppuun asti. Arvon voi vetää ulos monadista useilla eri tavoilla. Implementaation ansiosta tässä yhteydessä arvon voi ottaa ulos monadista yksinkertaisesti käyttäen `value`-kenttää (koodiesimerkki 34)

```
const monad = Result.of(helloMaybe())
    .bind(addExclamation)
    .bind(toUpperCase)
    .bind(reverse)
 const value = monad.value
 // value on joko "!DLROW OLLEH" tai Error("Sorry, bad failure")
```

Koodiesimerkki 34: Arvon voi poistaa monadin kontekstista yksinkertaisesti hakemalla `value`-kenttää.

Eri ohjelmointikielissä tai kirjastoissa monadeista arvon ulos saaminen voi poiketa. Matemaattisessa määritelmässä arvoa ei ole tarkoitus koskaan saada pois monadista. Esimerkiksi Haskellissa lähtökohtaisesti arvo kuuluu aina jättää monadinsa kontekstiin, eikä sitä tulisi käsitellä suoraan [76].

Result-monadin luominen ja käyttäminen vaatii ohjelmiston kehittäjiltä ylimääräistä kontekstia funktionaalisista käsitteistä. Kontekstin poistamiseksi voidaan käyttää räätälöidyn monadin sijasta JavaScriptin `Promise`-tietorakennetta (koodiesi-

merkki 35).

```
const mightSayHello = async () => {
    if (Math.random() > 0.5) {
        throw new Error("Sorry, bad failure")
    }
    return 'hello world'
}

const promise = Promise.resolve(mightSayHello())
    .then(x => x+"!")
    .then(x => x.toUpperCase())
    .then(x => x.split("").reverse().join(""))
//  .catch(x => x)
```

Koodiesimerkki 35: Result-monadi-esimerkki käyttäen JavaScriptin sisäänrakennettua Promise-tietorakennetta. Toimintaperiaate vastaa aiempaa koodiesimerkkiä, eikä vaadi erillisiä kirjastoja.

`Promise`:n käyttäminen Result-monadin sijasta voi selkeyttää ohjelmakoodin luettavuutta JavaScript-kehittäjille. Toisaalta `Promise`:n käyttäminen esimerkin tavalla tuo turhaa monimutkaisuutta, sillä `Promise`:n luontainen asynkronisuus tuo turhia tehokkuushaittoja, kun operaatiot eivät muuten olisi asynkronisia [69].

Ohjelmistokehittäjä Jake Archibald suosisi ohjelman kirjoittamista vielä enemmän JavaScriptin tyylillä (koodiesimerkki 36) [57] .

```
function someWellNamedFunction(): string | null {
    const result = mightSayHello();
    if (result === null) return;
    return reverse(result + '!').toUpperCase();
}
```

Koodiesimerkki 36: Ohjelmistokehittäjä Jake Archibaldin versio funktiosta [57]. Versio suosii imperatiivisia tapoja toteuttaa funktio.

Saman asian voi siis toteuttaa monella eri tavalla. On suositeltavaa asettaa tiukan arvioinnin alle se, mikä versio sopii mihinkin tilanteeseen.

# 5 Kokemukset

Tässä luvussa käsitellään ohjelmointikokemuksia erityisesti funktionaalisen ja oliopohjaisen ohjelmoinnin näkökulmista. Pohditaan, miten kokemukset, ohjelmointiparadigmat ja käytetyn kielen ominaisuudet vaikuttavat luettavuuteen, uudelleenkäytettävyyteen ja tehokkuuteen.

Kokemuksella on suuri rooli siinä, miten ohjelmoija lähestyy näitä teemoja. Vastaavasti eri ohjelmointiparadigmat — kuten funktionaalinen ja olio-ohjelmointi — tarjoavat erilaisia työkaluja ja periaatteita, jotka voivat joko helpottaa tai hankaloittaa näiden tavoitteiden saavuttamista. Ohjelmointikielen ominaisuudet, kuten Pythonin yksinkertaisuus tai JavaScriptin toiminnalliset työkalut, voivat myös muokata koodikäytäntöjä ja vaikuttaa siihen, mikä on optimaalista kussakin tilanteessa.

## 5.1 Luettavuus

Puidaan luettavuutta hieman eri näkökulmista. X-alustalla keskusteltiin entisen Google kehittäjän Jake Archibaldin kanssa, joka suostui vastaamaan funktionaaliseen ohjelmointiin liittyviin kysymyksiin [77].

Yli 20 vuotta JavaScriptia ohjelmoinut Archibald pyrkii ohjelmoimaan koodia, joka on kuin suoraan ohjelmoinnin alkeita käsittelevästä oppikirjasta [34; 57]. Hän pyrkii olemaan käyttämättä monimutkaisempia ohjelmoinnin työkaluja [34], kuten esimerkiksi `Array.prototype.reduce`, osittaissoveltaminen (partial application) tai säännölliset lausekkeet (regular expressions) [34; 77].

Hän suosii for-silmukoiden käyttämistä funktionaalisen `reduce`-funktion sijasta, ja perustelee valintaa luettavuudella [34; 57; 77]. Yleisesti Archibald kuitenkin kokee funktionaalisen ohjelmoinnin olevan tuonut positiivisa vaikutteita JavaScriptiin [77]. Luettavuuden kannalta merkittävimpiä tekijöitä ovat keskimääräinen rivin pituus, tunnisteiden määrä per rivi ja keskimääräinen sulkeiden määrä [78, s. 8].

Edsger Dijkstran mukaan ohjelman luettavuus riippuu suurelta osin sen ohjausrakenteiden yksinkertaisuudesta [79]. Ohjelmoijat oppivat ensin for-silmukoita, ennen kuin ymmärtävät, mitä `reduce` (tai `fold`) tarkoittaa. Kuitenkin tunnisteet ja rakenteet, jotka ohjelmoija tuntee, ovat henkilökohtaisia ja perustuvat kokemuksiin sekä opiskelutaustaan.

Myös Pythonin luojana tunnettu Guido van Rossum halusi poistatti Python 3 versiosta `reduce`-funktion, ja perusteli sitä luettavuudella, ja kielen yksinkertaistamisella [80].

Voi siis ajatella, että luettavuus on tarpeellista suhteuttaa työympäristöön. Jos koodikannassa suositaan funktioiden yhdistelemistä, tätä käytäntöä kannattaa noudattaa. Jos taas for-silmukoita käytetään mieluummin `reduce`-funktion sijasta, ne ovat parempi valinta.

Kokemuksien mukaan voi olla vaikeaa olla käyttämättä työkaluja, joiden näkee sopivan tilanteeseen. Kuitenkin kannattanee ottaa askel taaksepäin ja pitää huolta, että ohjelmakoodi on aina luettavaa tilanteeseen sopien.

## 5.2 Uudelleenkäytettävyys

Työpaikalla keskusteluissa on näkynyt käytänteitä, että ennen kuin jotakin tarvitaan kolmesti, ei sille ole kannattavaa kirjoittaa uudelleenkäytettävää abstraktiota. Toisaalta kehitettäessä ulkoisia rajapintoja, on tilanne eri, sillä on pidettävä huolta taaksepäin suuntautuvasta yhteensopivuudesta.

Uudelleenkäytettävien palikoiden löytäminen ei myöskään ole itsestäänselvyys. On keskusteltu siitä, että kannattaisi välttää logiikan koostamista, ja toisinaan suosia yksinkertaisuuksia ja kovakoodattua tietoa, jotta sen etsiminen olisi koodikannasta helpompaa (esimerkiksi käyttämällä grep-komentorivikomentoa).

Uudelleenkäytettävyyttä voidaan harjoittaa erinäköisillä koodikäytänteillä, kuten erinäköisin funktionaalisen ohjelmoinnin periaattein, tai olio-ohjelmoinnin periaattein. Archibaldin mukaan koodikäytänteet usein epäonnistuvat tehtävässään, kun

niissä yritetään liikaa pysytellä niiden puhtaissa akateemisissa raameissa [57].

### 5.3 Tehokkuus

Tehokkuutta ei ole suotavaa miettiä liikaa ennen kuin ohjelmoidaan systeemejä, jotka todella riippuvat siitä. Monet ovat tätä mieltä [28; 57]. Kun tehokkuusongelmat alkavat näkyä, on kokemusten mukaan todennäköisempää, että yhden ohjelman kriisipesäkkeen (hot spot) korjaamisella tilanne voidaan korjata, eikä tuhannen sivalluksen kuolema ole todennäköisin.

Ohjelman tehokkuuteen vaikuttavien asioiden tiedostaminen kuitenkin ennaltaehkäisee ongelmatilanteita. Esimerkiksi tiedon kopioinnin tai `reduce`-funktion välttämisellä voidaan ajaa proaktiivisesti ohjelmaa toimivampaan suuntaan.

## 6 Johtopäätökset ja suositukset

Funktionaalisen ohjelmointi eroaa fundamentaalisesti tavanomaisesti opetetuista ohjelmointiparadigmoista. Tässä luvussa käydään työn keskeiset johtopäätökset ja mahdolliset jatkotutkimuskohteet. Johtopäätöksissä pohditaan funktionaalisen ohjelmoinnin soveltamista perinteisempään ohjelmointiin painottaen maltillisuutta. Jatkon osalta käsitellään mahdollisia aihepiirin syventymisalueita, joilla voi rikastuttaa ohjelmoinnin käsityksen monimuotoisuutta.

### 6.1 Johtopäätökset

Funktionaalisen ohjelmoinnin periaatteet voidaan integroida perinteisempään ohjelmointiin. Liiallinen pedanttisuus käsitteitä kohtaan kuitenkin saattaa johtaa ohjelman ylläpidettävyyden ongelmiin, erityisesti muuttuvan koodin tuottamisessa asiakaskäyttöön. Ongelmat tulee mallintaa oikein, sillä tämä vähentää tarpeettomien testausvaiheiden ja if-lauseiden tarvetta. Hyvin mallinnettuna ohjelmakoodin määrä vähenee ja ylläpidettävyys paranee.

Akateemiset käsitteet funktionaalisessa ohjelmoinnissa ovat tiukasti määritelty-

jä, mutta niiden noudattaminen ei vaadi pilkuntarkkuutta. Sokeaa sääntöjen noudattamista ilman perusteltua syytä tulee välttää. Yhdistetyt funktiot, funktioiden puhtaus, tiedon muuttumattomuus, algebralliset tietotyypit ja rakenteet ovat toimivia työkaluja ohjelmien rakentamisessa. Jos ohjelmointikieli ei tue niitä natiivisti, niiden mukaan ottaminen yleisesti heikentää koodin luettavuutta. On kuitenkin tärkeää kunnioittaa määritettyjä käytänteitä; kokemusten mukaan keskinkertaisetkin käytänteet ovat parempia kuin niiden puuttuminen kokonaan.

On ollut avartavaa huomata, kuinka esimerkiksi `Promise` ja `Array` ovat pohjimmiltaan monadirakenteita. Vaikka tämä tieto auttaa ymmärtämään näiden tietorakenteiden samanlaisuuksia, voi olla hyödyllistä olla tuomatta tietoa ilmi jokaisessa koodikatselmuksessa. Tiedon konkreettista hyötyä on vaikea mitata.

## 6.2 Aihepiirin syventäminen

Jatkuva kehitys on ohjelmointialan koettu vaatimus. Kirjallisuus, ohjelmointikielet, kirjastot, filosofiat ja näiden kaikkien puolesta ja vastaan puhujat eivät tule loppumaan kesken.

Funktionaalinen ohjelmointi on laaja ja monimuotoinen alue, joka kattaa teoreettisia käsitteitä ja käytännön sovelluksia. Alan jatkotutkimus voisi käsittää useita näkökulmia, ohjelmointikieliä ja tunnettujen ohjelmoijien julkaisuja. Moniulotteinen lähestymistapa on oleellinen, sillä pelkkä funktionaalisen ohjelmoinnin periaatteiden tutkiminen ei välttämättä riitä pragmaattisten ohjelmointitaitojen kehittämiseen.

### 6.2.1 Kategoriateoria

Kategoriateorian tutkiminen voi syventää ymmärrystä funktionaalisen ohjelmoinnin periaatteista ja rakenteista. Kategoriateoria auttaa ymmärtämään funktioiden ja rakenteiden suhteita, mikä on keskeistä funktionaalisessa ohjelmoinnissa. Tämän ymmärryksen kautta voidaan saavuttaa tehokkaampia ja elegantimpia koodiratkaisuja, joissa turha toisto poistuu. Teorian läsnäolon tunnistaminen voi vahvistaa omia näkemyksiä toimimisesta, vaikka sen käyttö käytännössä ei olisi pakol-

lista. Bartosz Milewskin kirja ’Category Theory for Programmers’ vaikuttaa erittäin lupaavalta tavalta opiskella kategoriateoriaa ohjelmoijan näkökulmasta [49].

### 6.2.2 Fantasyland-spesifikaatio

Fantasyland-spesifikaatio on joukko sääntöjä algebrallisten rakenteiden, kuten monadien ja funktioiden, yhdistämiseksi JavaScriptissä [81]. Se perustuu kategoriateoriaan ja määrittelee, miten funktiot ja rakenteet vuorovaikuttavat, parantaen niiden yhteensopivuutta. Spesifikaatio sisältää esimerkiksi monadien `of` ja `bind` -operaatioiden säännöt. Fantasyland on erittäin teoreettinen ja määritelmällisesti tiukka, mikä kannustaa pedanttiseen funktionaaliseen ohjelmointiin, eroten tämän opinnäytetyön filosofiasta.

### 6.2.3 Fp-ts-kirjasto

Fp-ts-kirjasto auttaa puhtaan funktionaalisen ohjelmoinnin ymmärtämisessä TypeScript-ympäristössä. Kirjaston syntaksi poikkeaa perinteisestä TypeScript-koodista merkittävästi. Se on kuitenkin yksi parhaista vaihtoehdoista puhtaan funktionaalisen ohjelmoinnin opiskeluun suoraan TypeScriptissä [59].

### 6.2.4 Eri ohjelmointikielet

Eri ohjelmointikielet tarjoavat uusia näkökulmia funktionaaliseen ohjelmointiin. Haskellin tiukka tyyppijärjestelmä, pohja kategoriateoriassa, ja puhtaat funktiot opettavat funktionaalisen ohjelmoinnin keskeisiä periaatteita puhtaasti [36; 58; 76]. Elixir yhdistää funktionaalisuuden ja ohjelman suorituksen rinnakkaisuuden, mikä auttaa ymmärtämään tehokasta moniydinkäsittelyä [82]. Go-ohjelmointikielen opiskelu voisi myös muuttaa merkittävästi ajattelua ohjelmoinnista [83]. Go ei ole funktionaalinen ohjelmointikieli, vaikka se tukee joitain funktionaalisen ohjelmoinnin käsitteitä, kuten korkeamman asteen funktioita. Go:n käytännönläheisyys osoittaa, miten funktionaalisia periaatteita voi soveltaa tai jättää soveltamatta [83].

### 6.2.5 Tunnetut ohjelmistokehittäjät

Funktionaalisen ohjelmoinnin ymmärtäminen vaatii monipuolista näkökulmaa eri lähteistä. Eri ohjelmistokehittäjien työ tarjoaa arvokkaita esimerkkejä ja oivalluksia, jotka rikastuttavat ymmärrystä funktionaalisten periaatteiden soveltamisesta ohjelmistokehityksessä.

Esimerkiksi Rich Hickey (Clojure-kielen kehittäjä) on tunnettu hyvin artikuloiduista ja mielekkäistä puheistaan ohjelmointifilosofioista funktionaaliseen ohjelmointiin liittyen [61; 73]. Jake Archibald on ylläpitänyt HTTP 203 -sarjaa, nykyään Off the Main Thread -podcastia, sekä pragmaattista blogia, joissa hän käsittelee web-teknologioita vankalla kokemuksella [34; 67]. Richard Feldman, tunnettu Elm-kielen puolestapuhuja, on osallistunut keskusteluihin funktionaalisesta ohjelmoinnista ja pitänyt puheita sen periaatteista. Hän korostaa ennustettavuutta ja virheiden vähentämistä, erityisesti käyttöliittymäkehityksessä [3; 31; 75].

Opiskelemalla esimerkiksi näiden kehittäjien töitä löytyy arvokkaita oivalluksia ohjelmistokehitykseen niin funktionaalisen ohjelmoinnin, kuin muun ohjelmoinnin puolelta. Heitä yhdistää pragmaattisuus: ei turhaa teoriaa ilman käytäntöön perustuvaa perustelua.

# 7 Yhteenveto

Psykologi Kurt Lewin teki tunnetuksi sanonnan ’mikään ei ole niin käytännöllistä kuin hyvä teoria’ [84]. Tämä näkyy funktionaalisessa ohjelmoinnissa sekä hyvässä että pahassa: funktionaalisilla käsitteillä voidaan rakentaa teoreettisesti tehokkaita, luotettavia ja kompakteja ohjelmia, mutta sokea teorian seuraaminen johtaa tehokkuushaittoihin, luettavuusongelmiin ja liian nopeaan abstrahointiin.

Ohjelmien mallinnuksessa teoreettisuus ja funktionaalisen ohjelmoinnin deklaratiivisuus on hyödylllistä. Mallit, jotka eivät valehtele, vähentävät ylläpidettävän koodin määrää ja testaamisen tarvetta. Mallintamista voidaan viedä pidemmälle rakenteilla kuten monadeilla, jolloin mallin ja toteutuksen raja hämärtyy.

Työn alkuvaiheessa olin vakuuttunut funktionaalisen ohjelmoinnin merkittävistä eduista. Nyt kokemukset vievät suuntaan, jossa funktionaalinen ohjelmointi näyttäytyy pikemminkin epäsuorana voittona. Kuvainnollisesti sanottuna kaikukammion seinä on räjäytetty.

Ohjelmointia tehdään yhteistyössä, joten yhteisiä käytäntöjä on syytä noudattaa. Funktionaalisen ohjelmoinnin käsitteet ja käytänteet ovat ilmaisuvoimaisia ja ennustettavia, mutta myös erittäin teoreettisia ja tiukkoja, mikä vaatii opiskelua ja perustelua.

Löydettiin käytänteitä, joita on luonteva ottaa käyttöön tuleviin projekteihin, ja käytänteitä, joiden käyttöä kannattaa harkita tarkasti. Tutkimuksen lähteissä funktionaalisen ohjelmoinnin hyödyt ilmenivät parhaiten alunperin funktionaalisissa ohjelmointikielissä. Kunkin kielen ominaisuudet vaikuttavat merkittävästi siihen, kuinka tehokkaasti ja sujuvasti funktionaalisia periaatteita voidaan soveltaa.

## Lähteet


1 TypeScript: JavaScript With Syntax For Types 2024. Microsoft. Verkkoaineisto. <https://www.typescriptlang.org/>. Luettu 09. 09. 2024.

2 Stack Overflow. 2008. What is a monad? Verkkoaineisto. <https://stackoverflow.com/questions/44965/what-is-a-monad>. Luettu 29. 08. 2024.

3 — 2017. Why are promises monads? Verkkoaineisto. <https://stackoverflow.com/questions/45712106/why-are-promises-monads>. Luettu 29. 08. 2024.

4 Miller, Paul et al. 2013. Incorporate monads and category theory. Promises/A+. Verkkoaineisto. <https://github.com/promises-aplus/promises-spec/issues/94>. Luettu 31. 08. 2024.

5 CodeAesthetic. Joulukuu 2023. Dear Functional Bros. Video. <https://www.youtube.com/watch?v=nuML9SmdbJ4>. Katsottu 10. 09. 2024.

6 Cantarella, Toni. 2024. Funktionaalisen ohjelmoinnin hyödyt ja haitat. Opinnäytetyö. Karelia-ammattikorkeakoulu. Theseus-tietokanta.

7 Gartner. Toukokuu 2022. Gartner Predicts 65% of B2B Sales Organizations Will Transition from Intuition-Based to Data-Driven Decision Making by 2026. Stamford, Conn. Verkkoaineisto. <https://www.gartner.com/en/newsroom/press-releases/gartner-predicts-65--of-b2b-sales-organizations-will-transition->. Luettu 08. 09. 2024.

8 Tan, Gang. 2004. A Brief History of Functional Programming. Penn State University. Verkkoaineisto. <https://www.cse.psu.edu/~gxt29/historyOfFP/historyOfFP.html>. Luettu 08. 09. 2024.

9 Computerphile. 2017a. Lambda Calculus. Video. <https://www.youtube.com/watch?v=eis11j_iGMs>. Katsottu 31. 08. 2024.

10 Okhravi, Christopher. 2024. The Object Oriented Way. Leanpub. Luettu 29. 08. 2024.

11 Martin, Robert C. 2019. Why Clojure? The Clean Code Blog. Verkkoaineisto. <https://blog.cleancoder.com/uncle-bob/2019/08/22/WhyClojure.html>. Luettu 31. 08. 2024.

12 — 2017. Pragmatic Functional Programming. The Clean Code Blog. Verkkoaineisto. <https://blog.cleancoder.com/uncle-bob/2017/07/11/PragmaticFunctionalProgramming.html>. Luettu 31. 08. 2024.

13 Google Trends. 2024. Erlangin, Haskelin, OCamlin, Elixirin ja Scalan suosiokehitys. Verkkoaineisto. <https://trends.google.com/trends/explore/TIMESERIES/1724633400?hl=fi&tz=-180&date=all&hl=fi&q=%2Fm%

2F02mm3,%2Fm%2F03j_q,%2Fm%2F09wmx,%2Fm%2F0pl075p,%2Fm%2F091hdj&sni=3>. Luettu 26. 08. 2024.

14 Oracle. 2024a. Package java.util.function - Java Platform SE 8. <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>. Luettu 26. 08. 2024.

15 — 2024b. Interface Stream - Java Platform SE 8. <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>. Luettu 26. 08. 2024.

16 Sundström, Jarno. 2021. Java funktionaalisessa ohjelmoinnissa. Insinöörityö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

17 MDN Web Docs. 2024a. Array.prototype.map. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>. Luettu 26. 08. 2024.

18 — 2024b. Array.prototype.filter. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter>. Luettu 26. 08. 2024.

19 — 2024c. Array.prototype.reduce. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce>. Luettu 26. 08. 2024.

20 Vakil, Anjana. Kesäkuu 2016. Learning Functional Programming with JavaScript. JSConf. Video. <https://www.youtube.com/watch?v=e-5obm1G_FY>. Katsottu 29. 08. 2024.

21 Web Dev Simplified. 2019. 8 Must Know JavaScript Array Methods. Video. <https://www.youtube.com/watch?v=R8rmfD9Y5-c>. Katsottu 29. 08. 2024.

22 MDN Web Docs. 2024d. Set.prototype.union. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/union>. Luettu 26. 08. 2024.

23 — 2024e. Set.prototype.intersection. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/intersection>. Luettu 26. 08. 2024.

24 — 2024f. New JavaScript Set methods. Verkkoaineisto. <https://developer.mozilla.org/en-US/blog/javascript-set-methods/>. Luettu 26. 08. 2024.

25 TC39. 2024a. Iterator Helpers. Verkkoaineisto. <https://tc39.es/proposal-iterator-helpers/>. Luettu 26. 08. 2024.

26 Wagner ym. 2024. Functional programming vs. imperative programming (LINQ to XML). Verkkoaineisto. <https://learn.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming#functional-programming-vs-imperative-programming>. Luettu 30. 08. 2024.

27 Wikipedia. 2024a. Functional fixedness. Verkkoaineisto. <https://en.wikipedia.org/wiki/Functional_fixedness>. Luettu 08. 09. 2024.

28 C2 Wiki. 2024. Premature Optimization. Verkkoaineisto. <https://wiki.c2.com/?PrematureOptimization>. Luettu 26. 08. 2024.

29 PYPL. 2024. PopularitY of Programming Language. Verkkoaineisto. <https://pypl.github.io/PYPL.html>. Luettu 29. 08. 2024.

30 Dijkstra, Edsger W. Lokakuu 1972. ”The humble programmer”. Commun. ACM 15.10, s. 859–866. <https://doi.org/10.1145/355604.361591>.

31 Feldman, Richard. 2021. Functional Programming for Pragmatists. GOTO. Video. <https://www.youtube.com/watch?v=3n17wHe5wEw>. Katsottu 14. 09. 2024.

32 Blelloch, Guy E. & Harper, Robert. Lokakuu 2015. λ-Calculus: The Other Turing Machine. Verkkoaineisto. <https://www.cs.cmu.edu/~rwh/talks/cs50talk.pdf>. Luettu 08. 09. 2024.

33 Vlasblom, Aldwin. 2020. Common combinators in JavaScript. Verkkoaineisto. <https://gist.github.com/Avaq/1f0636ec5c8d6aed2e45>. Luettu 31. 08. 2024.

34 Chrome for Developers. Tammikuu 2020. Is reduce() bad? - HTTP 203. Google. Video. <https://www.youtube.com/watch?v=qaGjS7-qWzg>. Katsottu 09. 09. 2024.

35 Wlaschin, Scott. 2012. Function Composition. Verkkoaineisto. <https://fsharpforfunandprofit.com/posts/function-composition/>. Luettu 09. 09. 2024.

36 Haskell Wiki. 2006. Function Composition. Verkkoaineisto. <https://wiki.haskell.org/Function_composition>. Luettu 09. 09. 2024.

37 TC39. 2024b. ES pipe operator (2021). Verkkoaineisto. <https://tc39.es/proposal-pipeline-operator/>. Luettu 09. 09. 2024.

38 Dalgård, Kristian et al. Helmikuu 2016. Why prefer compose over pipe? ramda. Verkkoaineisto. <https://github.com/ramda/ramda/issues/1642>. Luettu 09. 09. 2024.

39 MDN Web Docs. 2024g. Set. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set>. Luettu 26. 08. 2024.

40 Kennedy, K. & Schwartz, J. 1975. ”An introduction to the set theoretical language SETL”. Computers & Mathematics with Applications 1.1, s. 97–119. <https://www.sciencedirect.com/science/article/pii/0898122175900115>.

41 Wikipedia. 2024b. Type theory. Verkkoaineisto. <https://en.wikipedia.org/wiki/Type_theory>. Luettu 15. 09. 2024.

42 Jurgelėnas, Marius. Kesäkuu 2019. Algebraic Data Types. You Gotta Love Frontend. Video. <https://www.youtube.com/watch?v=JYWJzaiCtEw>. Katsottu 15. 09. 2024.

43 Stack Exchange. 2018. What exactly is the semantic difference between set and type? Verkkoaineisto. <https://cs.stackexchange.com/questions/91330/what-exactly-is-the-semantic-difference-between-set-and-type>. Luettu 15. 09. 2018.

44 Pennanen, Viljo. Syyskuu 2024. Hedelmät. Kuvatiedostoon saatu oikeudet suoraan tekijältä.

45 Ecma International. Kesäkuu 2024. ECMA-262, 15th edition, June 2024 ECMAScript® 2024 Language Specification. Verkkoaineisto. <https://262.ecma-international.org/>. Luettu 09. 09. 2024.

46 Milewski, Bartosz. 2016a. Category Theory 1.1: Motivation and Philosophy. Video. <https://www.youtube.com/watch?v=I8LbkfSSR58>. Katsottu 09. 09. 2024.

47 Mellin, Ilkka. 2005. Johdatus todennäköisyyslaskentaan: Joukko-oppi. <https://math.tkk.fi/opetus/sovtoda/luennot/vanhat/JOLIITE1.pdf>.

48 Wikipedia. 2024c. Kategoriateoria. Verkkoaineisto. <https://fi.wikipedia.org/wiki/Kategoriateoria>. Luettu 09. 09. 2024.

49 Milewski, Bartosz. 2017. Category Theory for Programmers. <https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>.

50 Beck, Jonathan Mock. 2003. ”Triples, algebras and cohomology”. Reprints in Theory and Applications of Categories 2, s. 1–59. <http://www.tac.mta.ca/tac/reprints/articles/2/tr2abs.html>.

51 Computerphile. 2017b. Lambda Calculus. Video. <https://www.youtube.com/watch?v=t1e8gqXLbsU>. Katsottu 13. 10. 2024.

52 Milewski, Bartosz. 2016b. Category Theory 10.2: Monoid in the category of endofunctors. Video. <https://www.youtube.com/watch?v=GmgoPd7VQ9Q>. Katsottu 09. 09. 2024.

53 Wikipedia. 2024d. Monad (functional programming). Verkkoaineisto. <https://en.wikipedia.org/wiki/Monad_(functional_programming)>. Luettu 30. 09. 2024.

54 Stack Overflow. 2015. Is a collection with flatMap a monad? Verkkoaineisto. <https://stackoverflow.com/questions/27750046/is-a-collection-with-flatmap-a-monad>. Luettu 30. 09. 2024.

55 Are arrays monads in modern JavaScript? 2021. <https://stackoverflow.com/questions/65905682/are-arrays-monads-in-modern-javascript>. Luettu 13. 10. 2024.

56 guevara. 2024. Why JavaScript promises aren't technically monads. Verkkoaineisto. <https://github.com/guevara/read-it-later/issues/11481>. Luettu 31. 08. 2024.

57 Archibald, Jake & Pennanen, Arttu. Lokakuu 2024a. Keskustelu. Github. Verkkoaineisto. <https://gist.github.com/pennane/dd0bf878bf9cb1371f8c2916fb35c01

58 Haskell Wiki. 2024a. Monad laws. Verkkoaineisto. <https://wiki.haskell.org/Monad_laws>. Luettu 13. 10. 2024.

59 Holvikari, Antti & Lahti, Esko. 2021. Category Theory for the Non-PhD - and What to Use It For. Reaktor. Podcast. <https://podcast.reaktor.com/forkpullmergepush/category-theory-for-the-non-phd-and-what-to-use-it-for/>. Kuunneltu 12. 09. 2024.

60 Clingonboy et al. 2020. Poor question. Verkkoaineisto. <https://github.com/chrokh/fp-games/issues/6>. Luettu 31. 08. 2024.

61 Hickey, Rich. 2018. Maybe Not. ClojureTV. Video. <https://www.youtube.com/watch?v=YR5WdGrpoug>. Katsottu 14. 09. 2024.

62 Ramda. 2024a. Ramda. GitHub repositorio. Verkkoaineisto. <https://github.com/ramda/ramda>. Luettu 14. 09. 2024.

63 Sanctuary. 2024. Sactuary. GitHub repositorio. Verkkoaineisto. <https://github.com/sanctuary-js/sanctuary>. Luettu 14. 09. 2024.

64 Hofmann-Hicks, Ian. 2024. crocks. GitHub repositorio. Verkkoaineisto. <https://github.com/evilsoft/crocks>. Luettu 14. 09. 2024.

65 Canti, Giulio. 2024. fp-ts. GitHub repositorio. Verkkoaineisto. <https://github.com/gcanti/fp-ts>. Luettu 14. 09. 2024.

66 BuiltWith. 2024. Prototype JavaScript Framework Usage Trends. Verkkoaineisto. <https://trends.builtwith.com/javascript/Prototype>. Luettu 31. 08. 2024.

67 Archibald, Jake. 2023. The case against self-closing tags in HTML. Verkkoaineisto. <https://jakearchibald.com/2023/against-self-closing-tags-in-html/>. Luettu 14. 09. 2024.

68 Harband, Jordan et al. 2023. Why just on iterators and not on arrays? TC39. Verkkoaineisto. <https://github.com/tc39/proposal-joint-iteration/issues/1>. Luettu 31. 08. 2024.

69 MDN Web Docs. 2024h. Promise. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Luettu 30. 09. 2024.

70 CallBackHell. 2024. Callback Hell. <http://callbackhell.com/>. Luettu 30. 09. 2024.

71 Ramda. 2024b. Ramda Documentation. Versio v0.30.1. <https://ramdajs.com/docs/>. Luettu 22. 09. 2024.

72 Immutable.js 2024. <https://immutable-js.com/>. Luettu 10. 10. 2024.

73 Hickey, Rich. 2009. Persistent Data Structures and Managed References. Verkkoaineisto. <https://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey/>. Luettu 10. 10. 2024.

74 Turner-Trauring, Itamar. 2020. Copying Data is Wasteful, Mutating Data is Dangerous. Verkkoaineisto. <https://pythonspeed.com/articles/minimizing-copying/>. Luettu 10. 10. 2024.

75 Feldman, Richard. 2016. Making Impossible States Impossible. elm-conf. Video. <https://www.youtube.com/watch?v=IcgmSRJHu_8>. Katsottu 20. 08. 2024.

76 Haskell Wiki. 2024b. All About Monads. Verkkoaineisto. <https://wiki.haskell.org/All_About_Monads>. Luettu 13. 10. 2024.

77 Archibald, Jake & Pennanen, Arttu. Lokakuu 2024b. Keskustelu. X (Twitter).

78 Buse, Raymond P.L. & Weimer, Westley R. 2010. ”Learning a Metric for Code Readability”. IEEE Transactions on Software Engineering 36.4, s. 546–558.

79 Dijkstra, Edsger W. 1976. A Discipline of Programming. Prentice Hall PTR.

80 Rossum, Guido van. 2008. The Fate of reduce() in Python 3000. Verkkoaineisto. <https://www.artima.com/weblogs/viewpost.jsp?thread=98196>. Luettu 15. 10. 2024.

81 GitHub. 2024. Fantasy Land spesifikaation edistäjät. Verkkoaineisto. <https://github.com/fantasyland/fantasy-land/graphs/contributors>. Luettu 31. 08. 2024.

82 The Elixir Programming Language 2024. <https://elixir-lang.org/>. Luettu 16. 10. 2024.

83 The Go Programming Language 2024. <https://go.dev/>. Luettu 16. 10. 2024.

84 Bedeian, Arthur G. Huhtikuu 2016. ”A note on the aphorism “there is nothing as practical as a good theory””. Journal of Management History 22, s. 236–242.