Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Saini Patala

Funktionaalinen ohjelmointi JavaScriptissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

13.5.2020

Tekijä Otsikko Sivumäärä Aika	Saini Patala Funktionaalinen ohjelmointi JavaScriptissä 82 sivua 13.5.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander

Tämän opinnäytetyön tarkoituksena on selvittää, mitä funktionaalinen ohjelmointi on ja miten se soveltuu JavaScript-ohjelmointikielelle. Samalla tarkoituksena on syventää ymmärrystä myös itse JavaScriptiin kielenä ja oppia hyödyntämään sitä paremmin ja monipuolisemmin funktionaalisien periaatteiden avulla.

Tavoitteena on kirjoittaa kokonaisuus, jossa selvitetään, mitä funktionaalinen ohjelmointi on, miten sitä voidaan soveltaa JavaScriptiin ja mitä hyötyä siitä voi olla JavaScriptillä ohjelmoitaessa.

Opinnäytetyössä käydään läpi funktionaalisen ohjelmoinnin periaatteita. Työssä esitellään, miten dataa käsitellään funktionaalisesti JavaScriptissä, miten funktioista saadaan puhtaita ja miten niitä voidaan hyödyntää mahdollisimman monipuolisesti käyttämällä niitä ensimmäisen luokan kansalaisina. Työssä tutustutaan funktionaaliselle ohjelmoinnille tyypillisiin map-, filter- ja reduce-funktioihin sekä esitellään rekursio. Lopuksi selvitetään vielä, miten funktioita voidaan ketjuttaa ja yhdistellä luoden monimutkaisempia toimintokokonaisuuksia sekä mitä hyötyä funktori- ja monadi-suunnittelumallit tuovat funktioiden ketjutukseen.

Johtopäätöksenä todetaan, että funktionaalinen ohjelmointi perustuu pitkälti datan muuttumattomuuteen, puhtaisiin funktioihin ja sivuvaikutuksettomuuteen, funktioiden käsittelyyn ensimmäisen luokan kansalaisina ja funktioiden yhdistelyyn luoden monimutkaisempia toimintoja. Yleisesti ottaen JavaScript sopii kielenä hyvin funktionaaliseen ohjelmointiin. Datan muuttumattomuus ja puhtaat funktiot ovat toteutettavissa JavaScriptillä. Funktioita kohdellaan ensimmäisen luokan kansalaisina JavaScriptissä, mikä mahdollistaa funktioiden monipuolisen käsittelyn. Funktionaaliset periaatteet menevät myös hyvin yksi yhteen yleisesti hyvinä pidettyjen ohjelmointitapojen kanssa, joten jo pelkästään funktionaalisten periaatteiden noudattaminen voi tehdä ohjelmasta helpommin luettavaa, virheettömämpää ja uudelleenkäytettävämpää.

Avainsanat	JavaScript, funktionaalinen ohjelmointi

Author Title Number of Pages Date	Saini Patala Functional Programming in JavaScript 82 pages 13 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer

The purpose of this thesis is to find out what functional programming is and how its principles can be applied to JavaScript programming language. At the same time the purpose is to deepen the understanding of JavaScript as a language and learn to use it more versatilely by using functional programming principles.

The goal of the thesis is to get the big picture of what functional programming is, how it can be applied to JavaScript and what benefits it might bring to programming in JavaScript.

This work goes through the principles of functional programming. It shows how to handle data in a functional manner, how to make functions pure and how functions benefit from being first class citizens. The typical functional features map, filter and reduce are demonstrated in JavaScript as well as recursion. Finally, function composition and chaining of functions are examined. It is also demonstrated how small functions can be used together to create more complex functionalities and what benefits functor and monad design patterns bring to function chaining.

As a conclusion, functional programming is largely based on immutability, pure functions, functions having no side effects, functions being treated as first class citizens and the composition of functions to build more complex functionalities. In general, JavaScript programming language suits well for functional programming. Immutability and pure functions can be implemented in JavaScript. Functions are treated as first class citizens, which makes it possible to use functions in multiple manners. Functional principles also go well hand in hand with general good practices of programming, so following the principles of functional programming can make code easier to understand, less error prone and more reusable.

Keywords	JavaScript, functional programming

Sisällys

1 Johdanto

JavaScriptiä on mahdollista ohjelmoida joustavasti erilaisten ohjelmointiparadigmojen mukaan, koska JavaScriptissä ei ole tiukkoja sääntöjä kielen käytöstä. JavaScript taipuu siten käyttötarpeiden mukaan imperatiivisesta ja proseduaalisesta ohjelmoinnista olio-ohjelmointiin kuin myös funktionaaliseenkin ohjelmointiin. [Fogus 2013: Forewords by Jeremy Ashkenas.]

Vaikka funktionaalinen ohjelmointi on ollut käsitteenä olemassa lähes koko ohjelmoinnin historian alusta saakka, on se vasta viime vuosina noussut enemmän esille. Funktionaalisen ohjelmoinnin tuki on kasvanut niin kielitasolla kuin kirjastojen ja sovellusviitekehysten myötä. [Simpston 2017: Chapter 1.]

Funktionaalinen ohjelmointi lisää parhaimmillaan koodin luettavuutta, vaikka syntaksi voi näyttää alkuun vieraalta. Funktionaalinen ohjelmointi on deklaratiivista koodia, eli siinä määritellään haluttu tulos sen sijaan, että ohjelmoitaisiin miten-näkökannasta. Tästä syystä koodi on nopea ymmärtää, koska pelkkä lopputuloksen määrittely kertoo tarpeeksi, kun taas imperatiivisessa koodissa toteutus täytyy käydä rivi riviltä läpi, jotta sen voi ymmärtää. [Simpston 2017: Chapter 1.]

Tässä työssä käydään ensin läpi funktionaalisen ohjelmoinnin keskeisiä käsitteitä sekä sitä, miten JavaScript soveltuu funktionaaliseksi kieleksi. Sitten siirrytään selvittämään, miten dataa käsitellään funktionaalisesti. Seuraavaksi käsitellään yleisesti funktioita ja selvitetään, miten niitä voidaan hyödyntää mahdollisimman monipuolisesti JavaScriptissä. Funktionaalisen ohjelmoinnille tyypilliset map-, filter- ja reduce-funktiot esitellään ja selvitetään, kuinka rekursiota käytetään JavaScriptissä. Lopuksi käydään läpi, miten funktioita voidaan koostaa ja ketjuttaa JavaScriptissä luoden deklaratiivista funktionaalista ohjelmaa.

2 Funktionaalinen ohjelmointi

Funktionaalisen ohjelmoinnin pohjana on lambdakalkyyli, eli laskennan malli, jonka kehitti Alonzo Church 1930-luvulla. Tämä malli oli Churchin vastaus filosofiseen kysymykseen, joka tunnetaan nimellä Entscheidungsproblem, voidaanko kaikki

universaalilla kielellä muodostetut ongelmat ratkaista? Ratkaistakseen ongelman Churchin täytyi määritellä käsite laskettavalle funktiolle. Lambdakalkyyli perustuu puhtaaseen abstraktioon, ja vaikka se on hyvin yksinkertainen, se on silti tehokas. Lambdakakyyli perustuu lausekkeiden tai termien käyttöön, joita ovat muuttujat, abstraktiot sekä sovellukset. [Barendsen 2000: sivu 5; Michaelson 2011: 1.9 **λ** Calculus, 2.1 Abstraction.]

Lambda-lauseke (Lambda expression) voi koostua seuraavista termeistä:

- Muuttujalla (variable) identifioidaan abstraktio.
- Abstraktio (abstraction) on abstraktio lambdalausekkeesta, eli funktio.
- Sovelluksella (application) erikoistetaan abstraktio antamalla abstrahoidulle asialle arvo, eli funktiolle annetaan argumentti. [Michaelson 2011: 2.4 **λ** Expression.]

Lambda-kalkyylissä lambda-merkki, eli λ, merkitsee funktion alkua ja, se esiintyy aina muuttujan esittelyä ennen. Funktio koostuu head- sekä body-osioista. Abstraktiossa λx.x head-osioon kuuluu pisteen vasemmalla puolella olevat λx. Pisteen oikealla puolella oleva lauseke kuuluu body-osioon. Headissä esiintyvä kirjain, esimerkiksi, x, y tai z, esittää muuttujaa, jonka arvolla korvataan funktion bodyssä esiintyvät saman nimiset muuttujat. Näitä headissä esiintyviä muuttujia kutsutaan sidotuiksi muuttujiksi. Ne toimivat argumentteina funktiolle. [Michaelson 2011: 2.4 **λ** Expression.]

Esimerkiksi termi λx.x on funktioabstraktio, joka kuvaa identiteettifunktiota. Identiteettifunktiota on sellainen funktio, joka palauttaa saman arvon, jonka se sai parametriksi. Lambda-merkin jälkeinen kirjain, tässä tapauksessa x, on muuttuja, jonka funktio saa argumentiksi. Kun funktiota sovelletaan, bodyssä esiintyvän määritelmän saman nimiset muuttujat korvataan saadulla parametrilla. JavaScriptissä vastaava funktio olisi x => x. Toinen yksinkertainen abstraktio on λx.y, joka palauttaa y:n riippumatta x:n arvosta. JavaScriptissä se esitetään x => y -muodossa.

Identiteettifunktiota voidaan soveltaa syntaksilla (λx.x)y, jolloin identiteettifunktio sovelletaan y:lle. Funktio saa parametriksi y:n ja korvaa kaikki body:n määrittelyssä

ilmenevät x:t y:llä. Funktion bodyssä on vain yksi muuttuja, ja se on x, eli tämä korvataan y:llä. Funktion body on näin ratkaistu ja tulos on y.

2.1 Pääpiirteet

Funktionaalinen ohjelmointi on deklaratiivista. Se pyrkii ilmaisemaan ohjelman logiikkaa ja sitä, mitä halutaan saavuttaa, eikä kuvaa ohjelman ohjausvuota askel askeleelta, kuten imperatiivinen ohjelmointi, mihin kuuluu esimerkiksi olio-ohjelmointi. Ohjelman kulku perustuu funktiokutsuihin ja rekursioon perinteisten silmukoiden ja ehtolauseiden sijaan. [Mantyla 2015: Chapter 2. Fundamentals of Functional Programmin.]

Funktionaaliselle ohjelmoinnille tärkeitä piirteitä ovat funktioiden ja taulukkojen käyttö. Pienistä funktioista koostetaan suurempia ohjelmakokonaisuuksia. Funktionaalisen ohjelmoinnin funktiot ovat puhtaita, eikä niillä ole sivuvaikutuksia. Funktioita voidaan myös määritellä muuttujiin, niitä voidaan antaa parametreina sekä niitä voidaan palauttaa paluuarvoina. [Antani ym. 2016: Module 2. 6. Functional Programming. Immutability; Mantyla 2015: Chapter 2. Working with functions. Pure functions.]

Olemassa olevaa dataa ei muuteta funktionaalisessa ohjelmoinnissa, vaan se pysyy muuttumattomana. Siten ohjelmassa ei myöskään ole globaalia muuttuvaa tilaa. Koska funktiot ovat puhtaita, eikä ohjelmassa ylläpidetä globaalia muuttuvaa tilaa, ohjelman osien suoritusjärjestys ei ole yhtä tärkeä kuin se on imperatiivisessa ohjelmoinnissa, missä funktioiden tuloste voi riippua funktion ulkopuolella ylläpidettävästä tilasta. Funktionaalinen tilaton lähestymistapa sopii hyvin monisäikeiselle ohjelmoinnille, koska ei ole yhteistä tilaa, josta ohjelma on riippuvainen tai jota useampi säie manipuloisi. [Antani ym. 2016: Module 2. 6. Functional Programming. Functional functions are side-effect-free.]

2.2 JavaScript ja funktionaalinen ohjelmointi

On olemassa ohjelmointikieliä, jotka on rakennettu funktionaalisiksi kuten Lisp, Haskell ja Clojure. JavaScript ei sen sijaan ota kantaa, mitä paradigmaa ohjelmoidessa

noudatetaan, vaan se taipuu moneen ohjelmointityyliin. Siten myös funktionaalista ohjelmointia voidaan toteuttaa eri tavoin JavaScriptissä. Funktionaalisen ohjelmoinnin periaatteita ei tarvitse välttämättä noudattaa täysin koko ohjelmassa vaan sitä voidaan myös hyödyntää imperatiivisen ohjelman osana. Koska JavaScript ei itsessään vaadi funktionaalisen paradigman noudattamista, jää sen noudattaminen ohjelmoija toteutettavaksi. [Antani ym. 2016: Module 2, Chapter 6; Mantyla 2015: Chapter 2.]

2.2.1 Apukirjastot

JavaScriptille on olemassa monia kirjastoja, jotka sisältävät hyödyllisiä funktioita jonkin tietyn asian saavuttamiseksi. Funktionaalinen ohjelmointi ei ole mikään poikkeus tässä asiassa, vaan JavaScriptille löytyy kirjastoja, jotka auttavat funktionaalisen ohjelman kirjoittamisessa. Muutamia esimerkkejä tällaisista kirjastoista ovat Ramda ja Folktale. On myös monia pienempiä kirjastoja, jotka tarjoavat yksittäisiä funktionaalisia toteutuksia, kuten esimerkiksi vain muuttumattoman tietorakenteen. Tällaisista kirjastoista hyvä esimerkki ovat Icepick ja Immutable.js. [Folktale; Icepick; Immutable: Immutable Collections for JavaScript; Ramda.]

2.2.2 JavaScriptiksi kääntyvät kielet

On myös monia funktionaalisia ohjelmointikieliä kuten esimerkiksi ClojureScript, Elm, Reason ja Scala.js, jotka voidaan kääntää JavaScriptiksi. Esimerkiksi Elm on täysin funktionaalinen ohjelmointikieli, jonka syntaksi on myös hyvin erilaista JavaScriptiin verrattuna. Se kuitenkin käännetään JavaScriptiksi, jotta selain voi ajaa sitä. [ClojureScript; Elm; Reason; Scala.js.]

3 Data

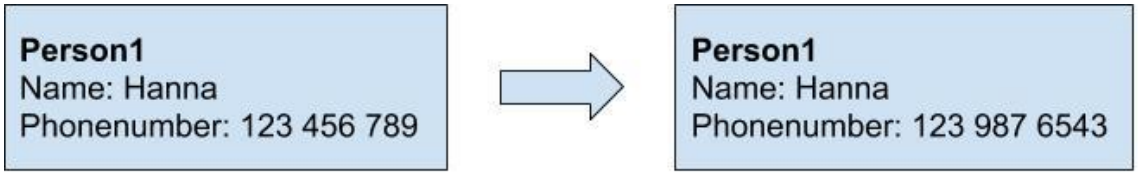
Funktionaalisen ohjelmoinnin lähestymistapa dataan poikkeaa olio-ohjelmoinnista, jossa tieto jäsennellään olioihin, jotka puolestaan pitävät tilaa yllä. Olio-ohjelmoinnissa olioiden tilaa muutetaan ohjelman ajon aikana. Ohjelma perustuu olioiden tilaan ja tilan muutoksiin, minkä vuoksi ohjelma on hyvin riippuvaista ohjelman tilasta. Erilaiset

operaatiot ovat riippuvaisia vallitsevasta ympäristöstään ja ympäristössä olevien olioiden tiloista. Muutokset olioiden tiloissa vaikuttavat operaatioiden lopputuloksiin.

Olio-ohjelmointi sopii tilanteisiin, jossa datasta on monia instansseja, mutta operaatioita on vähemmän. Funktionaalinen ohjelmointi sopii taas tilanteisiin, joissa pitää laskea paljon asioita datan pohjalta. Se perustuu muuttujiin, funktioihin ja operaatioihin. Funktionaalisessa ohjelmoinnissa olemassa olevaa dataa ei muuteta, vaan se pysyy alkuperäisessä muodossa, ja sen pohjalta luodaan ja lasketaan uutta dataa. Dataa ei siis koskaan manipuloida kuten olio-ohjelmoinnissa. Sitä, että alkuperäinen data pysyy koskemattomana, kutsutaan muuttumattomuudeksi. Se on yksi keskeisimmistä funktionaalisen ohjelmoinnin periaatteista.

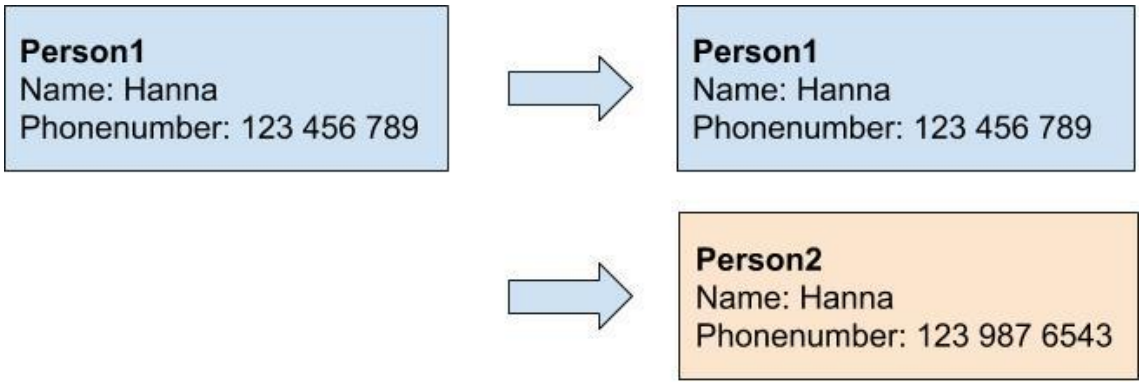
3.1 Muuttumattomuus (immutability)

Funktionaalisen ohjelmoinnin peruspilarina toimii muuttumattomuuden periaate. Tietorakenteet pysyvät muuttumattomina niiden luonnin jälkeen. Seuraavassa kuvissa 1 ja 2 havainnollistetaan olio-ohjelmoinnin ja funktionaalisen ohjelmoinnin eroa, kun halutaan suorittaa jokin operaatio olemassa olevan datan pohjalta.



Kuva 1. Olio-ohjelmoinnissa olion tietoa muutetaan suoraan

Kuva 1 havainnollistaa, mitä tapahtuu, kun olion tietoa muutetaan olio-ohjelmoinnissa. Olion instanssi pysyy samana, vaikka sen Phonenumber-arvoa muutetaan.



Kuva 2. Funktionaalisessa ohjelmoinnissa luodaan uusi olio, johon muutokset tehdään

Kuvassa 2 mallinnetaan tiedon muuttamista funktionaalisessa ohjelmoinnissa. Alkuperäinen olio Person1 pysyy muuttumattomana, mutta sen rinnalle luodaan uusi olio, Person2, joka kopioi alkuperäisen olion ominaisuudet ja arvot, ja muutokset toteutetaan uuteen olioon sen luonnin yhteydessä.

JavaScript ei tarjoa suoraan muuttumattomuutta, paitsi primitiiveille tietotyypeille eli numeroille, merkkijonoille, totuusarvoille, sekä null-, undefined- ja symbol-arvoille. Primitiivi tietotyyppi on ennalta määritelty tietotyyppi, jota ohjelmointikieli tukee. Muuttujia luodaan käyttäen primitiivisiä tietotyyppejä. Kaikki muut JavaScriptin tietotyypit ovat olioita, kuten taulukko tai funktio. [Frederico Kereki 2017: Ensuring Purity - Immutability.]

Vaikka JavaScriptissä on mahdollista määritellä const-tyyppinen muuttuja, joka viittaa aina samaan olioon tai primitiiviseen arvoon, ei se takaa muuttumattomuutta. Esimerkiksi, jos const-arvolla alustetaan taulukko, on totta, että viittaus tähän taulukkoon ei tule enää muuttumaan, sillä JavaScript ei salli sitä. JavaScript ei kuitenkaan estä itse taulukon sisällön muuttumista, sillä kyseessä ei ole primitiivinen arvo. Vaikka viite taulukkoon pysyy samana koko ohjelman aikana, taulukon sisältöä voidaan muuttaa vapaasti eikä tämä vastaa funktionaalisen ohjelmoinnin muuttumattomuutta. [Kereki 2017: Ensuring Purity - Immutability.]

```
const constarr = [1, 2, 3]
let secondarr = [4, 5, 6]

console.log(constarr) // [1, 2, 3]
constarr.push(2)
console.log(constarr) // [1, 2, 3, 2]
```

```
constarr = secondarr // TypeError: Assignment to constant variable.
```

Esimerkkikoodi 1. Const-muuttuja JavaScriptissä

Esimerkkikoodissa 1 asetetaan taulukko [1, 2, 3] const-muuttujaan nimeltä constarr. Määritellään myös toinen secondarr-taulukko arvoilla [4, 5, 6]. Ensin tulostetaan constarr-taulukon arvo, joka on [1, 2, 3]. Sen jälkeen siihen yritetään lisätä luku 2. Nyt constarr-taulukon arvo on [1, 2, 3, 2], eli sen arvoa pystyy manipuloimaan. Seuraavaksi koitamme vaihtaa constarr:in arvoksi secondarr-taulukon, mutta tämä aiheuttaa virhetilanteen ”TypeError: Assignment to constant variable.”. Huomaamme siis, että const-muuttuja ei anna vaihtaa viittausta olioon, mutta olion sisällä olevia arvoja voidaan muuttaa. [Antani ym. 2016: Module 2, Chapter 6, Immutability.]

JavaScriptiä on mahdollista ohjelmoida niin, että tietorakenteet pysyvät muuttumattomina, mutta koska se ei ole JavaScriptin vaatima ominaisuus, täytyy ohjelmoijan huolehtia siitä itse. Apuna voidaan käyttää myös apukirjastoja, kuten immutable.js tai ramda, jotka tarjoavat muuttumattomia tietorakenteita. Apufunktioita voidaan luoda myös itse, joiden avulla dataa muutetaan ja jotka pitävät huolen, että alkuperäinen data ei muutu, vaan ne palauttavat uuden kopion datasta muuttuneilla arvioilla. Tämä ratkaisu ei estä suoraa tiedon muuttamista vaan tarjoaa rajapinnan muuttumattomalle datan muuttamiselle. [Frederico Kereki 2017: Ensuring Purity - Immutability.]

Olioita voidaan jäädyttää käyttämällä Object.freeze()-funktiota, joka estää olion rakenteen tai sen sisällön muuttamisen. Se ei kuitenkaan ole turvallinen ratkaisu sellaisenaan, sillä se ei tehoa olion sisällä oleviin olioihin tai taulukoihin. Se pitää vain huolen, että viite niihin pysyy ennallaan, mutta niiden sisältö on muutettavissa. Tämä tarkoittaa sitä, että kaikki olion tasot on jäädytettävä itse ja se voidaan toteuttaa rekursiivisesti. [Frederico Kereki 2017: Ensuring Purity - Immutability.]

```
const cart = {
  id: 2394389,
  creationTime: "Wed Nov 20 19:36:19 EET 2019",
}

cart.id = 123456

console.log(cart.id) // 123456
```

Esimerkkikoodi 2. Olion arvon muuttaminen

Esimerkkikoodissa 2 demonstroidaan, kuinka olion arvoja voidaan muuttaa, vaikka olio onkin määritelty const-muuttujaksi. Ostoskorin id-arvo muutetaan sen luonnin jälkeen, ja vaikka olion viite pysyy samana, sen id-arvo muuttuu.

```
const cart2 = Object.freeze({
  id: 2394389,
  creationTime: "Wed Nov 20 19:36:19 EET 2019",
})

cart2.id = 123456

console.log(cart2.id) // 2394389
```

Esimerkkikoodi 3. Olion muuttaminen lukitaan Object.freeze-funktiolla.

Esimerkissä 3 olio syötetään Object.freeze-funktiolla, joka estää olion arvojen muuttamisen. Ostoskorin id:n muuttaminen ei enää onnistu kuten aikaisemmassa esimerkissä 2. Ostoskorin id-arvo pysyy alkuperäisenä.

```
const cart3 = Object.freeze({
  id: 2394389,
  creationTime: "Wed Nov 20 19:36:19 EET 2019",
  products: ["apple", "cookies", "cheese"],
})

console.log(cart3) // ['apple', 'cookies', 'cheese']

cart3.products.push("cherries")

console.log(cart3) // ['apple', 'cookies', 'cheese', 'cherries']
```

Esimerkkikoodi 4. Sisäkkäisen olion muuttaminen

Esimerkkikoodi 4:ssä jäädytetyn olion sisällä olevaa taulukkoa pystyy muuttamaan, sillä Object.freeze-funktio ei syväjäädytä oliota. Kaikki sisäkkäiset oliot pitää jäädyttää rekursiivisesti, jos tietorakenteen muuttumattomuus halutaan taata.

```
const cart = {
  id: 2394389,
  creationTime: "Wed Nov 20 19:36:19 EET 2019",
  products: [
    { name: "apple", price: 1 },
    { name: "cookies", price: 3 },
```

```
    { name: "cheese", price: 2 },
  ],
  customer: {
    id: 4952,
  },
}

const freezeObject = obj => {
  Object.keys(obj).forEach(key => {
    Object.freeze(obj[key])

    if (typeof obj[key] === "object") {
      freezeObject(obj[key])
    }
  })
}

freezeObject(cart)
cart.products.push({ name: "cherries", price: 5 }) // TypeError: Cannot add
property 3, object is not extensible
cart.customer.id = 3 // ei vaikutusta
```

Esimerkkikoodi 5. Esimerkki koko olion jäädyttämisestä rekursiivisesti

Esimerkkikoodissa 5 käytetään rekursiivista funktiota freezeObject olion syväjäädyttämiseen. Tämä tarkoittaa sitä, että kaikki olion alioliot jäädytetään myös, jotta siitä tulee täysin muuttumaton. Rekursiivinen funktio käy läpi olion kaikki avainarvoparit, ja mikäli arvon tyyppi on olio, täytyy se jäädyttää erikseen. Tämän operaation jälkeen cart-olion sisällä olevia olioita, kuten products-taulukkoa tai customer-oliota, ei voi enää muuttaa. Kun products-taulukkoon yritetään viedä uusi olio { name: "cherries", price: 5 }, saadaan virhe, joka ilmoittaa, että olio ei ole laajennettavissa. Kun customer-olion id-arvoa yritetään muuttaa, sillä ei puolestaan ole mitään vaikutusta, eikä id:n arvo muutu. Tällä tavoin voidaan taata täysin muuttumaton olio Object.freeze-metodia hyödyntäen.

3.1.1 Hyödyt

Verrattuna olio-ohjelmointiin muuttumattomuus on varsin vieras konsepti. Olio-ohjelmoinnissa tietorakenteita muutetaan olion metodeilla. Tietoa luetaan ja muutetaan yleensä useaan otteeseen ohjelman ajon aikana. Kun monet operaatiot ovat riippuvaisia ohjelman tai olion tilasta, täytyy tilaa muuttaessa tiedostaa, mitä operaatio oikeastaan tekee ja minkälaisia vaikutuksia sillä on itse muutettuun dataan tai muuhun ohjelmaan. Myös operaatioiden suoritusjärjestyksellä voi olla vaikutusta lopputulokseen tai voi olla tilanteita, joissa jotain operaatiota ei saisi tehdä. Ohjelmassa voi olla monia

reunatilanteita, joista ohjelmoijan sekä ohjelman tulisi tietää. Ohjelma on altis virheille, koska kokonaisuutta on vaikeampi hallita, ja sivuvaikutuksia on joskus mahdoton ennustaa.

Muuttumattomuus takaa sen, että odottamattomia sivuvaikutuksia ei tapahdu. Mikä tahansa operaatio datalle on turvallista toteuttaa, koska operaatio palauttaa vain uutta dataa eikä alkuperäinen data muutu lainkaan.

Muuttumattomuus pitää huolen, että monisäikeisen ohjelman käsittelemä data ei muutu kesken kaiken muiden säikeiden sivuvaikutuksesta. JavaScript ei kuitenkaan käytä kuin yhtä säiettä ja vain yksi prosessi voi tapahtua kerrallaan. Sen sijaan JavaScript on asynkroninen kieli, mikä tarkoittaa, että ohjelman suoritusjärjestys voi muuttua. Prosessi voidaan siirtää sivuun, suorittaa muita prosesseja välissä ja jatkaa sivuun siirrettyä prosessia myöhemmin. Siitä syystä muuttumattomuus on silti tärkeää JavaScriptissä.

Muuttumattomuus on suuressa keskiössä myös esimerkiksi Reactissa, joka on Facebookin kehittämä JavaScript-kirjasto web-käyttöliittymien kehittämiseen. Reactissa ei saa muuttaa suoraan ohjelman tilaa ylläpitävää oliota, vaan siitä on tehtävä muuttumattomuutta toteuttaen uusi kopio, jossa on muutetut tiedot. Täten React voi tarkistaa, mikäli tilaa pitävän olion viite on muuttunut ja siten se tietää, että siitä tilasta riippuvaisia komponentteja on päivitettävä. Vertaamalla pelkkää olion viitettä vältytään raskailta operaatioilta, jossa verrattaisi suuria olioita tai tauluja toisiinsa arvo kerrallaan. Hyödyntämällä muuttumattomuutta React optimoi sivun renderöintiä niin, että vain tarvittavat palaset päivittyvät ja vain silloin, kun olion viite muuttuu. [Reactjs. Docs. Rendering Elements.]

3.1.2 Muuttumattomuus testauksessa

Olio-ohjelmoinnissa operaatioiden testaus vertaamalla ohjelman muuttunutta tilaa on siinä mielessä hankalaa, että ohjelman edellinen tila ylikirjoitetaan muutoksen tapahtuessa. Jotta edellistä ja nykyistä tilaa voitaisiin verrata, täytyy tilasta ottaa aina kopio ennen suoritettavaa operaatiota.

Funktionaalisen ohjelmoinnin muuttumattomuus tekee testien toteuttamisesta huomattavasti suoraviivaisempaa, sillä alkuperäinen data säilyy muuttumattomana oletuksena ja sitä voidaan aina verrata uuteen luotuun dataan. Seuraavassa esimerkissä verrataan funktioiden testausta sekä havainnollistetaan muuttumattomuuden puuttumisesta koituvia ongelmia testatessa funktioita. Molemmissa esimerkeissä testataan kahta funktiota, jotka tekevät näennäisesti saman asian. Toinen funktio vähentää player-olion score-arvosta annetun value-luvun, ja toinen lisää siihen annetun value-luvun.

Ensimmäinen esimerkki toteuttaa muuttumattomuutta, ja siinä testataan kahta funktiota.

```
let player = {
  name: "someplayer",
  score: 1304,
}

const increaseScoreByImmutable = (player, value) => {
  return { ...player, score: player.score + value }
}

const decreaseScoreByImmutable = (player, value) => {
  return { ...player, score: player.score - value }
}

const testIncreaseScoreByImmutable = player => {
  return increaseScoreByImmutable(player, 10).score - player.score === 10
}

const testDecreaseScoreByImmutable = player => {
  return decreaseScoreByImmutable(player, 10).score - player.score === -10
}

console.log(testIncreaseScoreByImmutable(player)) // true
console.log(testDecreaseScoreByImmutable(player)) // true
```

Esimerkkikoodi 6. Esimerkkikoodissa testataan, että increaseScoreByImmutable- ja decreaseScoreByImmutable-funktiot palauttavat odotetunlaisen tuloksen

Molemmat esimerkkikoodi 6:n testattavista funktioista tekevät uuden kopion player-oliosta muuttaen vain uuden kopion score-arvoa ja palauttaa tämän. Testifunktioille testIncreaseScoreByImmutable ja testDecreaseScoreByImmutable annetaan alkuperäinen player-olio, jonka se välittää testattaville funktiolle. Funktioiden palauttamaa uutta kopiota voidaan verrata alkuperäiseen olioon ja tarkistaa, että palautunut tulos on odotetunlainen. Molemmat testit palauttavat totuusarvon ”true”, eli uuden kopion arvot ovat muuttuneet odotetunlaisesti ja testit menivät läpi. Jokaiselle

testille parametrina annettu player-olio on sama, ja se pysyy koko testauksen ajan alkuperäisessä muodossa. Muut testit eivät vaikuta toisten testien parametriksi saamaan player-olioon, joten testien ajojärjestyksellä ei ole merkitystä.

```
class Player {
  constructor(name, score) {
    this.name = name;
    this.score = score;
  }

  increaseScoreBy(value) {
    this.score = this.score + value;
    return this;
  }

  decreaseScoreBy(value) {
    this.score = this.score - value;
    return this;
  }
}

let player = new Player("someplayer", 1304);

const testIncreaseScoreBy = player => {
  let oldPlayerState = { ...player };
  return player.increaseScoreBy(10) - oldPlayerState.score === 10;
};

const testDecreaseScoreBy = player => {
  let oldPlayerState = { ...player };
  return player.decreaseScoreBy(10) - oldPlayerState.score === -10;
}

console.log(testIncreaseScoreBy(player)) // true
console.log(testDecreaseScoreBy(player)) // true
```

Esimerkkikoodi 7. Testaus olio-ohjelmoinnilla toteutetussa ratkaisussa

Olio-ohjelmoinnissa testaus ei toimi aivan samalla lailla. Alkuperäinen olio tulee muuttumaan, joten sitä on verrattava olion edelliseen tilaan tai olion edellisen tilan arvoon. Suoralta kädeltä vertaaminen ei onnistu ilman, että luodaan manuaalisesti kopio alkuperäisestä oliosta tai otetaan talteen muutettava arvo.

Esimerkissä 7 player-oliosta otetaan kopio muuttujaan oldPlayerState. Olion metodi palauttaa uuden olion tilan. Vanhaa olion tilaa verrataan alkuperäisen player-olion nykytilaan. Saamme saman lopputuloksen, mutta se vaatii ylimääräisen askeleen. Funktionaalisessa ratkaisussa voi suoraan verrata alkuperäistä oliota funktion palauttamaan olioon. Olio-ohjelmoinnissa joudutaan luomaan ylimääräinen muuttuja.

Funktionaalisessa ohjelmoinnissa uusi kopio tehdään tietysti myös, mutta se tapahtuu varsinaisen editoivan funktion toimesta.

Tässä testausesimerkissä olio-ohjelmointi ei varsinaisesti tuota hankaluuksia. Mutta niitä voisi syntyä, mikäli testit olisivat riippuvaisia olion tietystä tilasta. Jokainen testi muokkaa olion tilaa, joten sen tila testin ajon hetkellä riippuu testien ajojärjestyksestä. Mikäli testit olisivat riippuvaisia olion tilasta, pitäisi sen tila aina palauttaa alkutilanteeseen testien välissä tai luoda aina uusi olio ennen jokaista testiä. Suurissa testikokonaisuuksissa tästä voi syntyä ongelmia, joita on vaikea selvittää jälkikäteen.

3.1.3 Historian tallennus

Historian tallennuksella tarkoitetaan sitä, että koko ohjelman, tai sen osan tila tallennetaan muistiin muutosten tapahtuessa. Tämä tilanne vastaa pitkälti edellä mainittua ohjelman testausta. Olio-ohjelmoinnissa historian tallennus ei ole suoraviivaista paradigman kanssa. Ohjelmassa kutsutaan metodeja ja funktioita, joilla on sivuvaikutuksia ja joiden ajojärjestyksellä voi olla merkitystä lopputulokselle. Ohjelman tilaa voidaan manipuloida suoraan, jolloin sen vanha tila tulee ylikirjoitetuksi.

Funktionaalisen ohjelmoinnin muuttumattomuuden toteuttaminen helpottaa historian tallennusta, koska muutoksia tehdessä alkuperäinen tieto pysyy muuttumattomana ja on yhä olemassa, vaikka sen pohjalta on johdettu uutta tietoa. Lähtökohtana tämä asetelma on jo otollinen historian tallennukselle, jokainen versio tiedosta on olemassa ja nämä versiot voidaan tallentaa taulukkoon, josta niihin päästään tarvittaessa käsiksi.

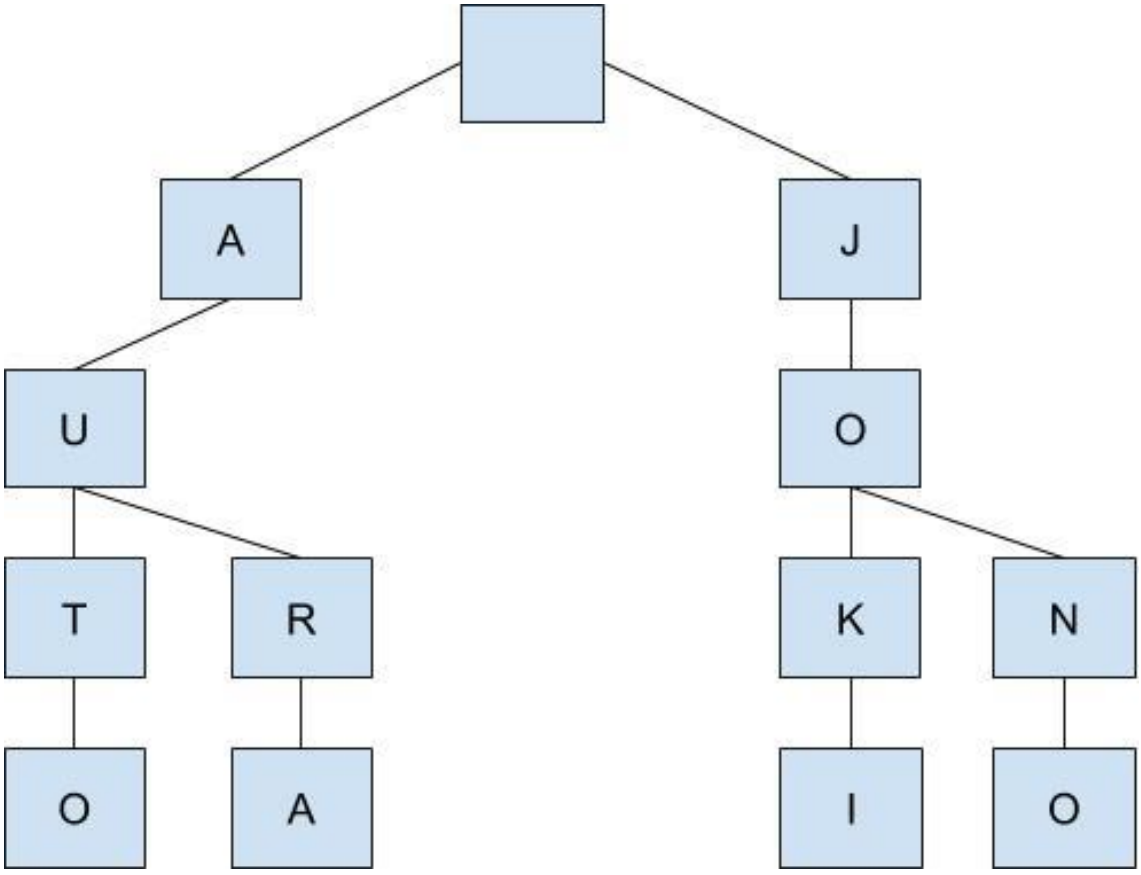
3.2 Rakenteellinen jakaminen (Structural sharing)

Tietorakenteet voivat olla hyvinkin suuria. Niiden kopioiminen alusta loppuun uudelleen ja uudelleen muuttumattomuuden nimissä voi käydä raskaaksi. Siihen löytyy ratkaisuksi rakenteellinen jakaminen.

Muuttumattomuus voidaan toteuttaa niin, että alkuperäinen muuttumaton tietorakenne, sekä uusi kopio siitä, jakavat yhteisiä rakenteita, jotka eivät ole muuttuneet. Tämä voi

olla suorituksen kannalta oleellinen optimointi, mikäli kopioitava tietorakenne on suuri. Uudessa kopiossa viitataan alkuperäisen tietorakenteen osiin. Vain muuttuneet osat korvataan uusilla tietorakenteilla. Alkuperäinen tietorakenne pysyy muuttumattomana.

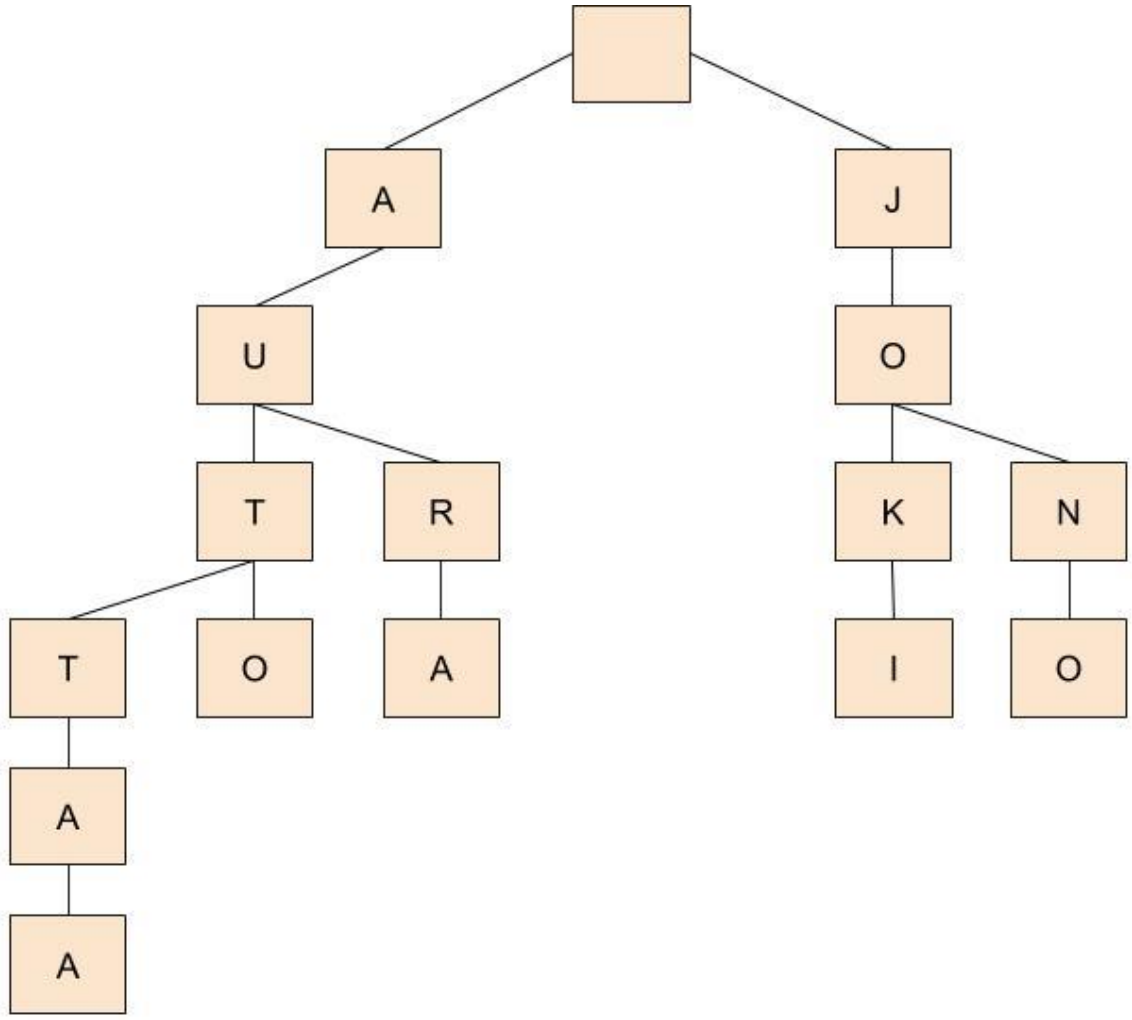
Seuraavassa esimerkissä havainnollistetaan rakenteellista jakamista trie-rakenteella. Trie on järjestetty etsintäpuu ja sen avaimet ovat usein merkkijonoja. Esimerkissä käytetään trie-rakennetta hakutoiminnon hakusanan täydennystoiminnolle. Jos käyttäjä kirjoittaa esimerkiksi hakukenttään ”au”, täydennys tarjoaisi au-alkuisia sanoja kuten ”auto” tai ”aura”. Jokainen merkkijono, joka lisätään hakusanantäydennystoimintoon, lisätään puuhun. Merkkijono asetetaan puuhun kirjain kerrallaan. Esimerkin puu on yksinkertaistettu esimerkki, joka pitää sisällään sanat auto, aura, joki ja jono.



Kuva 3. Yksinkertainen neljän sanan trie-puu

Kuvassa 3 havainnollistetaan täydennyspuun rakennetta. Jokainen sanojen kirjain muodostaa oman solmunsa, joka johtaa seuraaviin sanojen kirjaimiin.

Mitä tapahtuu, kun puuhun halutaan lisätä sana ”auttaa”? Olio-ohjelmoinnissa voitaisiin muuttaa alkuperäistä puuta ja lisätä puuttuvat kirjaimet puun jatkoksi. Funktionaalisessa ohjelmoinnissa on noudatettava muuttumattomuutta ja alkuperäinen tietorakenne on pysyttävä entisellään. Yleensä tässä tilanteessa tietorakenteen arvioista otetaan kopiot uuteen tietorakenteeseen, johon muutokset tehdään. Tällöin alkuperäinen tietorakenne pysyy muuttumattomana ja saadaan sitä vastaava uusi rakenne, johon muutokset on toteutettu.

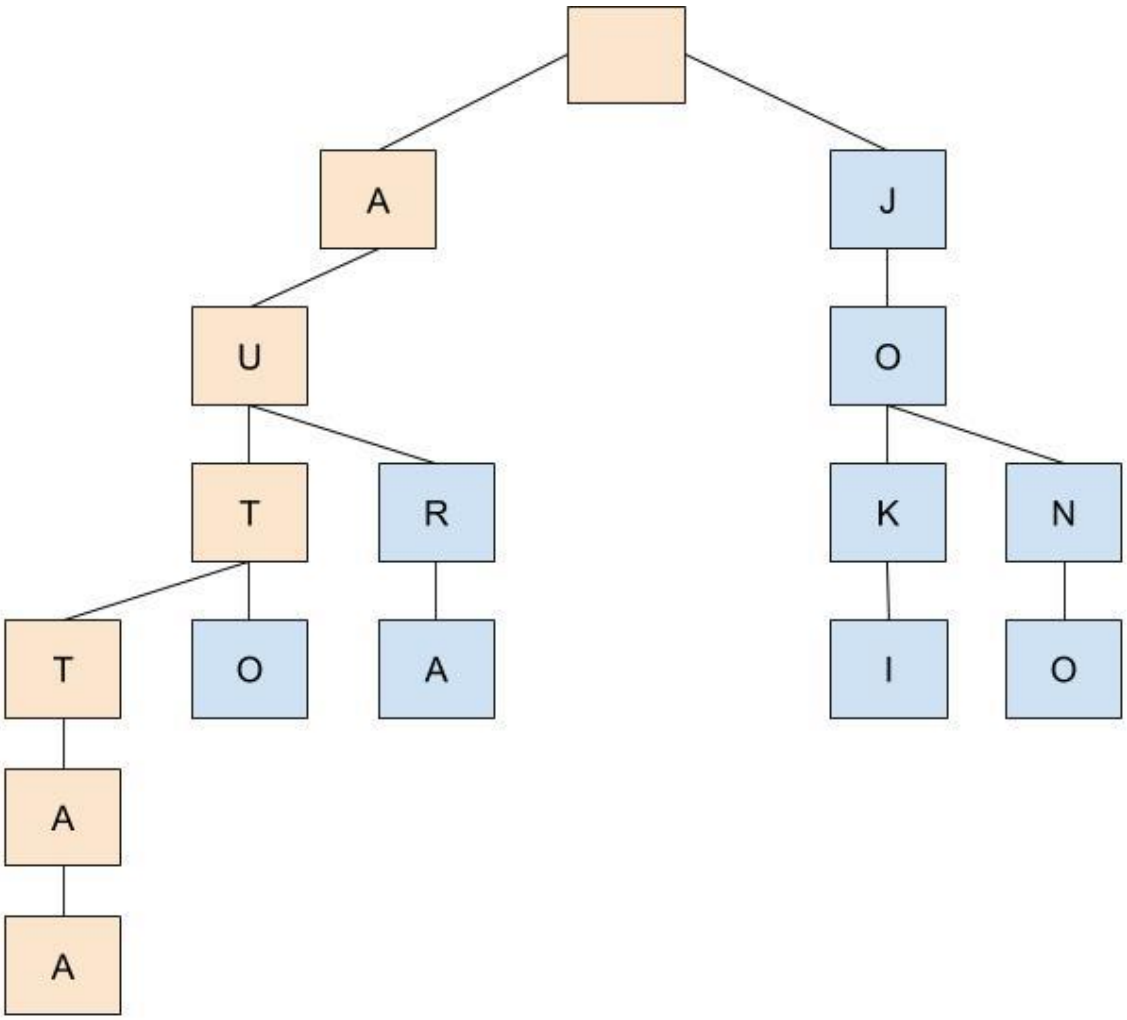


Kuva 4. Uusi kopio täydennyspuusta, johon on lisätty sana ”auttaa”.

Kuvan 4 oranssi väri esittää uuttaa puurakennetta. Koko puurakenne on uusi, siihen on lisätty sana ”auttaa”, ja alkuperäinen puu on pysynyt koskemattomana. Puurakenteet

saattavat kuitenkin usein olla suuria ja niiden täysi kopioiminen jokaisessa muutoksessa voi käydä kalliiksi suoritusajalle.

Rakenteellisessa jakamisessa voidaan hyödyntää niitä alkuperäisen puun rakenteita, joihin ei tule muutoksia.



Kuva 5. Rakenteellinen jakaminen puurakenteessa

Kuvan 5 puussa siniset solmut ovat viitteitä alkuperäiseen puuhun ja oranssit edustavat muuttuneita solmuja. Alkuperäinen puu pysyy yhä muuttumattomana, vaikka nyt sen solmuihin viittaa myös toinen puu. Voidaan huomata, että iso osa alkuperäisistä solmuista pystyttiin jakamaan uudelle puulle. Tämä mahdollistaa suorituskykyisen funktionaalisen muuttumattomuuden suurillekin tietorakenteille.

4 Funktiot

Ohjelmoinnissa funktioilla tarkoitetaan uudelleenkäytettävää ohjelman osaa, joka suorittaa jonkin toiminnon. Matematiikassa funktiolla tarkoitetaan hieman eri asiaa kuin ohjelmoinnissa. Matematiikan funktio saa syötteen, jonka perusteella saadaan tuloste. Tuloste on riippuvainen syötteestä ja yhdelle syötteelle on vain olemassa yksi tuloste. Funktionaalinen ohjelmointi perustuu pitkälti matematiikkaan ja siten funktionaalisessa ohjelmoinnissa funktioiden käytölle on asetettu tiettyjä sääntöjä. Funktioiden tulee olla puhtaita ja niiden on oltava ensimmäisen luokan kansalaisia. Puhtaat funktiot peilautuvat hyvin matemaattiseen funktioon.

4.1 Funktiot JavaScriptissä

JavaScriptissä voidaan luoda funktioita muutamalla eri syntaksilla, ja ne voivat olla nimettyjä ja anonyymejä. JavaScriptissä määritellään funktio käyttämällä function-avainsanaa. Seuraavassa esimerkissä määritellään nimetty sum-funktio.

```
function sum(x, y) {
  return x + y
}

sum(2, 3)
```

Esimerkkikoodi 8. Nimetyn funktion määrittely JavaScriptissä

Esimerkin 8 sum-funktio on nimetty funktio, joka saa kaksi parametria, x:n ja y:n, ja se summaa ne yhteen palauttaen tuloksen paluuarvona.

4.1.1 Anonyymit funktiot

Anonyymeillä funktioilla ei ole nimeä. Funktiolle ei kannata välttämättä aina antaa nimeä, mikäli sitä tullaan käyttämään ohjelmassa vain kerran. Anonyymiä funktiota ei alusteta tyylillä function functionName() {}, vaan se jätetään nimeämättä. Anonyymi funktio voidaan kuitenkin nimittää muuttujalle tyylillä let variableName = function() {}, jolloin funktio on kutsuttavissa muuttujan nimellä. Tätä kutsutaan funktiolausekkeeksi (function

expression). [Mantyla 2015: Chapter 2 Anonymous functions & Chapter 6 Function expressions.]

```
function notAnonymous() {
  console.log("this is a normal function")
}

let assignedAnonymous = function() {
  console.log("this is an anonymous function assigned to a variable")
}

notAnonymous()
assignedAnonymous()
```

Esimerkkikoodi 9. Anonyymi funktio

Esimerkissä 9 on kaksi funktiota. Ensimmäinen notAnonymous-funktio on nimetty funktio. Muuttujaan assignedAnonymous nimitetään anonyymi funktio, jota voidaan kutsua käyttämällä muuttujan nimeä.

4.1.2 Nuolifunktio

Nuolifunktiot ovat anonyymeja funktiota, joiden syntaksi poikkeaa normaalista funktion syntaksista. Nuolifunktion syntaksi on kompaktimpaa kuin normaalin funktion syntaksi.

```
const sum = (x, y) => x + y

const double = value => value * 2

sum(2, 3) // 5
double(2) // 4
```

Esimerkkikoodi 10. Nuolifunktion toteutus

Esimerkissä 10 luodaan kaksi nuolifunktiota. Ensimmäinen sum-funktio saa kaksi parametria ja summaa niiden arvot yhteen palauttaen tuloksen. Toinen funktio saa vain yhden parametrin ja kertoo sen kahdella ja palauttaa tuloksen. Nuolifunktiosta voi jättää parametrien sulut pois, mikäli parametreja annetaan vain yksi. Funktion body ei myöskään tarvitse kaarisulkeita eikä return-avainsanaa, mikäli nuolifunktion koko body halutaan palauttaa.

4.1.3 Hoistaus

Hoistaaminen tarkoittaa sitä, että ohjelman määritykset siirretään näkyvyysalueen alkuun ennen ohjelman ajoa. Funktiomäärittelyt hoistataan JavaScriptissä. Tämän vuoksi funktion kutsu voidaan tehdä ohjelmassa itse funktion määrittelyn yläpuolella. Ohjelmaa ajettaessa funktio on hoistattu, joten sen määrittely on siirretty ylös, ja se tapahtuu ennen funktion kutsua. JavaScript ei nosta muuttujiin määriteltyjä funktiolausekkeita näkyvyysalueen alkuun, vaan pysyvät ohjelmassa siinä kohtaa, missä ne on määritelty. Käyttämällä funktiolauseketta, eli antamalla funktio muuttujalle, voidaan funktion hoistaus estää. Näin funktion näkyvyys ei ala näkyvyysalueen alusta vaan vasta sen määrittelyn jälkeen. [Mantyla 2015: Chapter 2 Anonymous functions & Chapter 6 Function expressions.]

```
functionDeclaration()

function functionDeclaration() {
  console.log("this function gets hoisted")
}

let functionExpression = function() {
  console.log("this function does not get hoisted")
}

functionExpression()
```

Esimerkkikoodi 11. Funktioiden hoistaus

Esimerkin 11 ensimmäisen funktion functionDeclaration määrittely siirtyy näkyvyysalueen alkuun ja sitä voi näennäisesti kutsua ennen sen määrittelyä. Sen sijaan funktion functionExpression määrittely ei siirry ylös ja ohjelma tuottaisi virhetilanteen, jos sitä yritettäisi kutsua ennen funktion määrittelyä. Määrittelyn jälkeen funktio on kuitenkin käytettävissä normaalisti.

4.2 Puhtaat funktiot ja sivuvaikutukset

Puhtaan funktion paluuarvo riippuu myös täysin sen saamista parametreista ja samalla parametrilla saadaan johdonmukaisesti aina sama tulos. Tätä kutsutaan determinismiksi. Puhtaalla funktiolla ei ole myöskään sivuvaikutuksia funktion ulkopuolelle. Tässä suhteessa puhdas funktio muistuttaa matematiikan funktiota.

Parametrit toimivat syötteenä ja paluuarvo tulosteena. Tämä takaa sen, että samalla parametrilla saadaan aina sama arvo käyttäessä samaa funktiota. [Aravinth yms. 2018: Chapter 1. Functional Programming Benefits; Kereki 2017: Behaving properly – Pure Functions; Mantyla 2015: Chapter 2, Pure Functions.]

Puhtaat funktiot eivät käytä sen ulkopuolella olevia muuttujia, ohjelman tilaa, i/o-tuloksia tai sattumanvaraisia arvoja, sillä niiden käyttäminen on ristiriidassa puhtaan funktion määrityksen kanssa. Jos jokin funktion ulkopuolinen arvo, jota ei ole saatu parametrina, voi muuttaa funktion palauttamaa arvoa, ei se ole enää puhdas funktio. Kaikki data, mitä se käyttää, tulee tulla parametreina funktiolle, jolloin funktio palauttaa aina saman tuloksen samoille argumenteille. [Aravinth yms. 2018: Chapter 1. Functional Programming Benefits; Kereki 2017: Behaving properly – Pure Functions; Mantyla 2015: Chapter 2, Pure Functions.]

Samaan tapaan puhdas funktio ei myöskään muuta mitään sen ulkopuolella, kuten esimerkiksi objektia, muuttujaa, ohjelman tilaa tai i/o-laitteita, joten se ei tuota sivuvaikutuksia funktion ulkopuolelle. Tyypillisiä tilanteita, joissa funktiolla on sivuvaikutuksia ovat ohjelman globaalin tilan manipulointi funktion sisältä käsin, argumenttien manipulointi, i/o-toiminnot tai epäpuhtaiden funktioiden kutsuminen. [Aravinth yms. 2018: Chapter 1. Functional Programming Benefits; Kereki 2017: Behaving properly – Pure Functions; Mantyla 2015: Chapter 2, Pure Functions.]

Puhtaat funktiot tekevät testaamisesta ja virheiden paikantamisesta erittäin helppoa, sillä funktiolle täytyy vain antaa parametrit. Tällöin tiedetään, mikä tulos sen pitäisi antaa. Puhtaiden funktioiden käyttäminen on turvallista ja helppoa, koska funktion toteutusta ei tarvitse erikseen tutkia, tietääkseen, että sen käyttö ei aiheuta epätoivottuja sivuvaikutuksia. Puhtaita funktioita voidaan myös kutsua missä järjestyksessä hyvänsä, ja ne toimivat aina oletetusti, koska ne eivät riipu ohjelman tilasta vaan pelkästään parametreina saaduista arvoista. Puhtaiden funktioiden toiminnan ymmärtäminen koodia lukiessa on myös helppoa, koska ainoa, mikä on oleellista funktion kannalta, löytyy itse funktion koodista. Sen sijaan epäpuhtaiden funktioiden tapauksessa ohjelmaa joutuisi lukemaan monesta paikkaa ymmärtääkseen koko funktion toiminnan ja sen sivuvaikutukset. Sivuvaikutuksettomia funktioita pidetään myös yleisesti suositeltavana ohjelmoinnissa, se ei siis ole suotavaa pelkästään funktionaalisessa ohjelmoinnissa.

[Kereki 2017: Behaving properly – Pure Functions; Martin 2009: Chapter 3: Functions. Have No Side Effects.]

Täysin sivuvaikutukseton ohjelma olisi melko hyödytön. Käytännössä katsoen, ohjelma sisältää usein epäpuhtaita funktioita, esimerkiksi datan hakemista API-kutsuin tai datan näyttöä käyttäjälle. Silti on mahdollista välttää epäpuhtaita funktioita monissa tilanteissa, joista seuraavaksi esitellään muutama esimerkki.

Jos ohjelmassa on funktionaalisen ohjelmoinnin rinnalla jokin globaali tila, jota funktio tarvitsee, tulee se antaa funktiolle parametrina. Sen sijaan, että funktio manipuloisi tilaa funktiosta käsin, se palauttaa kutsujalle uuden version ohjelman tilasta ja antaa kutsujan päivittää tila uuteen tilaan, mikäli on tarvetta. Näin voidaan pitää funktio edelleen puhtaana, kun ohjelman tilaa manipuloivat osat pidetään erillään. [Kereki 2017: Behaving properly – Pure Functions.]

```
const someObject = {
  value: 124,
}

const updateObject = (someObject, newValue) => {
  return { ...someObject, value: newValue }
}

const updateObjectImpure = (newValue) => {
  someObject.value = newValue
}
```

Esimerkkikoodi 12. Esimerkki puhtaasta ja epäpuhtaasta funktiosta.

Esimerkissä 12 on kaksi funktiota, updateObject ja updateObjectImpure, joista ensimmäinen on puhdas funktio. Se saa kaiken tarvitsemansa datan parametreina, se ei manipuloi saamaansa dataa suoraan eikä aiheuta sivuvaikutuksia. Se luo uuden olion someObject-olion pohjalta ja palauttaa tämän.

Toinen funktio, updateObjectImpure, on sen sijaan riippuvainen sen ulkopuolisesta someObject-oliosta, jota se ei saa parametrina. Sen lisäksi se aiheuttaa sivuvaikutuksia manipuloimalla someObject-olion values-arvoa. Tämä aiheuttaa helposti ongelmia. Suoritusjärjestyksen muuttuminen muuttaisi funktioiden tuloksia. Jos ensin kutsuttaisiin epäpuhdasta funktiota, ja sitten vasta puhdasta, vaikuttaisi epäpuhtaan funktion

sivuvaikutukset myös seuraavan funktion parametriksi saamaan olioon ja siten sen tuloksiin.

Epäpuhdasta funktiota on myös vaikea uudelleenkäyttää kahdesta syystä. Se on riippuvainen siitä, että sen näkyvyysalueella sijaitsee someObject-olio. Sen lisäksi sitä ei voida käyttää vain uuden olion tilan laskemiseen, vaan se muuttaa oliota samalla. Se tekee siis kaksi asiaa yhden sijaan, mikä on jo lähtökohtaisesti huono asetelma funktiolle niin funktionaalisessa ohjelmoinnissa kuin muutoinkin ohjelmoidessa. [Martin 2009: Chapter 3: Functions. So One Thing.]

Funktion testaaminen on myös hankalaa samasta syystä, että se tarvitsee tietyn ympäristön toimiakseen. Toisekseen, funktio ei palauta mitään, joten paluuarvoa ei voida tarkistaa testaamisen yhteydessä. Paluuarvon puuttuminen paljastaa jo heti, että kyseessä on joko turha funktio, joka ei tee mitään, tai luultavammin kyseessä on epäpuhdas sivuvaikutteinen funktio.

Tässä pienessä esimerkkitilanteessa tilanne on vielä hallittavissa, vaikka funktio ei ole puhdas, mutta mitä suuremmaksi ohjelma kasvaa, sitä monimutkaisemmaksi riippuvuussuhteet ja sivuvaikutukset muuttuvat. Lopulta ohjelmoijan on mahdoton tietää varmasti, mitä tulee tapahtumaan funktion käytön seurauksena, mikä altistaa ohjelman ennakoimattomille virheille.

```
// impure
const fetchData = () => {
  // Do some fetching here
  const data = { firstname: "John", lastname: "Doe", birthyear: "1978" }
  return data
}

// impure
const getCurrentYear = () => {
  let currentTime = new Date()
  return currentTime.getFullYear()
}

// pure
const getFullName = person => {
  return `${person.firstname} ${person.lastname}`
}

// pure
const getAge = (person, currentYear) => {
  return currentYear - person.birthyear
```

```
}

// impure
const printData = data => {
  console.log(data)
}

let person = fetchData()
let currentYear = getCurrentYear()
let fullName = getFullName(person)
let age = getAge(person, currentYear)

printData(`${fullName} is ${age} years old.`) // John Doe is 42 years old.
```

Esimerkkikoodi 13. Koodin pilkkominen moneen funktioon, jolloin mahdollisimman moni funktio olisi puhdas

Esimerkissä 13 on pilkottu jokainen ohjelman osa omaan funktioonsa, jotta epäpuhtaat ohjelman osat pysyisivät eristyksissä puhtaista ohjelman osista. Tiedon haku, jota demostroi funktio fetchData, olisi epäpuhdas, mikäli data haetaan esimerkiksi jostain rajapinnasta, jolloin funktion palauttama arvo ei ole sidoksissa saatuihin parametreihin. GetCurrentYear-funktio on samasta syystä epäpuhdas. GetFullName- ja getAge-funktiot ovat puhtaita. Niiden paluuarvo riippuu täysin parametrina saadusta person-oliosta. PrintData-funktio on epäpuhdas, koska sillä on sivuvaikutuksia, kun se tulostaa merkkijonon console.log-funktiolla. Tämä sama logiikka olisi helposti voitu kirjoittaa niin, että koko ohjelma olisi epäpuhdas. Mutta nyt pieniksi pilkottuina funktioina kaksi funktiota ovat puhtaita, ne ovat uudelleenkäytettäviä sekä helposti testattavissa.

4.3 Ensimmäisen luokan kansalainen (First Class Citizen)

Funktionaalisessa ohjelmoinnissa funktiot ovat ensimmäisen luokan kansalaisia, mikä tarkoittaa, että niitä voidaan kohdella primitiivien, eli esimerkiksi numeroiden ja merkkijonojen, tavoin. Niitä voidaan tallentaa muuttujiin, antaa parametreina, tai palauttaa paluuarvoina aivan kuten muitakin primitiiviarvoja. JavaScript kohtelee funktioita ensimmäisen luokan kansalaisina, joten se täyttää tältä osin hyvin funktionaalisen ohjelmoinnin vaatimukset. [Antani ym. 2016: Chapter 2. Organizing Code. Objects in JavaScript.]

```
let func1 = () => {console.log("I am a function")}
func1() // I am a function
```

Esimerkkikoodi 14. Esimerkki funktion tallentamisesta muuttujaan

Esimerkissä 14 funktio, joka tulostaa ”I am a function”, tallennetaan muuttujaan func1. Funktiota voidaan kutsua nyt käyttämällä muuttujaa.

```
let func1 = () => {console.log("I am a function")}
let func2 = (somefunction) => {somefunction()}
func2(func1) // I am a function
```

Esimerkkikoodi 15. Esimerkki funktion antamisesta parametriksi

Esimerkissä 15 käytetään edellisen esimerkin alkutilannetta, jossa tulostava funktio tallennetaan muuttujaan func1. Seuraavaksi tehdään toinen funktio, func2, joka vastaanottaa funktion parametrina ja kutsuu sitä. Siten kutsumalla func2:sta ja antamalla sille parametriksi func1-funktion, saamme tulostettua taas ”I am a function”. [Mantyla 2015: Chapter 2, Higher-order functions.]

```
let func3 = () => {return () => {console.log("I am a function")}}
let func4 = func3()
func4() // I am a function
```

Esimerkkikoodi 16. Esimerkki funktion palauttamisesta paluuarvona

Esimerkissä 16 funktio func3 palauttaa toisen funktion, joka tulostaa ”I am a function”. Tallentamalla func3:sen paluuarvon func4:seen ja kutsumalla func4:sta saamme taas tulostettua I am a function -tekstin. [Aravinth yms. 2018: Chapter 3. Understanding data.]

4.4 Korkeamman kertaluvun funktio (Higher Order Function)

Korkeamman kertaluvun funktio tarkoittaa funktiota, joka käyttää jollain tapaa muita funktioita. Tällaiselle funktiolla voidaan antaa parametreina funktio ja/tai se palauttaa funktion. Korkeamman kertaluvun funktio mahdollistaa koodin yleistämisen siten, että jokaiselle yksittäistapaukselle ei tarvitse tehdä omaa räätälöityä funktiota, vaan funktiot voivat hyödyntää samoja elementtejä. Tällä tekniikalla funktioista tulee usein pieniä, ja ne hoitavat vain yhden asian. Niitä yhdistelemällä voidaan luoda vaativampia

kokonaisuuksia. Korkeamman kertaluvun funktio voi lisätä olemassa olevalle funktiolle uusia toiminnallisuuksia muuttamatta alkuperäisen funktion toimintaa tai ne voivat myös muuttaa alkuperäisen funktion toimintaa jollakin tapaa, esimerkiksi käsittelemällä paluuarvoa. [Aravinth yms. 2018: Chapter 3. Higher order functions; Kereki 2017: Producing Functions – Higher-Order Functions.]

```
let sum = (a, b) => a + b

let addVat = (fn, vat) => (...args) => {
  return fn(...args) * (vat / 100 + 1)
}
let sumWithVat = addVat(sum, 24)
sumWithVat(2, 10) // 14.879999999999999
```

Esimerkkikoodi 17. Tuloksen muuntaminen käyttäen korkeamman kertaluvun funktiota

Esimerkkikoodissa 17 funktion toimintaa muutetaan käyttämällä korkeamman kertaluvun funktiota. Ohjelmassa on funktio sum, joka laskee kaksi parametriksi saatua arvoa yhteen. Ohjelmassa on myös addVat-funktio, joka saa parametrina jonkin funktion sekä arvonlisäveroprosentin. Se palauttaa uuden funktion, jota kutsutaan samalla lailla kuin sen parametriksi saatua funktiota. Esimerkissä luodaan uusi funktio, joka käyttää parametrina saatua sum-funktiota. Sen sijaan, että palautettaisiin sum-funktion tulos, käsitellään sen palauttamaa tulosta vielä lisää ennen palauttamista lisäämällä siihen 24 prosentin arvonlisävero. Näin voidaan luoda uusia funktiota hyödyntäen jo olemassa olevia funktiota.

Ilman funktionaalista tapaa, jossa funktioita voidaan antaa parametreina ja palauttaa paluuarvoina, vastaava lopputulos saataisiin joko tekemällä yksi funktio, joka tekee molemmat asiat, jolloin funktiolla olisi hyvin rajattu käyttötarkoitus tai käyttämään kahta funktiota vuorotellen, jolloin ohjelmassa jouduttaisiin usein käyttämään kahta funktiota peräkkäin ja antamaan samat parametrit niille, tai tekemään kolmas funktio, joka käyttää kumpaakin funktiota vuorotellen.

```
let sumWithVat = (a, b, vat) => {
  return (a + b) * (vat / 100 + 1)
}

sumWithVat(2, 10, 24) // 14.879999999999999
```

Esimerkkikoodi 18. sumWithVat-funktio, joka on sovellettu hyvin yksityiskohtaiseen käyttötarkoitukseen, eikä sen logiikkaa voida hyödyntää muualla.

Yllä olevan esimerkin 18 funktio ei ole joustava. Se on räätälöity yhteen tilanteeseen ja sen lisäksi se tekee kaksi asiaa. Yleinen nyrkkisääntö on, että funktion tulisi aina tehdä vain yksi asia.

```
let sum = (a, b) => a + b
let addVat = (value, vat) => value * (vat / 100 + 1)

addVat(sum(2, 10), 24) // 14.879999999999999
```

Esimerkkikoodi 19. Kahden pienemmän funktion käyttö peräkkäin.

Esimerkissä 19 funktio on jaettu kahdeksi eri funktioksi, jotka kummatkin tekevät yhden asian. Jos tätä yhdistelmää tulee kuitenkin kutsuttua usein, ensin siis sum-funktiota, ja sen tulokselle addVat-funktiota, joudutaan ohjelmassa kirjoittamaan usein tämä hankalan oloinen kahden funktion yhdistelmä.

```
let sum = (a, b) => a + b

let addVat = (value, vat) => value * (vat / 100 + 1)

let sumWithVat = (a, b, vat) => {
  return addVat(sum(a, b), vat)
}

sumWithVat(2, 10, 24) // 14.879999999999999
```

Esimerkkikoodi 20. Kolmas esimerkki ilman korkeamman kertaluvun funktiota

Esimerkissä 20 tehdään funktio, joka puolestaan tekee hankalan kahden funktion yhdistelmäkutsun. Tämä ratkaisu on kuitenkin riippuvainen sen ulkona olevista funktioista, joita se ei saa parametreina, sekä se vastaanottaa itse kolme parametria, joka on jo suhteellisen suuri määrä. Toki tilanteen voisi ratkaista vielä monella eri tavalla, mutta korkeamman kertaluvun funktio on hyvä tällaisissa tilanteissa. Se pitää funktiot pieninä, mahdollistaa funktioiden joustavan yhdistelyn myös muiden tilanteiden ratkaisemiseksi, pitää funktiot puhtaina ja pitää funktioiden saamien parametrien määrän kohtuullisena.

JavaScriptissä on monia korkeamman kertaluvun funktioita sisäänrakennettuna, kuten esimerkiksi map, sort tai forEach. Näille kaikille annetaan funktio parametrina, ja ne hyödyntävät saatua funktiota eri tavoin.

Myös suositussa JavaScript-kirjastossa Reactissa käytetään korkeamman kertaluvun periaatetta React-komponenteille. Reactissa siihen viitataan nimellä Higher-Order Component (HOC). HOC on puhdas funktio, jolla ei ole sivuvaikutuksia, ja se lisää ominaisuuksia olemassa olevaan komponenttiin. Se saa argumenttina komponentin ja palauttaa uuden komponentin. HOCissa ei muuteta argumenttina saatua komponenttia, vaan se kääritään toiseen komponenttiin. Samalla lisätään alkuperäiseen komponenttiin uusia ominaisuuksia tai toiminnallisuuksia. Monia HOC:eja voidaan yhdistellä ilman, että ne ylikirjoittavat toisiaan tai ovat ristiriidassa toistensa kanssa. [Reactjs. Docs. Higher order components.]

4.5 Takaisinkutsufunktio (Callback)

JavaScriptissä paljon käytetyt takaisinkutsufunktiot liittyvät korkeamman kertaluvun funktioihin, sillä funktio, jolle annetaan parametrina takaisinkutsufunktio, on korkeamman kertaluvun funktio. Takaisinkutsufunktiolla tarkoitetaan funktiota, joka annetaan parametrina toiselle funktiolle, jonka odotetaan kutsuvan takaisinkutsufunktiota. Takaisinkutsufunktioita käytetään usein asynkronisen operaation lopussa, kun halutaan kutsua jotain tiettyä funktiota operaation päätyttyä. [Mantyla 2015: s. 28; Mozilla: Callback function.]

```
const getUserData = callback => {
  console.log("getUserData called")
  setTimeout(() => {
    callback({ id: "1234", name: "Mikko" })
  }, 3000)
}

const handleData = data => {
  console.log("handleData called")
}

const someOtherOperation = () => {
  console.log("someOtherOperation called")
}

getUserData(handleData)
```

```
someOtherOperation()
```

Esimerkkikoodi 21. getUserData-funktio saa takaisinkutsufunktion parametrina, jota se kutsuu lopuksi

Esimerkki 21 mallintaa tilannetta, jossa funktio hakee dataa, jonka jälkeen data halutaan siirtää eteenpäin toiselle funktiolle. Funktiossa getUserData saa parametrina takaisinkutsufunktion. Funktiossa käytetään kolmen sekunnin viivettä, enne kuin takaisinkutsufunktiota kutsutaan. Tämä demonstroi asynkronista API-kutsua, jossa voi olla viivettä ennen kuin haettu data saadaan. Tämän koko demostroinnin tarkoitus on siinä, että ohjelma ei lakkaa etenemästä viiveen aikana, vaan se jatkaa ohjelman kulkua eteenpäin. Kuitenkin kolmen sekunnin jälkeen, kun haluttu data on saatavilla, voidaan kutsua handleData-takaisinkutsufunktiota, välittää tälle parametrina data, jolloin suoritus siirtyy tähän funktioon. Ohjelmassa on mukana lokeja, jotka todentavat ohjelman suoritusjärjestyksen. Ensin lokiin tulee ”getUserData called”, sitten ”someOtherOperation called” ja vasta kolmen sekunnin kuluttua ”handleData called”. Ohjelman suoritus jatkuu viiveen aikana, ja se suorittaa someOtherOperation-funktiokutsun ennen takaisinkutsufunktiota.

4.6 IIFE

IIFE eli Immediately-invoked function expression, on funktio, joka kutsuu heti määrittelynsä jälkeen itse itseään. Funktiot kapseloivat niiden sisällä olevat muuttuja- ja funktiomäärittelyt funktion sisällä olevalle näkyvyysalueelle. IIFE:n sisällä voidaan toteuttaa ohjelmaa, joka halutaan sulkea muun koodin näkyvyysalueelta tai vain pitää globaali nimiavaruus siistimpänä. Joitain ohjelman alkutoimia voidaan esimerkiksi ajaa IIFE:n sisällä, jos siinä käytettäviä funktioita ja muuttujia ei tarvita enää myöhemmin ohjelmassa. [Kereki 2017: Starting out with functions – A core concept, Using functions in FP way, Immediate invocation.]

```
(() => {
  console.log("IIFE running")
  const value = 1234
  const fn = () => console.log("does something inside iife")
})()
```

Esimerkkikoodi 22. IIFE-funktio täytyy kääriä sulkeisiin ja lopuksi vielä kutsua sitä uusilla sulkeilla.

Esimerkin 22 anonyymi nuolifunktio on käärittyy sulkeisiin, ja funktion lopussa on vielä uudet sulkeet, jotka kutsuvat tätä funktiota saman tien sen määrittelyn jälkeen. Kaikki IIFE:n sisällä olevat muuttuja- ja funktiomääritykset jäävät sen sisälle, eikä niihin pääse enää käsiksi myöhemmin.

4.7 Sulkeuma (Closure)

Sulkeumalla voidaan sulkea muuttujia ja funktioita pienempään näkyvyysalueeseen. Motiivi tällaiselle sulkemiselle on se, että globaaleita muuttujia ja funktioita on helppo sorkkia vahingossa, ja jos näitä muuttujia tai funktioita ei tarvita globaalissa näkyvyysalueessa, ne täyttävät nimiavaruutta turhaan. Funktiot sulkevat niiden sisällä määritellyt muuttujat ja funktiot funktion sisälle. Ne eivät näy muulle ohjelmalle funktion ulkopuolelle. Funktio voi kuitenkin palauttaa sen ulkopuolella tarvittavia muuttujia tai funktioita, jotta niitä voidaan käyttää sen ulkopuolella. [Mantyla 2015: Chapter 2, Self-invoking functions and closures & Chapter 6, Closures.]

Seuraavissa esimerkeissä toteutetaan ensin funktio plantTrees ilman sulkeumaa, ja sen jälkeen sama toiminto toteutetaan sulkeumaa hyödyntäen.

```
let trees = 50

let plantTrees = (n) => {
  if (n > 0) {
    trees += n
  }
}

let getTrees = () => trees

console.log(getTrees()) //50
plantTrees(5)
console.log(getTrees()) //55
console.log(trees) //55
```

Esimerkkikoodi 23. Funktio ilman sulkeumaa.

Esimerkkikoodi 23:ssä muuttuja trees sekä metodit plantTrees ja getTrees ovat globaalissa näkyvyysalueessa. Ohjelma pääsee käsiksi muuttujaan ja metodeihin suoraan. Seuraavassa esimerkissä tämä koodi suljetaan sulkeuman sisään.

```
let forest = (() => {
  let trees = 50
  return {
    plantTrees: (n) => {
      if (n > 0) {
        trees += n
      }
    },
    getTrees: () => trees
  }
})()

console.log(forest.getTrees()) //50
forest.plantTrees(5)
console.log(forest.getTrees()) //55
console.log(forest.trees) //undefined
```

Esimerkkikoodi 24. Sulkeuma

Tässä esimerkissä 24 muuttujat sekä metodit on suljettu funktion sisään, jolle on annettu nimi forest. Funktio ajaa heti itsensä määrittelynsä yhteydessä, eli se on IIFE-funktio. Funktio palauttaa olion, jolla on kaksi metodia, plantTrees ja getTrees. Nämä metodit pääsevät edelleen käsiksi sulkeuman sisällä määriteltyyn trees-muuttujaan. Kutsumalla forest.getTrees()-funktiota huomataan, että trees-muuttujan arvo saadaan ulos sulkeumasta. Koodiesimerkin viimeisellä rivillä yritetään päästä suoraan käsiksi trees-muuttujaan, mutta tämän arvo on undefined. Tällä tavoin voidaan luoda hallittuja näkyvyysalueita erilaisille ohjelman osille.

4.8 Funktion dynaaminen määrittäminen

Funktioita voidaan määrittää dynaamisesti JavaScriptissä samaan tapaan kuin muitakin muuttujien arvoja voidaan määrittää muuttujalle ohjelman ajon aikana. Tämä mahdollistaa funktion vaihtamisen tarpeen tullen ilman, että funktion kutsua tarvitsee muuttaa eri puolilla koodia.

```
const date = new Date() // 2020-03-16T16:27:39.276Z
```

```
// date.month.year
let getDateInString = date => {
  return `${date.getDate()}.${date.getMonth() + 1}.${date.getYear() + 1900}`
}

getDateInString(date) // 16.3.2020

// month.date.year
getDateInString = date => {
  return `${date.getMonth() + 1}.${date.getDate()}.${date.getYear() + 1900}`
}

getDateInString(date) // 3.16.2020
```

Esimerkkikoodi 25. Funktion getDateInString uudelleenmäärittely ajon aikana

Esimerkissä 25 on funktio getDateInString, jonka tarkoitus on palauttaa Date-olion päivämäärä merkkijonona. Alun perin funktio on määritelty näyttämään päivämäärä suomalaisille tutussa formaatissa, jossa ensin tulee päivämäärä, sitten kuukausi ja viimeisenä vuosi. Tällöin ensimmäinen funktiokutsu päivämäärällä 16.3. palauttaa ”16.3.2020”. Tämän jälkeen funktio määritellään uudelleen uudeksi funktioksi, joka esittää kuukauden ennen päivämäärää. Tämän jälkeen funktion kutsu palauttaakin ”3.16.2020”.

Funktioiden dynaamisen määrittelyn edut tulevat siitä, että funktion kutsua ei tarvitse muuttaa ohjelmassa, vaan samaa funktiokutsua voidaan edelleen kutsua ympäri ohjelmaa, mutta funktion toteutus vain muuttuu toiseksi funktioksi. Funktio voidaan vaihtaa kesken ohjelman ajon toiseen toteutukseen esimerkiksi käyttäjän valintojen tai toimien johdosta. Jos funktioita ei voitaisi määritellä dynaamisesti, jouduttaisiin oikean toiminnan takaamiseksi suorittaa erilaisia ehtolauseita ja valita toiminto sen mukaan.

4.8.1 Funktion määrittely ympäristön mukaan

Kehitysympäristössä halutaan usein toimia hieman eri lailla kuin itse tuotantoympäristössä. Funktionaalinen ohjelmointi mahdollistaa funktion toteutuksen vaihtamisen dynaamisesti. Kun funktion toteutus muutetaan ohjelman alussa ympäristön mukaan, ei ympäristöä tarvitse tarkistaa monessa paikassa ohjelmaa. Riittää, kun funktio määritellään dynaamisesti heti ohjelman alussa, kun ympäristö on selvillä. Ohjelma voi normaalisti jatkaa näennäisesti saman funktion kutsumista, eikä varsinaista ohjelmaa

tarvitse muuttaa. [Kereki 2017: Starting out with functions. Using functions in FP ways. Stubbing.]

```
let environment = "development"

let sum = (a, b) => {
  return a + b
}

let addLogging = fn => (...args) => {
  let result = fn(...args)
  console.log(fn.name, args, result)
  return result
}

if (environment === "development") {
  sum = addLogging(sum)
}

sum(14, 6) // sum [ 14, 6 ] 20
sum(1, 2) // sum [ 1, 2 ] 3
```

Esimerkkikoodi 26. Ohjelma tarkistaa, onko ympäristö developement, ja muuttaa funktion määrittelyä, mikäli se on.

Esimerkkikoodissa 26 vaihdetaan funktion toteutus, kun ollaan kehitysympäristössä. Funktion halutaan kirjaavan lokiin funktion nimi, sen saamat parametrit sekä sen palauttaman paluuarvon, kun ollaan kehitysympäristössä. Muussa tapauksessa funktion ei tule kirjata mitään. Tässä yksinkertaistetussa esimerkissä muuttuja environment määrittelee, onko kyseessä tuotanto- vai kehitysympäristö. Tuotantoympäristössä käytetään sum-funktiota yhteenlaskun suorittamiseksi. Kehitysympäristössä sum-funktion toteutusta laajennetaan dynaamisesti niin, että se tulostaa konsoliin funktion saamat parametrit sekä paluuarvon. Funktiota voidaan käyttää samalla lailla molemmissa tapauksissa, eikä ohjelmaa tarvitse muuttaa muualla kuin funktion määrityksessä, eikä ympäristöä tarkastavia if-lauseita tarvitse ripotella ympäri ohjelmaa muuttaakseen funktion toimintaa ympäristöstä riippuen.

4.8.2 Stubbing

Stubbing on testauksessa käytetty termi, jolla tarkoitetaan funktion korvaamista yksinkertaisemalla funktiolla testausta varten. Sitä käytetään, kun jonkin funktion toteutus ei ole valmis tai jos halutaan korvata esimerkiksi i/o-tapahtumia kovakoodatuilla arvoilla. Se mahdollistaa ohjelman osien testaamisen, vaikka koko ohjelma ei olisi vielä

valmis. Samaan tapaan kuin esimerkkikoodissa 26, testiympäristössä voitaisiin vaihtaa dynaamisesti haluttujen funktioiden toteutus yksinkertaisempiin funktioihin. [Antani ym. 2016: 12. Patterns for Testing. Stubs.]

4.8.3 Polyfill

Polyfillillä tarkoitetaan skriptiä, joka lisää ohjelmaan uuden funktion tai päivittää funktiota. Sitä käytetään tilanteissa, joissa käytetään jotain uutta JavaScriptin funktiota, jota kaikki JavaScript-moottorien kehittäjät eivät ole vielä toteuttaneet. Polyfill siis täydentää puuttuvan toteutuksen tälle uudelle JavaScriptin ominaisuudelle, jotta ohjelma toimisi myös selaimilla, jotka eivät tue kyseistä ominaisuutta vielä. [Antani ym. 2016: 6. ECMAScript 6. Shims or polyfills; JavaScript info. Polyfills; Kereki 2017: Starting Out with Functions – A Core Concept. Using functions in FP ways. Polyfills.]

Seuraavassa esimerkissä 27 tehdään polyfill JavaScriptin uudelle Object.fromEntries()-metodille. Metodille annetaan argumentiksi taulukollinen avain- ja arvopareja taulukkomuodossa, jotka metodi muuttaa olioksi. Polyfilliä toteuttaessa ensin on tarkistettava, mikäli kyseinen JavaScriptin ominaisuus on jo toteutettu. Tässä tapauksessa tarkastetaan, onko Object.fromEntries olemassa seuraavalla tarkistuksella Object.fromEntries === undefined. Mikäli tarkistus palauttaa truen, tulee polyfillin toteuttaa ominaisuus.

```
const person = [
  ["firstname", "John"],
  ["lastname", "Doe"],
]

if (Object.fromEntries === undefined) {
  Object.fromEntries = array => {
    return array.reduce((obj, pair) => ((obj[pair[0]] = pair[1]), obj), {})
  }
}

Object.fromEntries(person) // { firstname: 'John', lastname: 'Doe' }
```

Esimerkkikoodi 27. Polyfill Object.fromEntries-funktiolle

Esimerkkikoodissa 27 luodaan ensin person-taulukko, jossa on kaksi sisäkkäistä key-value-taulukkoa. Seuraavaksi tarkistetaan, onko Object.fromEntries toteutettu. Mikäli

selain ei tue kyseistä toimintoa vielä, tarkistus Object.fromEntries === undefined palauttaa true-totuusarvon ja funktion toteutus tehdään dynaamisesti määrittelemällä Object.fromEntries-funktio. Tämän jälkeen Object.fromEntries-funktiota on turvallista käyttää, vaikka selain ei vielä tukisikaan sitä natiivisti. Ilman polyfilliä ohjelma kaatuisi ensimmäiseen Object.fromEntries-kutsuun.

4.9 Viittausten läpikuultavuus (Referential Transparency)

Funktionaalisessa ohjelmoinnissa viittausten läpikuultavuudella tarkoitetaan sitä, että mikä tahansa funktion lausekkeista (expression) voidaan korvata lausekkeen arvolla vaikuttamatta ohjelman lopputulokseen. [Kereki 2017. Behaving Properly – Pure Functions. Pure functions. Referential Transparency; Simpson 2018: Chapter 5: Reducing Side Effects. There or not.]

```
const addVat = value => {
  return value * (1 + 0.24)
}

console.log(addVat(12)) // 14.879999999999999

const addVat2 = value => {
  return value * 1.24
}

console.log(addVat2(12)) // 14.879999999999999
```

Esimerkkikoodi 28. Lauseke vaihdetaan toiseen ilman sivuvaikutuksia

Esimerkissä 28 on funktio addVat, joka lisää argumentiksi saatuun arvoon 24 prosentin arvonlisäveron. Funktion lauseke (1 + 0.24) vaihdetaan lausekkeen arvoon 1.24. Vaikka lauseke on vaihdettu sen arvoon, pysyy ohjelman tulos täysin samana.

Tätä samaa periaatetta noudattaen kokonaisia funktiokutsuja voidaan korvata niiden palauttamalla tuloksella.

```
const getLetterCount = word => {
  return word.length
}

let string = `The word "Hello" is ${getLetterCount("Hello")} letters long.`
```

```
console.log(string) // The word "Hello" is 5 letters long.
string = `The word "Hello" is 5 letters long.`
console.log(string) // The word "Hello" is 5 letters long.
```

Esimerkkikoodi 29. Funktion viitteiden läpikuultavuus

Esimerkkikoodissa 29 käytetään funktiota getLetterCount parametrina saadun sanan kirjaimien laskemiseen. Ohjelmassa tulostetaan lause, jossa käytetään kyseistä funktiota laskemaan sanan ”Hello” kirjaimien määrä. Funktio palauttaa tulokseksi 5, joten string-muuttujan lauseeksi saadaan ” The word "Hello" is 5 letters long.”. Koska funktio getLetterCount noudattaa funktionaalisen ohjelmoinnin asettamia rajoja, se on puhdas ja sen tuloste riippuu täysin sen saamasta syötteestä, voidaan funktiokutsu getLetterCount(”Hello”) korvata suoraan sen tuloksella 5. Lopputulos tulee olemaan sama.

Monet ohjelmointikääntäjät tekevät constant folding -toiminnon käännön aikana. Tämä tarkoittaa sitä, että yksinkertaiset lausekkeet kuten numerolaskutoimitukset voidaan laskea jo käännösvaiheessa, eikä ohjelman ajon aikana. Tässä tapahtuu siis juuri se, mitä viittausten läpikuultavuudella tarkoitetaan. Esimerkiksi 1 + 0.24 voidaan korvata suoraan lausekkeen arvolla 1.24 vaikuttamatta ohjelman lopputulokseen. [Kereki 2017. Behaving Properly – Pure Functions. Pure functions. Referential Transparency.]

4.10 Muistintaminen (Memoization)

Muistintaminen on keino nopeuttaa raskaiden funktioiden suorittamista, erityisesti, jos niitä kutsutaan usein ohjelman aikana. Funktion tulos tallennetaan välimuistiin tietyillä parametreilla, ja kun funktiota kutsutaan uudelleen samoilla parametreilla, voidaan tulos lukea suoraan välimuistista suorittamatta raskasta ohjelmakoodia uudelleenlaskemiseksi. Muistintaminen vie muistia, mutta vastineeksi saadaan nopeampi suoritus. Ilman puhtaita funktioita muistintaminen ei toimisi, sillä tulos saattaisi vaihdella, vaikka funktion saamat argumentit olisivat tismalleen samat. [Kereki 2017: Behaving Properly – Pure functions. Advantages of pure functions. Memoization; Antani ym. 2016: Module 2, Chapter 6, Memoization.]

```
const calculation = value => {
  return value * value
}

const memo = func => {
  let cache = {}
  return value => {
    if (cache[value]) {
      console.log("Value from cache")
      return cache[value]
    } else {
      console.log("Calculating value")
      let result = func(value)
      cache[value] = result
      return result
    }
  }
}

const cachedCalculation = memo(calculation)

console.log(cachedCalculation(5)) // calculating
console.log(cachedCalculation(5)) // using cache
console.log(cachedCalculation(10)) // calculating
```

Esimerkkikoodi 30. Memoization-esimerkkikoodi

Esimerkkikoodi 30:ssa käytetään memo-funktiota välimuistina. Memo-funktio saa parametriksi funktion, jonka tulokset halutaan tallentaa välimuistiin. Se palauttaa uuden funktion, jota käytetään alkuperäisen funktion tavoin. Sen sijaan, että toimitettaisiin aina alkuperäisen calculation-funktion laskutoimitus, tarkistetaan aina ensin, onko kyseinen laskutoimitus tehty jo. Tämä tarkistus perustuu siihen, että samalla parametrilla value, saadaan aina sama lopputulos. Kun laskutoimitus suoritetaan ensimmäisen kerran tietylle value-arvolle, tallennetaan sen paluuarvo cache-olioon. Kun samaa funktiota kutsutaan uudelleen samalla parametrilla, löytyy arvo jo valmiiksi cache-oliosta, eikä laskutoimitusta tarvitse suorittaa uudelleen.

4.11 Osittainen soveltaminen (Partial Application)

Osittainen soveltaminen tarkoittaa tilannetta, jossa funktiolle annetaan parametreja, ja se palauttaa uuden funktion, joka käyttää jollain tapaa jo annettuja parametreja. Funktio on siis osin toteutettu jo aiemmin saaduilla parametreilla ja paluuarvona saatu funktio tarvitsee vähemmän parametreja, koska osa parametreista on jo saatu sen luonnin

yhteydessä. [Kereki 2017: Transforming functions – Currying and Partial Application. Partial application.]

```
let cakeBuilder = size => {
  return (taste, isGlutenfree) => {
    return `A ${size} ${
      isGlutenfree ? `glutenfree ` : ``
    }cake that tastes like ${taste}.`
  }
}

let smallCakeBuilder = cakeBuilder("small")

console.log(smallCakeBuilder("watermelon", false)) // A small cake that tastes
like watermelon.

console.log(smallCakeBuilder("strawberry", true)) // A small glutenfree cake
that tastes like strawberry.

let hugeCakeBuilder = cakeBuilder("large")

console.log(hugeCakeBuilder("watermelon", true)) // A large glutenfree cake
that tastes like watermelon.

console.log(hugeCakeBuilder("strawberry", false)) // A large cake that tastes
like strawberry.
```

Esimerkkikoodi 31. cakeBuilder saa koon size-parametrina ja palauttaa uuden funktion, joka hyödyntää aikaisemmin saatua size-parametria.

Koodissa 31 cakeBuilder-funktio saa size-parametrin ja palauttaa uuden funktion, joka hyödyntää tätä parametriksi saatua arvoa. Näin voidaan kiinnittää jokin parametri, joka toistuu useasti ja vähentää välitettävien parametrien määrää. Tämä on erityisen hyödyllistä, mikäli esimerkin tapauksessa saman kokoisia kakkuja tehdään useasti, niin vältytään saman parametrin toistolta.

```
let cakeBuilder = (size, taste, isGlutenfree) => {
  return `A ${size} ${
    isGlutenfree ? `glutenfree ` : ``
  }cake that tastes like ${taste}.`
}

let largeCakeBuilder = (taste, isGlutenfree) => {
  return `A large ${
    isGlutenfree ? `glutenfree ` : ``
  }cake that tastes like ${taste}.`
}

let smallCakeBuilder = (taste, isGlutenfree) => {
  return `A small ${
    isGlutenfree ? `glutenfree ` : ``
  }cake that tastes like ${taste}.`
}
```

Esimerkkikoodi 32. Toteutus ilman osittaista soveltamista

Ilman osittaisen soveltamisen hyödyntämistä vaihtoehtoina on käyttää yhtä geneeristä funktiota, jolla joutuu antamaan useamman parametrin tai luoda kaksi erillistä funktioita, joihin on kovakoodattu yksi parametreista. Esimerkkikoodissa 32 on havainnollistettu molemmat tilanteet. Funktio cakeBuilder on geneerinen funktio, jolle täytyy aina antaa kaikki 3 parametria. Funktiot largeCakeBuilder ja smallCakeBuilder käyttävät kovakoodattua arvoa parametrin sijaan. Usean erikoistilanteeseen luodun funktion haittapuolena on kuitenkin se, että itse funktion logiikka on kuitenkin sama molemmille, jolloin koodia joutuu toistamaan ja jokaisen uuden tilanteen tullen täytyy kopioida sama koodilogiikka uudelle tilanteelle. Tämä vaikeuttaa ohjelman ylläpitoa, koska mahdolliset tulevat muutokset pitää tehdä useaan paikkaan.

Osittaisen soveltamisen avuksi voidaan tehdä apufunktio, joka mahdollistaa olemassa olevien funktioiden osittaisen soveltamisen.

```
const greeting = (greeting, name) => `${greeting} ${name}!`

const partialApply = (func, ...arguments) => func.bind(null, ...arguments)

const finnishGreeting = partialApply(greeting, "Moikka")
const englishGreeting = partialApply(greeting, "Hello")

finnishGreeting("maailma") // Moikka maailma!
finnishGreeting("kaikki") // Moikka kaikki!
englishGreeting("world") // Hello world!
```

Esimerkkikoodi 33. Apufunktio partialApply auttaa soveltamaan greeting-funktiota osittain

Esimerkkikoodissa 33 luodaan apufunktio partialApply, jolla voidaan muuntaa olemassa oleva funktio uudeksi funktioksi, jolle on sovellettu osa argumenteista jo valmiiksi. Apufunktio hyödyntää JavaScriptin Function-olion bind-metodia, jolla voidaan luoda partialApply-funktion parametrina saadusta funktiosta uusi versio. bind-metodille voi antaa ensimmäisen parametrin jälkeen parametreja, jotka välitetään sen tuottamalle funktiolle parametreina, kun sitä kutsutaan. Nämä parametrit annettaan järjestyksessä ennen niitä parametreja, jotka annetaan itse funktiokutsun yhteydessä. Esimerkissä 33 luodaan kaksi osittain sovellettua funktiota finnishGreeting ja englishGreeting greeting-funktion pohjalta. Nämä osittain sovelletut funktiot odottavat enää vain name-argumenttia, kun niitä kutsutaan. [Mozilla. Function.prototype.bind().]

4.12 Currying

Currying on tekniikka, jossa funktio, jolla on monta parametria, muutetaan yksiparametrisiksi funktioiksi, eli unaarifunktioiksi. Currying tuottaa paluuarvoksi funktion, jolla on vain yksi parametri. Esimerkiksi kolmiparametrisesta funktiosta saadaan currying-tekniikan avulla kolme yksiparametrista funktiota. Kun moniparametrinen funktio muutetaan yksiparametrisiksi funktioiksi, mahdollistaa se uusien osittain sovellettujen funktioiden luomisen funktion pohjalta. [Kereki 2017: Transforming functions – Currying and Partial Application. Currying.]

```
// Funktio ilman currya
let sum = (x, y) => x + y
sum(5, 6) // 11

// Funktio currylla
let curriedSum = x => y => x + y
let sum5 = curriedSum(5)
sum5(6) // 11
curriedSum(5)(6) // 11
```

Esimerkkikoodi 34. Currying-tekniikan hyödyntäminen sum-funktiolle

Esimerkissä 34 on hyvin yksinkertainen currying-tilanne, jossa kaksiparametrisestä funktiosta tehdään currying-tekniikan avulla kaksi yksiparametrista funktiota. curriedSum-funktiolle annetaan ensin parametrina numero 5, ja funktio palauttaa uuden funktion, johon tämä parametri on osittain sovellettu. Tämä osittain sovellettu funktio tallennetaan muuttujaan sum5. Tälle funktiolle annetaan seuraava parametri, ja saadaan ensimmäisen funktion parametrin ja seuraavan funktion parametrin summa. curriedSum-funktiota voidaan kutsua myös heti kaksi kertaa, mutta parametrit on eroteltava suluilla kuten kutsussa curriedSum(5)(6).

Curryn voi myös toteuttaa funktioon niin, että se tuottaa joko curried-funktion tai lopullisen paluuarvon riippuen siitä, annettiinko funktiolla kaikki sen tarvitsemat parametrit. Tämä ei vastaa suoraan curry-määritelmää, vaan laajentaa toteutusta monipuolisemmaksi. Esimerkiksi kaksiparametrisen funktion sisällä voidaan tarkistaa, saatiinko kaksi, yksi vai ei yhtään parametria. Jos saatiin kaksi, palautetaan paluuarvo normaalisti. Jos saatiin vain yksi, palautetaan osittain sovellettu curry-funktio, joka ottaa parametriksi toisen funktion. Ja jos yhtään parametria ei annettu, voidaan palauttaa

alkuperäinen funktio sellaisenaan. [Kereki 2017: Transforming Functions – Currying and Partial Application.]

Olemassa olevien moniparametrisien funktioiden pohjalta voidaan luoda yksiparametrisia funktioita apufunktion avulla.

```
const curry = func => {
  const turnToUnaryFunctions = (argsCount, args = []) => {
    if (argsCount === 0) {
      return func.apply(null, args)
    }
    return arg => turnToUnaryFunctions(argsCount - 1, [...args, arg])
  }
  return turnToUnaryFunctions(func.length)
}

const sum = (x, y, z) => x + y + z
const curriedSum = curry(sum)
curriedSum(1)(2)(3) // 6
```

Esimerkkikoodi 35. Apufunktio curry luo yksiparametrisia funktioita parametrina saadun funktion pohjalta.

Esimerkkikoodissa 35 luodaan apufunktio curry, joka saa parametrina funktion, jonka pohjalta halutaan luoda yksiparametrisia funktioita. JavaScriptin Function-olion length-ominaisuus kertoo, kuinka monta parametria funktio odottaa, joten sen avulla curry-funktio suorittaa rekursiivisen turnToUnaryFunctions-operaation, jolla luodaan yksiparametrisia funktioita niin monta kappaletta, kuin alkuperäinen funktio odottaa parametreja. Palautettua funktioita voidaan kutsua currying-tyylillä curriedSum(1)(2)(3). Parametreina saadut arvot säilötään listaan, ja kun kaikki parametrit on saatu, ne välitetään alkuperäiselle funktiolle apply-metodilla. Tämä apufunktio käyttää lopussa alkuperäistä moniparametrista funktiota, mutta se kerää sen vaatimat parametrit palauttamiensa unaarifunktioiden avulla. [Mozilla. Function.length; Mozilla. Function.prototype.apply().]

4.13 Laiska suoritus (Lazy Evaluation)

Laiska suoritus tarkoittaa sitä, että jokin operaatio suoritetaan, jos ja vain, jos paluuarvoa tarvitaan oikeasti ohjelman ajon aikana. Tämä toiminto sopii hyvin raskaille

laskutoimituksella, jotka halutaan ajaa vain, jos niitä oikeasti tarvitaan. [Antani ym. 2016; Module 2, Chapter 6, Lazy Instantion; Mantyla 2015, Chapter 2, Lazy Evaluation.]

Thunk on esimerkki laiskasta suorituksesta. Tunk on funktio, joka ei saa parametreja. Tunkissa laskutoimitus sidotaan muuttujaan, mutta laskutoimitusta ei suoriteta tässä vaiheessa ohjelmaa. Vasta kutsumalla thunkkia, laskutoimitus suoritetaan ja saadaan tulos. Sen tarkoitus on siirtää ohjelman osan ajo siihen hetkeen, kun sen laskemaa arvoa todellisuudessa tarvitaan. Tämän hyöty on se, ettei arvoa lasketa etukäteen ja säästytään turhilta laskuilta, mikäli arvoa ei edes käytetä ohjelman suorituksen aikana. [Kereki 2017; Designing Functions – Recursion, Recursion techniques, Trampolines and Thunks.]

```
const minutesInDay = 60 * 24 // immediate evaluation

const lazyMinutesInDay = () => 60 * 24 // lazy evaluation
```

Esimerkkikoodi 36. Tässä esimerkissä havainnollistetaan ohjelman ajoa käyttäen thunk:ia ja ilman sitä.

Esimerkissä 36 määritellään kaksi muuttujaa, minutesInDay ja lazyMinutesInDay. Molemmat suorittavat laskutoimituksen, jolla lasketaan minuuttien määrä vuorokaudessa. Muuttujan minutesInDay arvo lasketaan heti määrityksen yhteydessä, vaikka arvoa ei käytettäisi koko ohjelman aikana. Muuttuja lazyMinutesInDay määritellään funktioksi, joka suorittaa saman laskutoimituksen kuin minutesInDay ja palauttaa laskutoimituksen tuloksen. Tätä arvoa ei kuitenkaan lasketa vielä määrityksen yhteydessä vaan vasta kun ohjelmassa kutsutaan kyseistä funktiota. Laskutoimitus siirretään siis hetkeen, jolloin arvio oikeasti tarvitaan, ja mikäli sitä ei tarvita ohjelman ajon aikana ollenkaan, sitä ei myöskään lasketa.

5 Map, Filter, Reduce

Map-, filter- ja reduce-funktiot ovat yleisiä funktionaalisen ohjelmoinnin operaatioita. Niitä käytetään monissa eri ohjelmointikielissä eikä JavaScript ole poikkeus. JavaScriptin map-, filter- ja reduce-funktiot ovat sisäänrakennettuja funktiota, ja ne ovat Arrayn metodeja. Niiden avulla voidaan luoda uusia, muunneltuja taulukoita alkuperäisen

taulukon pohjalta. Yhdessä map-, filter- ja reduce-funktiot luovat tehokkaan työvälineen tiedon käsittelylle. Yksinkertaisuudessaan map-funktio suorittaa operaation jokaiselle taulukon alkiolle, filter-funktio suodattaa taulukon alkioita ja reduce-funktio kasaa alkioista jonkin yhden lopputuloksen. Nämä kaikki funktiot pitävät alkuperäisen tietorakenteen muuttumattomana ja luovat vain uutta tietoa sen arvojen pohjalta.

5.1 Map

Map-funktio on Arrayn metodi. Mapin avulla taulukoita on helppo käsitellä funktionaalisesti. Taulukon arvoille voidaan suorittaa operaatioita deklaratiivisella syntaksilla. Map-funktiolle annetaan parametrina funktio, joka toteutetaan jokaiselle tietorakenteen alkiolle. Map ei siis yksin tee mitään, vaan se tarvitsee toteutettavan funktion alkioille. Map ei myöskään muuta alkuperäistä tietorakennetta, vaan käyttää vain sen arvoja, suorittaa annetun funktion jokaiselle alkiolla ja palauttaa uuden taulukon sen pohjalta. [Kereki 2017: Programming Declaratively – A Better Style. Transformations. Applying an operation – map; Mozilla. Array.prototype.map(); Mantyla 2015. s. 29.]

```
const values = [23, 95, 24, 0.6, 254, 60]

const doubledValues = values.map(value => value * 2)

console.log(values, doubledValues) // [ 23, 95, 24, 0.6, 254, 60 ] [ 46, 190, 48, 1.2, 508, 120 ]
```

Esimerkkikoodi 37. Uuden taulukon luonti mapin avulla. Alkuperäisen taulukon arvot kerrotaan kahdella mapissa.

Map-funktio korvaa for- ja forEach-silmukat. Map käy läpi jokaisen taulukon arvon, antaa sen arvon mapissa käytettävälle funktiolle ja muodostaa paluuarvoista uuden taulukon. Esimerkissä 37 jokainen values-taulukon arvo käydään läpi, kerrotaan kahdella ja muodostetaan uusi taulukko doubledValues. Lokittaessa values- ja doubledValues-taulukot, huomataan, että alkuperäinen values-taulukko on pysynyt alkuperäisessä muodossa ja sen pohjalta on luotu uusi doubledValues-taulukko.

Saman toiminnon toteuttaminen perinteisellä for-silmukalla muokkaa usein alkuperäistä taulukkoa, eikä ratkaisu ole yhtä elegantti. Toki alkuperäisen taulukon muokkaamisen sijaan voidaan luoda uusi taulukko, johon tuloksen tallennetaan, mutta for-silmukka ei erityisesti ohjaa ohjelmoijaa siihen suuntaan. Ratkaisussa pitää myös pelata taulukon indeksien kanssa, mikä on huomattavasti työläämpää kuin map-funktion käyttö.

```
const values = [23, 95, 24, 0.6, 254, 60];
for (i in values) {
  values[i] = values[i] * 2;
}

console.log(values); // [ 46, 190, 48, 1.2, 508, 120 ]
```

Esimerkkikoodi 38. Map-toimintoa vastaava toteutus for-silmukalla.

Esimerkissä 38 suoritetaan taulukon alkioiden käsittely perinteisesti for-silmukalla. Map-metodin avulla voidaan käydä läpi taulukoita samaan tapaan, mutta helpommin ja muuttamatta alkuperäistä taulukkoa.

Mapilla voidaan myös manipuloida taulukossa olevia olioita.

```
const people = [
  {
    name: "Sandra",
    city: "Helsinki",
  },
  {
    name: "Mikko",
    city: "Vantaa",
  },
  {
    name: "Olli",
    city: "Espoo",
  },
]

const addId = (person, id) => {
  return { ...person, id: id }
}

const peopleWithId = people.map((person, index) => addId(person, index))
```

Esimerkkikoodi 39. Taulukossa olevien alkioiden käsittely

Esimerkissä 39 on taulukko ”people”, johon on säilötty kolme olioita, joilla on name- ja city-attribuutit. Nyt näille olioille halutaan lisätä id-attribuutti. Se lisätään tässä

tapauksessa yksinkertaisesti taulukon indeksin mukaan. Sen lisäksi, että map antaa käsiteltäväksi jokaisen taulukon alkion yksitellen, se antaa myös indeksin. Esimerkissä 39 välitetään yksittäinen alkio sekä indeksi addId-funktiolle, joka puolestaan luo alkuperäisen olion pohjalta uuden olion, jolle se lisää id-attribuutin ja antaa sille indeksin arvon. Tuloksena saadaan uusi kolmealkioinen taulukko, jossa ovat oliot { name: 'Sandra', city: 'Helsinki', id: 0 }, { name: 'Mikko', city: 'Vantaa', id: 1 } ja { name: 'Olli', city: 'Espoo', id: 2 }.

5.2 Filter

Array-tietorakenteen metodi Filter suodattaa alkioita. Filter-funktiolle annetaan funktio, joka testaa alkiot yksi kerrallaan annetulla ehdolla. Ne alkiot, jotka palauttavat totuusarvon tosi, pääsevät uuteen palautettavaan array-rakenteeseen. Muut alkiot, jotka palauttavat epätosi-arvon suodattuvat pois. [Kereki 2017: Programming Declaratively – A Better Style. Logical higher-order functions. Filtering an array; Mantyla 2015: Sivu 30. Mozilla. Array.prototype.filter().]

```
const arr = [-4, 5, -6, 7]

console.log(arr.filter(x => x >= 0)) // [5, 7]
```

Esimerkkikoodi 40. Negatiiviset luvut suodatetaan pois

Esimerkkikoodissa 40 käytetään filter-funktiota, jolla annetaan funktio, joka tarkistaa ehdon x >= 0. Jokainen taulukon alkio käydään läpi ehdolla. Mikäli vertaus palauttaa true, pääsee arvo uuteen palautettavaan taulukkoon. Alkuperäisestä arr-taulukosta [-4, 5, -6, 7] saadaan suodatuksen jälkeen uusi taulukko [5, 7] muuttamatta alkuperäistä taulukkoa.

Filter-funktiota voidaan käyttää hyväksi myös taulukoissa, joka pitää sisällään olioita. Seuraavassa esimerkissä käytetään taulukkoa, jonka alkioina on olioita. Oliot pitävät sisällään henkilön id:n sekä nimen. Filter-funktion avulla etsitään henkilö id:n avulla.

```
var people = [
  {id: 1, name: "Petri"},
```

```
  {id: 2, name: "Joonas"},
  {id: 3, name: "Pirjo"},
  {id: 4, name: "Hanna"}
]

console.log(people.filter(x => x.id == 3))
```

Esimerkkikoodi 41. Filteröinti taulukon sisällä olevien olioiden mukaan.

Esimerkissä 41 haetaan people-taulukosta henkilö, jonka id on 3. Filter-metodille annetaan funktio, joka tarkistaa onko olion id-attribuutin arvo 3. Jos ehto täyttyy, funktio päästää olion suodatuksen läpi. Filter palauttaa taulukon, joka sisältää yhden henkilön tiedot [{id: 3, name: ’Pirjo’}].

5.3 Reduce

Reduce-funktio on Array:n metodi, jolla on tarkoitus tuottaa jokin yksi tulos koko taulukosta. Reduce käyttää laskuria (accumulator), johon kootaan lopputulos. Reducelle on annettava minimissään parametriksi funktio, joka suorittaa halutun laskutoimituksen taulukon alkioille, mutta sen lisäksi voidaan antaa initialValue-argumentti, mikäli accumulator halutaan alustaa jollain tietyllä arvolla. Parametrina saadulle funktiolle on annettava ainakin kaksi argumenttia, accumulator sekä currentValue. Funktiolle on myös tämän lisäksi mahdollista antaa currentIndex sekä array. [Dan Mantyla 2015, Chapter 2 Array.prototype.reduce()] [Kereki 2017: Programming declaratively – A Better Style, Transformations, Reducing an array to a value; Mozilla. Array.prototype.reduce(); Mantyla 2015. s. 31.]

Accumulator kasaa funktion paluuarvot yhdeksi tulokseksi. CurrentValue on taulukon alkion arvo, jota juuri nyt käsitellään. CurrentIndex on taulukon alkion indeksi, jota juuri nyt käsitellään. Array on se taulukko, jonka yhteydessä reducea kutsuttiin.

```
let arr = [-4, 5, -6, 7]
let result = arr.reduce((accumulator, currentValue) => accumulator +
currentValue)
```

Esimerkkikoodi 42. Reduce-esimerkki

Esimerkissä 42 käytetään reduce-operaatiota kokoamaan taulukossa olevien lukujen summa. Tässä esimerkissä käytetään vain pakollisia argumentteja, eli funktiota, jolle annetaan accumulator sekä currentValue. Accumulator pitää jo laskettua summaa yllä, ja currentValue on aina taulukon tämänhetkisen alkion arvo. Mikäli accumulatorille ei aseteta alkuarvoa, niin alkuarvo on taulukon ensimmäisen alkion arvo. Muuttuja result saa reducen jälkeen arvokseen 2. Tähän tulokseen päästään kolmen laskutoimituksen kautta, -4 + 5 = 1, 1 + -6 = -5, -5 + 7 = 2.

```
let arr = [-4, 5, -6, 7]
let result = arr.reduce((accumulator, currentValue) => accumulator + currentValue, 100)
```

Esimerkkikoodi 43. Reduce-esimerkki alkuarvolla alustettuna

Esimerkissä 43 reducelle annetaan alkuarvoksi 100. Näin ollen result-muuttujan arvoksi saadaan 102. Nyt tapahtuu neljä laskutoimitusta 100 + -4 = 96, 96 + 5 = 101, 101 + -6 = 95, 95 + 7 = 102.

JavaScriptin reduce-funktio on oletuksena foldl eli fold left, mikä tarkoittaa, että taulukon läpikäynnin järjestys on vasemmalta oikealle. Monissa tapauksissa läpikäynnin suunnalla ei ole merkitystä, mutta on myös tilanteita, joissa sillä on väliä. JavaScriptissä on myös reduceRight-funktio (foldr, fold right), joka toimii samoin kuin reduce, mutta suunta on oikealta vasemmalle. [Frederico Kereki 2017: Programming declaratively – A Better Style, Transformations, Folding left and right.]

5.4 Map, Filter ja Reduce yhdessä

Map-, filter- ja reduce-funktiot toimivat tehokkaasti yhdessä. Otetaan esimerkiksi taulukko käyttäjän urheilusuorituksia. Jokaisella suorituksella on aikaleima, kilometrimäärä sekä urheilusuorituksen laji. Halutaan selvittää käyttäjän kokonaiskilometrit tietyllä urheilulajilla. Imperatiivisessa ohjelmoinnissa lähdetään tyypillisesti ratkaisemaan ongelma for-silmukalla. Välissä tarvitaan välimuuttujia eivätkä tietorakenteet ole muuttumattomia kuten seuraavassa esimerkissä käy ilmi.

```
const fitnessCalendar = [
  {
    timestamp: 1578936994,
    kilometers: 13,
    type: "running"
  },
  {
    timestamp: 1578937029,
    kilometers: 8,
    type: "biking"
  },
  {
    timestamp: 1578937050,
    kilometers: 4.5,
    type: "biking"
  },
  {
    timestamp: 1578937165,
    kilometers: 5,
    type: "running"
  },
  {
    timestamp: 1578937185,
    kilometers: 7.4,
    type: "running"
  }
]

const runningKilometers = []

for (i in fitnessCalendar) {
  if (fitnessCalendar[i].type === "running") {
    runningKilometers.push(fitnessCalendar[i].kilometers);
  }
}

let runningKilometersSum = 0
for (i in runningKilometers) {
  runningKilometersSum += runningKilometers[i];
}

console.log(runningKilometersSum) //25.4
```

Esimerkkikoodi 44. For-silmukalla laskettu kokonaiskilometrimäärä.

Imperatiivisessa esimerkkikoodin 44 toteutuksessa käytetään globaalia tilaa runningKilometers, jota muutetaan ohjelman ajon aikana. Toteutuksessa ei ole uudeleenkäytettäviä funktioita, eikä ohjelmaa ymmärrä kuin vain lukemalla sen kokonaan läpi.

```
const fitnessCalendar = [
  {
    timestamp: 1578936994,
    kilometers: 13,
    type: "running"
  },
```

```
  {
    timestamp: 1578937029,
    kilometers: 8,
    type: "biking"
  },
  {
    timestamp: 1578937050,
    kilometers: 4.5,
    type: "biking"
  },
  {
    timestamp: 1578937165,
    kilometers: 5,
    type: "running"
  },
  {
    timestamp: 1578937185,
    kilometers: 7.4,
    type: "running"
  }
]

const totalRunningKilometers = fitnessCalendar
  .filter(entry => entry.type === "running")
  .map(entry => entry.kilometers)
  .reduce((sum, entry) => sum + entry)

console.log(totalRunningKilometers) // 25.4
```

Esimerkkikoodi 45. Filter, map ja reduce ketjutettuna

Kuten esimerkistä 45 voi nähdä, filter-, map- ja reduce-funktiot toimivat tehokkaasti yhdessä. Niillä ohjelmointi on deklaratiivista, selkeää ja helposti luettavaa. Funktiokutsut voidaan ketjuttaa, jolloin edellisen funktion palauttama arvo annetaan seuraavalle funktiolle parametrina. Monimutkaisiakin laskutoimituksia saadaan suoritettua vain muutamalla funktiokutsulla.

5.5 MapReduce-paradigma

On olemassa ohjelmointiparadigma MapReduce, joka pohjautuu map- ja reduce-funktioiden logiikkaan. Sillä käsitellään yleensä suurta joukkoa dataa, eli bigdataa. Mapilla järjestellään sekä suodatetaan tietoa ja reducella lasketaan jokin lopputulos mapin palauttamasta tiedosta. Tässä paradigmassa tärkeää on se, että map-laskutoimituksia voidaan hajauttaa ja ajaa rinnakkain, jolloin suuren tietovolyymin käsittely on huomattavasti nopeampaa. JavaScriptissä ei voida jakaa ohjelman ajoa erillisille säikeille, jotta se toimisi rinnakkain eikä sillä siksi voi toteuttaa täysin

MapReduce-paradigman mukaista toimintoa, jolla voisi käsitellä suurta joukkoa tietoa. JavaScriptin mapin, filterin ja reducen avulla voidaan tehdä kuitenkin samanlaista datan käsittelyä pienemmässä mittakaavassa. Dataa voidaan muuttaa ja suodattaa sekä koostaa siitä jokin lopputulema reducella.

6 Rekursio

Kun JavaScriptissä halutaan suorittaa jokin ohjelman osa toistuvasti, käytetään tyypillisesti for- ja while-silmukoita, jotka ovat yleisiä imperatiivisen ohjelmointikielen toimintoja. For-silmukka toimii hyvin, kun tiedetään haluttujen toistojen määrä etukäteen, kun taas while-silmukka sopii tilanteisiin, jossa toistojen määrää ei tiedetä entuudestaan vaan halutaan toistaa koodi, kunnes jokin määrätty ehto täyttyy.

On olemassa tilanteita, joissa imperatiivinen ratkaisu toistuvan ohjelman toteuttamiseen saattaa käydä hankalaksi. Hyvä esimerkki on puurakenteen läpikäyminen. Puu voi olla suuri, ja se saattaa haarautua monta kertaa. Sen sijaan rekursio, funktionaalisen ohjelmoinnin vastine silmukoille, toimii puurakenteiden läpikäymiseen mainiosti.

6.1 Rekursio

Rekursio tarkoittaa ohjelmoinnissa sitä, että funktio kutsuu itse itseään. Rekursiossa on oltava ehto, jolla itsensä kutsuminen päättyy. Tätä tapausta kutsutaan base caseksi. Mikäli se puuttuisi, jatkaisi funktio itsensä kutsumista ikuisesti. Rekursio on funktionaalinen vastine for- ja while-silmukoille ja kaikki tilanteet, jotka voidaan toteuttaa for- ja while-silmukoilla, voidaan myös toteuttaa rekursiivisesti. Kun base case saavutetaan, rekursio purkautuu. Paluuarvo palautuu jokaisesta funktiokutsusta niin, että lopulta ensimmäinen funktion kutsu palauttaa paluuarvon. On tilanteita, joissa rekursiivinen ratkaisu on helpompi toteuttaa kuin JavaScriptille tyypilliset for- tai while-silmukat. Tällaisia tapauksia ovat esimerkiksi tietyntyyppiset matemaattiset algoritmit, puurakenteet, listat tai syntaksianalyysi. [Mantyla 2015: Sivu 24 ja 100; Kereki 2017: Defining functions – Recursion, Using Recursion; Simpons 2018: Chapter 8: Recursion.]

```
function recursion(x) {
  if (x === 0) {
    return x
  }
  return (x * x) + recursion(x - 1)
}
```

Esimerkkikoodi 46. Funktio kutsuu lopussa itse itseään, jolloin syntyy rekursio. Base case, eli tilanne, jossa rekursio päättyy ja purkautuu, toteutuu mikäli x on 0. Silloin funktio ei kutsu enää itse itseään.

Edellisessä esimerkissä 46 demonstroidaan rekursiivisen funktion toimintaa. Funktiolla on base case if-lauseessa, mikäli x:n arvo on nolla, lopetetaan funktion itsensä kutsuminen ja palautetaan vain paluuarvo. Muussa tapauksessa paluuarvon yhteydessä funktio kutsuu itseään.

Rekursio voidaan myös toteuttaa niin, että kaksi funktiota kutsuvat toisiaan syklissä. Tällaisen rekursion nimi on ”mutual recursion”. [Simpons 2018: Chapter 8: Recursion.]

6.2 Kutsupino (Call stack) ja kutsukehys (Stack Frame)

Jokaisessa funktionkutsussa syntyy uusi kutsukehys kutsupinoon. Se pitää muistissa funktion tilaa. Jos funktio kutsuu sisällään toista funktiota, tulee toisen funktion kutsukehys kutsupinon päälle ja sitä kutsunut funktio jää yhä pinoon, kunnes sen suoritus on kokonaan ohi. Rekursiossa pino kasvaa aina yhdellä kutsukehyksellä, kun funktio kutsuu itseään uudelleen. Kun viimeinen funktiokutsu on kokonaan käsitelty, poistetaan se pinosta ja pino alkaa pienentyä yksi kutsukehys kerrallaan. Kutsupino ja kutsukehys pitää huolen siitä, että jokainen keskeytetty funktio voi jatkaa siitä, mihin se jäi kutsuttuaan toista funktiota ja keskeytettyään oman toimintansa. [Simpons 2018: Chapter 8: Recursion. Stack.]

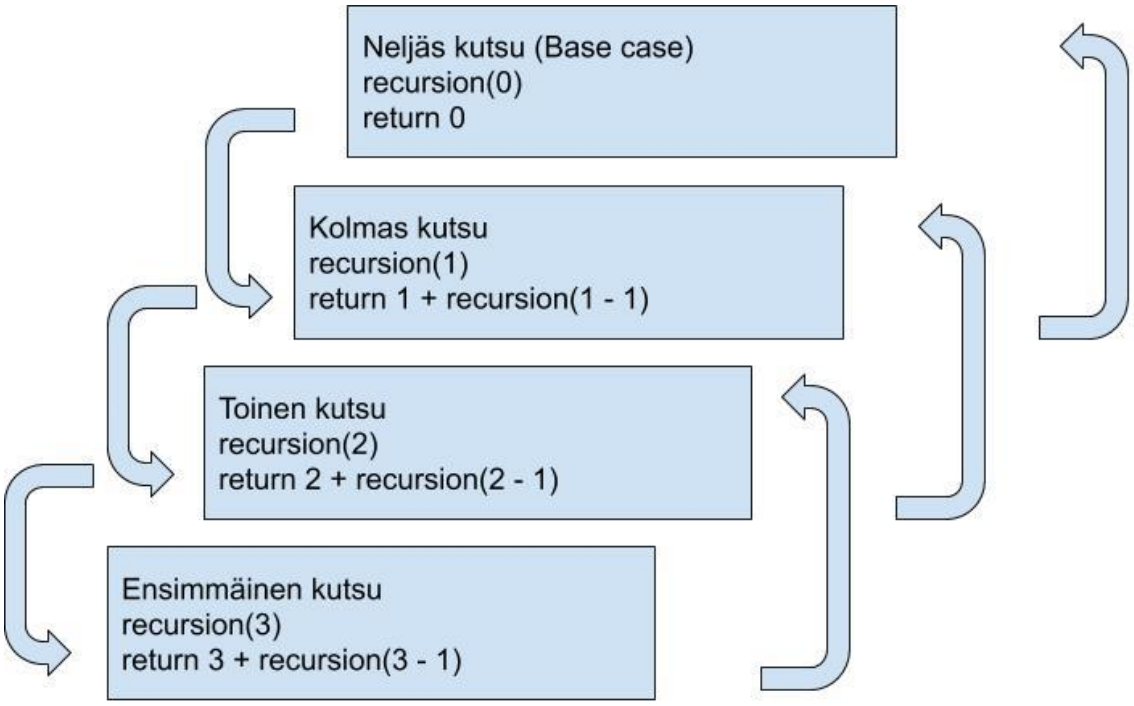
Kutsupino voi kasvaa rekursiossa suureksi. Se voi kasvaa niin suureksi, että pino täyttyy ja ohjelma kaatuu. On mahdotonta arvioida tarkkaan missä vaiheessa kutsupino on liian suuri, koska rajoitus määräytyy JavaScript-moottorin mukaan. [Simpons 2018: Chapter 8: Recursion. Stack.]

```
const recursion = number => {
```

```
  if (number === 0) {
    return number
  }
  return number + recursion(number - 1)
}

recursion(3)
```

Esimerkkikoodi 47. Tämä rekursio luo pinoon 4 kutsukehystä

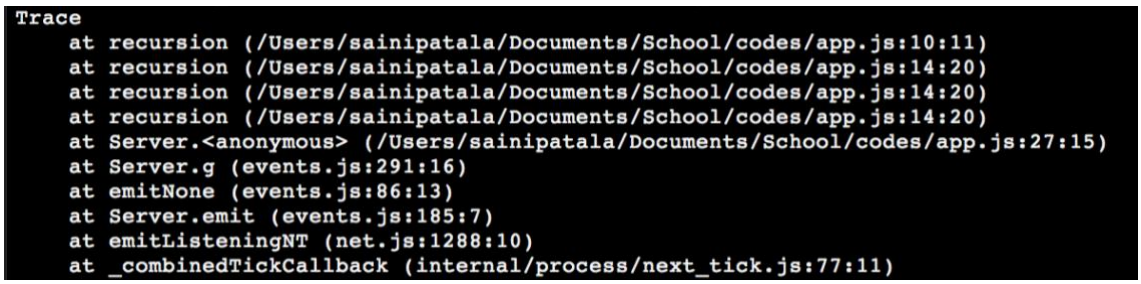
Esimerkkikoodin 47 rekursiivinen funktio saa ensimmäisessä kutsussa parametrina numeron, joka on tässä tapauksessa 3, ja sen tehtävänä on summata kaikki kokonaisluvut tästä numerosta alaspäin nollaan saakka. Eli esimerkin tilanteessa ohjelma summaa 3 + 2 + 1 + 0 ja saa lopputuloksen 6.



Kuva 6. Kuvassa havainnollistetaan esimerkkikoodin 44 kutsupinoa

Kuva 6 havainnollistaa rekursion etenemistä. Alin ensimmäinen kutsuu itseään uudelleen luoden uuden kutsukehyksen. Seuraava kutsukehys kutsuu taas itseään luoden kolmannen kutsukehyksen ja sama jatkuu, kunnes neljännessä kutsukehyksessä

saavutetaan base case eikä kutsuta enää itseä. Siinä tilanteessa funktio palauttaa paluuarvonsa edelliselle kehykselle, ja se palauttaa oman paluuarvonsa taas edelliselle, kunnes rekursio on purkautunut täysin.



Kuva 7. Kuvassa näkyy neljä peräkkäistä recursion-funktion kutsua

Kuvassa 7 näkyy ohjelman kutsuloki, joka saadaan näkyviin kutsumalla console.trace-funktiota. Siinä näkyy heti ylimpänä neljä kappaletta recursion-funktion kutsuja. Yksittäinen recursion-funktio lisätään pinoon sitä kutsuessa ja poistuu pinosta vasta, kun sen koko suoritus on päättynyt, eli tässä tapauksessa, kun rekursio purkautuu base case:n tapahduttua. Tällöin recursion-kutsukehykset poistuvat pinosta yksitellen, päällimmäisestä recursion-kutsusta alkaen ja lopuksi poistaen ensimmäisen recursion-kutsun, joka jäi pinon alimmaksi.

6.3 Häntärekursio (Tail-Call)

Häntärekursio on rekursion toteutus, jonka tarkoituksena on kutsua itseään aivan nykyisen funktion lopussa niin, että kutsun suorittavasta funktiosta ei jää ylläpidettävää tilaa. [Simpons 2018: Chapter 8: Recursion. Tail Calls.]

Seuraavassa esimerkkikoodissa on esitetty ensin rekursio, joka ei hyödynnä häntäkutsua, sekä sen jälkeen saman ominaisuuden toteuttava rekursio, joka on toteutettu häntärekursiona.

```
function recursion(x) {
  if (x === 0) {
    return x
  }
```

```
  return (x * x) + recursion(x - 1)
}

function tailrecursion(x, sum = 0) {
  if (x === 0) {
    return sum
  }
  return tailrecursion(x - 1, sum + x * x)
}
```

Esimerkkikoodi 48. Esimerkki normaalista rekursiototeutuksesta sekä häntärekursion toteutuksesta.

Ensimmäisessä esimerkkikoodin 48 recursion-funktiossa ei käytetä häntärekursiota. Ohjelman on pidettävä muistissa jokaisen kutsutun funktion tilaa, kunnes rekursio purkautuu, jotta (x * x) voidaan summata recursion(x – 1)-kutsun tulokseen. Tailrecursion-funktiossa funktion viimeinen tehtävä on kutsua itseään uudelleen ja välittää oma tila parametrina seuraavalle funktiokutsulle.

6.4 TCO (Taill call optimization)

Häntärekursiossa pinoon jäävien edellisten rekursiokutsujen funktiot eivät pidä enää yllä dataa, jota rekursio tarvitsisi purkautuessaan. Tällöin pinoon jäävät aiemmat funktiot voitaisiin pyyhkiä pois pinosta, koska rekursion seuraavat funktiokutsut eivät ole riippuvaisia päättyvän funktion tilasta. Käytännössä tämä ei kuitenkaan toimi JavaScriptissä vaan kutsupino kasvaa silti jättäen jokaisen rekursiokutsun pinoon.

TCO on funktionaalisessa ohjelmoinnissa toiminto, jolla voidaan estää pinon kasvaminen häntärekursiossa. Sen sijaan, että jokainen rekursion kutsu jäisi pinoon, päättyvä funktiokutsu korvataan vain uudella kutsulla. JavaScript ei käytä TCO:ta, vaikka häntärekursio olisi toteutettu.

6.5 PTC (Proper tail call)

PTC, eli proper taill call, tarkoittaa sitä, että häntärekursio on oikeasti optimoitu, eikä pino kasva. Vaikka rekursion toteuttaa JavaScriptissä häntärekursiota käyttäen, se ei

pelkästään riitä estämään pinon kasvamista [Simpons 2018: Chapter 8: Recursion. Proper Tail Calls.]

PTC ei ole laajalti tuettu JavaScriptissä. Käyttämällä strict-tilaa Safari (Version 13.1) tukee PTC:tä, mutta Chrome (Version 80.0.3987.163) tai Firefox (72.0.2) eivät. Node on myös tukenut PTC:tä joissain vanhemmissa versioissa, kuten esimerkiksi versiossa v6.10.1, kun käytetään strict-tilaa ja ohjelma ajetaan -harmony-flagia käyttäen. Uudemmat Node-versiot, kuten v12.9.1, eivät tue tätä toimintoa enää.

Safarilla tai vanhemmalla node-versiolla voidaan testata PTC ajamalla rekursiivinen funktio tarpeeksi monta kertaa aiheuttaen stack overflow -virheen.

```
function tailrecursion(x, sum = 0) {
   if (x === 0) {
      return sum
   }
   return tailrecursion(x - 1, sum + x * x)
 }

 console.log(tailrecursion(30000))
```

Esimerkkikoodi 49. Pinon kasvattaminen rekursiolla, joka kutsuu itseään monta kertaa

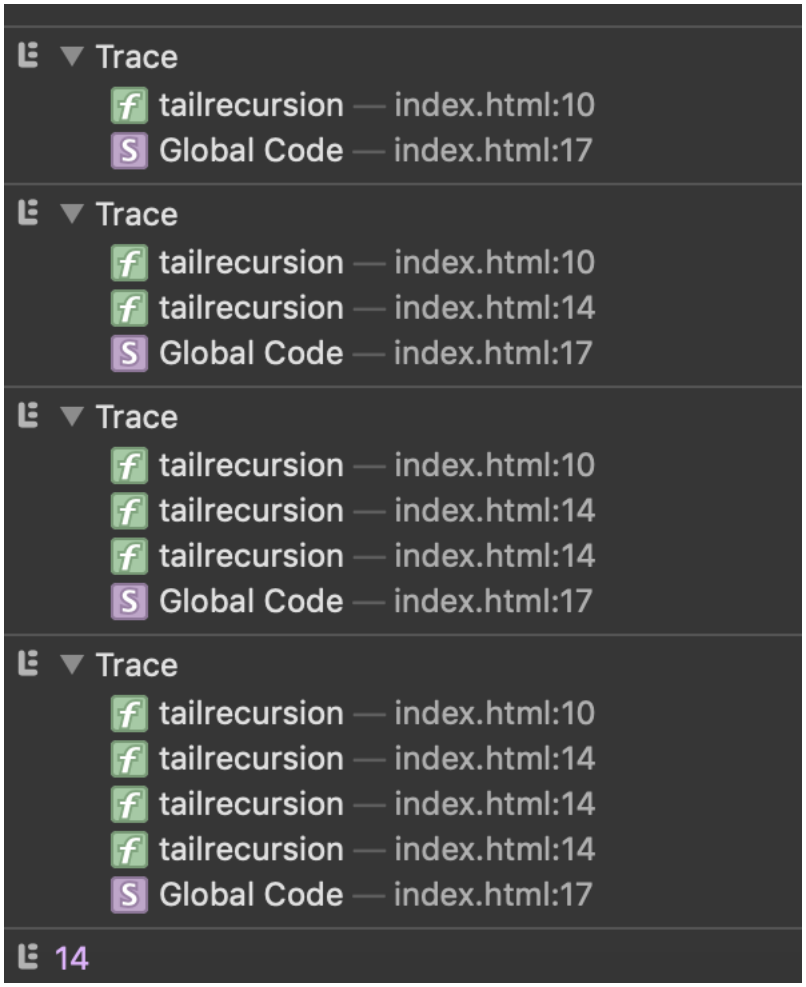
Pinon ylittyminen riippuu ympäristöstä, joten eri ympäristöissä pino kestää enemmän tai vähemmän kasvua. Tässä tapauksessa esimerkin 49 rekursio aiheuttaa Safarissa virheen RangeError: Maximum call stack size exceeded, kun sille annetaan parametrina 30000. Pino on ylittynyt ja ohjelma kaatuu. Kun ohjelman alkuun lisätään rivi ’use strict’, eli strict-tila otetaan käyttöön ja ohjelma ajetaan uudelleen, ei virhetilannetta synny enää ja saadaan tulos 9000450005000, eli PTC toimii eikä pino kasva jokaisella rekursiokutsulla.

Ohjelman pinon voidaan tarkastella console.trace()-metodilla.

```
function tailrecursion(x, sum = 0) {
   console.trace()
   if (x === 0) {
      return sum
   }
   return tailrecursion(x - 1, sum + x * x)
}

console.log(tailrecursion(3))
```

Esimerkkikoodi 50. Console.trace-funktio rekursion sisällä

Esimerkin 50 koodissa laitetaan console.trace()-kutsu rekursion sisään, jotta jokaisella rekursiivisella kutsulla voidaan tarkistaa pino. Tailrecursion-funktiota kutsutaan parametrilla 3, jolloin funktiossa käydään 4 kertaa.



Kuva 8. Console.trace-funktion tulostukset konsolissa.

Kuvassa 8 näkyy esimerkkikoodin 50 tuottamat tulostukset lokiin. Jokainen tailrecursion-kutsu jättää uuden funktiokutsun pinoon, joka näkyy console.tracen tulostuksessa.

```
'use strict'

function tailrecursion(x, sum = 0) {
 console.trace()
 if (x === 0) {
```

```
    return sum
 }
 return tailrecursion(x - 1, sum + x * x)
}

console.log(tailrecursion(3))
```

Esimerkkikoodi 51. Strict-tilan lisäys ohjelmaan

Kun käytetään strict-tilaa Safarilla, saadaan PTC taas käyttöön, ja se vaikuttaa myös console.tracen tulosteeseen.

Kuva 9. Console.trace-funktion tulostukset konsolissa, kun käytetään PTC:tä.

Kuvassa 9 näkyy konsolin tulosteet, kun ajetaan esimerkkikoodi 51. Console.trace-funktio tulostaa tällä kertaa neljä kertaa saman näköisen pinon, jossa on vain yksi tailrecursion kerrallaan sekä Global Code. Ennen PTC:tä, tailrecursion-kutsut kasvoivat jokaisella rekursiokutsulla niin, että lopuksi pinossa näkyi neljä tailrecursion-kutsua ja Global Code. PTC:ssä rekursion käyttämä pinon koko pysyy vakiona, eikä pino voi kasvaa hallitsemattomasti rekursion osalta.

6.6 Trampoliini

Trampoliini on keino toteuttaa rekursiivinen logiikka ilman pinon täyttymistä. Tällöin ei olla riippuvaisia selaimen tai noden PTC-tuesta, vaan ohjelma on toteutettu niin, ettei pino kasva alkujaankaan. Trampoliinissa funktio ei kutsu itseään rekursiivisesti, vaan se palauttaa thunkin, jota kutsutaan vasta edellisen funktion suorituksen päätyttyä, jolloin pino ei kasva. Kertauksena aiempaan, thunk on funktio, joka ei saa parametreja ja, jonka avulla voidaan viivästyttää ohjelman osan ajoa hetkeen, jona sitä oikeasti tarvitaan.

```
const trampoline = (result) => {
```

```
  while (typeof result === "function") {
    result = result()
  }
  return result
}

const sumThunk = (x, acc = 0) => {
  return () => {
    if (x === 0) {
      return acc
    }
    return sumThunk(x - 1, acc + x)
  }
}

const sumRecursion = (x, acc = 0) => {
  if (x === 0) {
    return acc
  }
  return sumRecursion(x - 1, acc + x)
}

console.log(trampoline(sumThunk(30000))) // 450015000
console.log(sumRecursion(30000)) // RangeError: Maximum call stack size
exceeded…
```

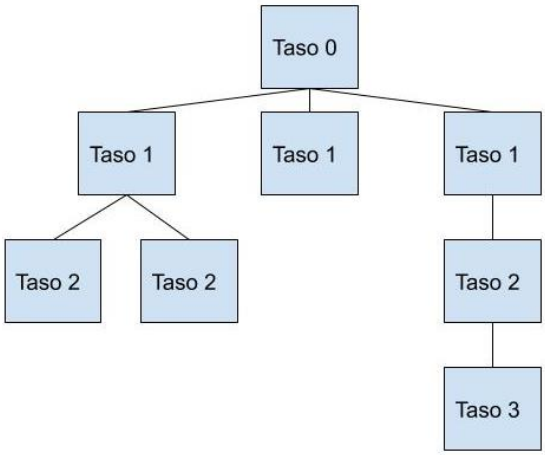
Esimerkkikoodi 52. Trampoliinin ja tunkin vertaus normaaliin rekursioon.

Esimerkkikoodissa 52 luodaan trampoline-funktio, joka saa parametrina joko funktion, eli tässä tapauksessa thunkin, tai lopputuloksen, kun base case on saavutettu. Jos trampoliini saa thunkin, se ajaa sen tai muutoin se palauttaa vain lopputuloksen. Funktio sum-Thunk toteuttaa saman, kuin perinteinen rekursio sumRecursion, mutta se palauttaa thunkin. Siinä missä pino täyttyy ja ohjelma kaatuu, kun sumRecursion-funktiota kutsutaan parametrilla 30000, trampoliinin ja thunkin yhdistelmässä pino ei täyty ja rekursiivista laskutoimitusta voidaan suorittaa kuinka pitkälle vain.

6.7 Rekursioesimerkki

Rekursio on hyvä keino puutyylisten rakenteiden läpikäymiseen. Seuraavassa esimerkissä käydään rekursiivisesti läpi oliota, jolla on sisäkkäisiä olioita. Olio kuvaa keskustelupalstan lankaa, jossa jokaisella langan kommentilla voi olla alikommentteja.

Kuvitellussa tilanteessa kommenttiketjun rakenne on seuraavanlainen:



Kuva 10. Visualisointi puurakenteesta. Puussa on neljä tasoa 0-3.

Kuvassa 10 siniset kuutiot esittävät keskustelupalstan yksittäisen langan kommentteja. Jokaisella kommentilla voi olla yksi tai useampi lapsi, joka on taas uusi kommentti. Taso 0 edustaa keskustelulangan avaavaa kommenttia ja taso 1 edustaa sen lapsia/alikommentteja. Tason 2 kommentit ovat puolestaan lapsia tason 1 kommenteilla ja niin edes päin.

Tässä esimerkkitilanteessa kommenttiketju on olio, jolla on ikään kuin puurakenne. Se halutaan kuitenkin muuttaa listaksi, joka pitää sisällään kaikki kommentit, ja kommentti pitää tiedoissaan parent-kommentin id:n, jolloin puurakenne puretaan. Seuraava kaavio esittää haluttua lopputulemaa:

Kuva 11. Puurakenne puretaan yksittäisiksi olioiksi

Kuvassa 11 havainnollistetaan purettua puurakennetta, jossa rakenteen sijaan oliot pitävät parent_id:n tiedossa. Tässä muunnoksessa olio ja sen sisäkkäiset oliot tulee

käydä kauttaaltaan läpi, eikä tiedetä etukäteen, miten syvä kommenttiketju on. Rekursio osoittautuu näppäräksi ratkaisuksi tilanteeseen.

```
let thread = {
  id: 1,
  content: "first comment",
  children: [
    {
      id: 2,
      content: "second comment",
      children: [
        {
          id: 3,
          content: "third comment",
          children: []
        },
        {
          id: 4,
          content: "fourth comment",
          children: []
        }
      ]
    },
    {
      id: 5,
      content: "fifth comment",
      children: []
    },
    {
      id: 6,
      content: "sixth comment",
      children: [
        {
          id: 7,
          content: "seventh comment",
          children: [
            {
              id: 8,
              content: "eigth comment",
              children: []
            }
          ]
        }
      ]
    }
  ]
}
```

Esimerkkikoodi 53. Keskustelulangan kommentit tallennettuna thread-muuttujaan.

Koodissa 53 esitetään keskustelulangan kommentit. Lanka koostuu kolmesta päätason kommentista. Kommenteilla voi olla niin sanottuja lapsikommentteja, eli alikommentteja. Lapsikommenteilla voi olla puolestaan omia lapsikommentteja.

```
const convertThreadToArray = (comment, parent = -1) => {
```

```
  if (comment.children.length < 1) {
    return [
      {
        id: comment.id,
        content: comment.content,
        parent_id: parent
      }
    ]
  } else {
    return comment.children
      .map(child => {
        return convertThreadToArray(child, comment.id, [
          {
            id: comment.id,
            content: comment.content,
            parent_id: parent
          }
        ])
      })
      .reduce((accumulator, current) => [...accumulator, ...current], [
        {
          id: comment.id,
          content: comment.content,
          parent_id: parent
        }
      ])
  }
}

convertThreadToArray(thread)
```

Esimerkkikoodi 54. Rekursiivinen langan läpikäynti, jossa hyödynnetään map- ja reduce-funktioita.

Koodiesimerkin 54 funktio convertThreadToArray käy läpi jokaisen puun solmun ja muodostaa niistä listan. Ratkaisu noudattaa myös funktionaalista muuttumattomuutta, eikä sillä ole sivuvaikutuksia. Funktio ei muuta alkuperäistä dataa, vaan se luo uuden listan ja palauttaa sen.

Vastaavaa ratkaisua on hankalampi tehdä ilman rekursiota. Rekursiossa rakenteen syvyyttä ei tarvitse tietää etukäteen.

```
const convertThreadToArray = comment => {
  const comments = []
  const stack = []
  stack.push(comment)

  while (stack.length > 0) {
    const currentComment = stack.shift()
    let comment = {
      id: currentComment.id,
      content: currentComment.content,
      parent_id: currentComment.parent_id ? currentComment.parent_id : -1
    }
```

```
    comments.push(comment)

    if (currentComment && currentComment.children) {
      for (i of currentComment.children) {
        let childComment = {
          id: i.id,
          content: i.content,
          children: i.children,
          parent_id: currentComment.id
        };
        stack.unshift(childComment)
      }
    }
  }
  return comments
};

const comments = convertThreadToArray(thread)
```

Esimerkkikoodi 55. Sama tilanne ratkaistuna ilman rekursiota

Esimerkissä 55 epärekursiivisessa esimerkissä pinoa ylläpidetään manuaalisesti taulukossa. Lapset laitetaan listaan, jotta ne voidaan käydä läpi myöhemmin. Listasta otetaan aina käsittelyyn yksi olio kerrallaan, jolloin poistetaan stack-taulukosta, käsitellään ja siirretään comments-taulukkoon, joka lopulta palautetaan, kun jokainen olio on käyty läpi. Ratkaisu on epäpuhdas, sillä se muuttaa dataa funktion ajon aikana. Ratkaisu saattaa näyttää selkeämmältä ohjelmoijalle, joka ei ole tutustunut funktionaaliseen ohjelmointiin, mutta ratkaisussa on enemmän manuaalista ylläpitotyötä. Siinä pidetään yllä stack-taulukkoa sekä kasataan tuloksia toiseen taulukkoon. Koodin rivimäärältä ratkaisut eivät poikkea juuri toisistaan.

7 Funktioiden kompositio ja ketjutus

Funktioiden yhdistelyä kompositiolla ja ketjutuksella hyödynnetään paljon funktionaalisessa ohjelmoinnissa. Yksinkertaisia funktioita voidaan suorittaa peräkkäin yksi toisensa jälkeen luoden monimutkaisempia toimintokokonaisuuksia. Edellisen funktion paluuarvo annetaan seuraavalle funktiolle parametrina.

7.1 Ketjutus (Chaining)

JavaScriptissä voidaan ketjuttaa metodikutsuja. JavaScriptissä metodeilla tarkoitetaan olioiden tarjoamia funktioita, jotka on määritelty niiden ominaisuuksiksi. Esimerkiksi Arrayn toString-funktio on Arrayn metodi. Metodit eivät useinkaan saa parametreja vaan suorittavat jonkin tietyn toiminnon kyseisen olioinstanssin datalle ja palauttavat sen. Tätä tekniikkaa kutsutaan toisinaan myös Build Patterniksi. Ketjutus yksinkertaistaa ohjelmaa, jossa suoritetaan monta peräkkäistä metodikutsua samalle oliolle. Ketjutuksen ehtona on, että ketjutettavat metodit ovat olion metodeja. JavaScriptissä ketjutetaan usein Arrayn metodeja, kuten esimerkiksi map-, filter- ja reduce-metodeja, jotka ovat sisäänrakennettuja JavaScriptissä. Nämä ovat kaikki Arrayn metodeja ja palauttavat uuden Arrayn. Kaikki metodit eivät kuitenkaan palauta uutta oliota, vaan muokkaavat alkuperäistä, jolloin niistä aiheutuu sivuvaikutuksia eivätkä ne ole täten puhtaita. Map-, filter- sekä reduce-metodit ovat kuitenkin puhtaita. [Mantyla 2015: Chapter 2, Method chains; Kereki 2017: Connecting functions – Pipelining and Composition, Pipelining, Chaining and fluent interfaces.]

```
let array = [1, 2, 3, 4]
let array2 = [5, 6]
let array3 = array.concat(array2).map(item => {
  return {
    value: item
  }
})


console.log(array3) // [ { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 },
  { value: 6 } ]
```

Esimerkkikoodi 56. Esimerkki metodien ketjutuksesta

Esimerkkikoodissa 56 ketjutetaan kaksi Arrayn metodia, concat ja map. Concat luo uuden taulukon, joka yhdistää array:n ja array2:sen taulukot. Map-funktio käy koko taulukon [1, 2, 3, 4, 5, 6] läpi, ja muuntaa jokaisen arvon olioksi ja palauttaen uuden taulukon.

Sama ketjutus toimii omatekemissä olioissa. Oliolle on luotava metodeja ja jotta ketjutus toimii, ehtona on, että edellinen metodi palauttaa sen muotoisen paluuarvon, jota seuraava metodi odottaa parametriksi.

```
function Coordinates(x, y) {
  this.x = x
  this.y = y
}

Coordinates.prototype.location = function() {
  return `Location: x=${this.x} y=${this.y}`
}

Coordinates.prototype.reset = function() {
  return new Coordinates(0, 0)
}

Coordinates.prototype.goRight = function() {
  return new Coordinates(this.x, this.y + 10)
}

Coordinates.prototype.goDown = function() {
  return new Coordinates(this.x + 10, this.y)
}

let someLocation = new Coordinates(200, 500)

console.log(
  someLocation
    .reset()
    .goRight()
    .goDown()
    .goRight()
    .location() // Location: x=10 y=20
)
```

Esimerkkikoodi 57. Esimerkki Coordinates-olion funktioiden ketjuttamisesta

Esimerkissä 57 luodaan olio Coordinates, joka pitää sisällään x- ja y-sijainnit. Sille luodaan neljä uutta funktiota location, reset, goRight ja goDown. Location palauttaa stringinä koordinaattiarvot, muut funktiot palauttavat uuden Coordinates-instanssin, jonka arvoja on muutettu. Reset-, goDown- ja goRight-funktiot ovat kaikki ketjutettavissa missä järjestyksessä tahansa, sillä ne palauttavat Coordinates-instanssin. Sen sijaan Location-metodi palauttaa Stringin, joten sitä ei voida enää ketjuttaa näillä muilla funktiolla.

7.2 Kompositio (Composition) ja putkitus (pipelining)

Kompositio ja putkitus auttavat yhdistämään funktioita niin, että edellisen funktion tulos annetaan seuraavalle funktiolle. Putkitus tarkoittaa sitä, että funktioita ketjutetaan yhteen niin, että edellisen funktion tulos tulee seuraavan funktion syötteeksi. [Kereki 2017; Connecting Functions – Pipelining and Composition.]

```
// Funktioiden kompositio
function4(function3(function2(function1('some argument'))))
```

Esimerkkikoodi 58. Funktioiden kompositio

Esimerkissä 58 yhdistellään funktioita ajamalla ensin function1, jonka tulos annetaan function2:lle ja jonka tulos annetaan function3:lle. Lopuksi tämä tulos annetaan function4:lle.

JavaScriptissä on myös hiljattain lisätty pipeline-operaattori, joka selkeyttää putkituksen syntaksia. Pipeline-operaattori on |> ja sitä käytetään niin, että lauseke jää operaattorin vasemmalle puolelle ja funktio oikealle puolelle. Tämä operaattori on kuitenkin vielä kokeellinen ominaisuus, eivätkä selaimet tue sitä vielä. [Mozilla. Pipeline operator.]

```
// Putkitus pipeline-operaattorilla
'some argument' |> function1 |> function2 |> function3 |> function4
```

Esimerkkikoodi 59. Funktioiden putkittaminen pipeline-operaattorilla

Pipeline-operaatiossa funktiot suoritetaan vasemmalta oikealle, kun taas edellisessä esimerkissä 59 funktiot suoritettiin oikealta vasemmalle. [Kereki 2017; Connecting Functions – Pipelining and Composition, Pipelining.]

Putkitus voidaan toteuttaa myös omalla apufunktiolla, joka luo uuden funktion, joka yhdistää monta funktiota ajamalla sille parametreina annetut funktiot peräkkäin. Putkitus helpottaa funktioiden yhdistelyä ja lisää myös ohjelman luettavuutta.

```
const string = "Hello world. What a wonderful day."

const negative = string => {
  return string.replace(/wonderful/g, "terrible")
```

```
}

const toUppercase = string => {
  return string.toUpperCase()
}

const shout = string => {
  return string.replace(/([.])/g, "!")
}

console.log(shout(toUppercase(negative(string)))) // HELLO WORLD! WHAT A
TERRIBLE DAY!

const pipe = (...functions) => string =>
  functions.reduce((result, func) => func(result), string);

console.log(pipe(negative, shout, toUppercase)(string)) // HELLO WORLD! WHAT A
TERRIBLE DAY!
```

Esimerkkikoodi 60. Putki-funktion käyttö

Esimerkkikoodissa 60 käsitellään merkkijonoa kolmen eri funktion avulla. Jokainen funktio palauttaa uuden modifioidun merkkijonon. Funktiokutsut voidaan koostaa sisäkkäisillä funktiokutsuilla, kuten esimerkissä näkyy shout(toUppercase(negative(string))). Tämä ratkaisu ei ole kuitenkaan kovin luettava, ja funktioiden lisääminen ja poistaminen koosteesta on hankalaa. Sen vuoksi luodaan uusi pipe-funktio, joka ottaa vastaan n-määrän funktioita ja muodostaa niistä uuden funktion, joka ottaa varsinaisen muunnettavan arvon parametriksi. Tämä ratkaisu on helposti luettava ja funktioiden lisääminen, poistaminen ja suoritusjärjestyksen muuttaminen on helppoa.

7.3 Funktori

Funktori on suunnittelumalli, joka pohjautuu matematiikan kategoriateoriaan. Kategoriateoria koostuu matemaattisista rakenteista sekä näiden välisistä suhteista. Funktori on kategoriateoriassa kuvaus (map) kategorioiden välillä. Ohjelmoinnissa funktori on säiliö, joka säilöö jotain rakenteellista dataa sisälleen. Se tarjoaa map-metodin, joka mahdollistaa funktioiden soveltamisen datalle niin, että arvoista lasketaan uutta dataa pitäen datan rakenteen samana. Funktorille annettujen funktioiden ei tarvitse tietää datan rakennetta, koska funktori pitää huolen siitä, että se soveltaa saamansa funktion säilömälleen datalle halutulla tavalla. Yksi entuudestaan tuttu funktori on Array, joka tarjoaa map-metodin. Arrayn map saa parametrina funktion, jonka se soveltaa

jokaiseen sen käärimään arvoon, ja palauttaa uuden taulukon, joka säiliöi uudet arvot. [Fantasyland: functor; Kereki 2017: Building Better Containers – Functional Data Types. Containers. Containers and functors. Enhancing our container: functors; Mantyla 2015: 5. Category Theory. Category Theory. Category Theory in a nutshell & Type safety.]

7.3.1 Toteutus

Funktorin toteutus on hyvin yksinkertainen. Funktori on olio, joka pitää sisällään arvon ja tarjoaa map-metodin, joka soveltaa parametriksi saamansa funktion tälle arvolle. Map-metodi palauttaa arvon uudessa funktori-säiliössä. [Fantasyland. Functor.]

```
class Functor {
  constructor(value) {
    this.value = value
  }

  map(fn) {
    return new Functor(fn(this.value))
  }
}
```

Esimerkkikoodi 61. Funktorin toteutus

Esimerkissä 61 on yksinkertainen funktorin toteutus, joka säilyttää konstruktorissa saamansa datan. Se tarjoaa map-metodin, joka saa parametrina funktion. Funktori antaa säilömänsä datan parametrina saadulle funktiolle ja käärii datan taas uuteen funktoriin, jonka se palauttaa. Alkuperäistä säiliössä olevaa dataa ei muuteta, vaan sen pohjalta luodaan uusi funktori.

7.3.2 Esimerkki

Seuraavassa esimerkissä esitetään kuinka funktoria käytetään. Funktioita suoritetaan ketjussa funktorin säilömälle arvolle map-metodin avulla.

```
class Functor {
  constructor(value) {
    this.value = value
  }

  map(fn) {
    return new Functor(fn(this.value))
```

```
  }
}

const functor = new Functor(3.45)

const double = x => x * 2
const round = x => Math.round(x)
const getResultMessage = x => `The result is ${x}.`

const result = functor.map(double).map(round).map(getResultMessage)

console.log(result) // Functor { value: 'The result is 7.' }
```

Esimerkkikoodi 62. Funktori, joka säilöö jonkin arvon sisäänsä ja suorittaa mapilla saatuja funktioita tälle arvolle

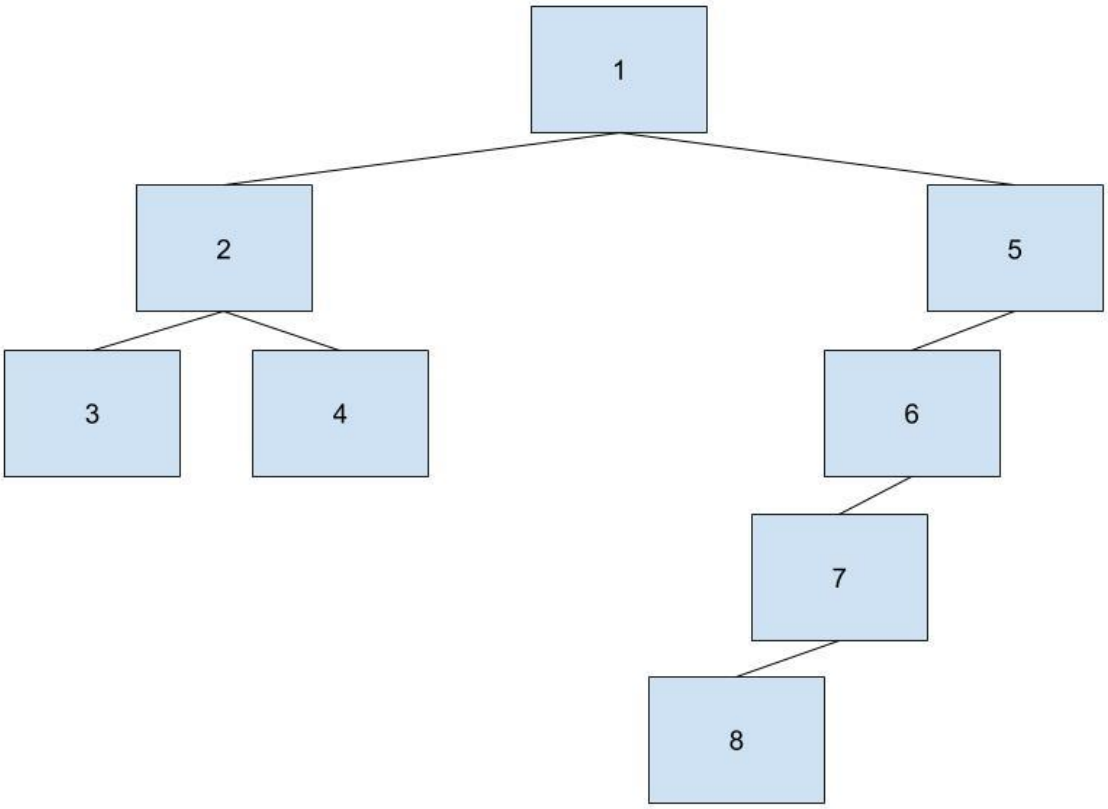
Esimerkissä 62 luodaan uusi funktori arvolla 3.45. Tälle funktorille ajetaan kolme funktiota peräkkäin hyödyntäen funktorin tarjoamaa map-metodia. Ensin funktorin säilömä arvo kerrotaan kahdella ja palautetaan tulos uudessa säiliössä. Sitten tälle uudelle palautetulle funktorille annetaan map-metodissa round-funktio, joka pyöristää arvon. Tämä uusi arvo kääritään taas uuteen funktoriin ja palautetaan. Sen jälkeen map-metodille annetaan getResultMessage-funktio, joka muodostaa funktorin säilömästä arvosta lauseen ”The result is 7.”. Funktorin käärimä tulos on sama kuin, jos funktiot oltaisiin koostettu suoraan arvolle syntaksilla getResultMessage(round(double(3.45))). Ehkä mielenkiintoisempi esimerkki on rekursiivinen funktori, joka esitetään seuraavassa esimerkkikoodissa.

```
class Functor {
  constructor(data, left, right) {
    this.data = data
    this.left = left
    this.right = right
  }

  map(fn) {
    const newData = fn(this.data)
    const newLeft = this.left ? this.left.map(fn) : undefined
    const newRight = this.right ? this.right.map(fn) : undefined
    return new Functor(newData, newLeft, newRight)
  }
}

const tree = new Functor(
  1,
  new Functor(2, new Functor(3), new Functor(4)),
  new Functor(5, new Functor(6, new Functor(7, new Functor(8))))
)

// Impure function...
const print = x => {
```

```
  console.log(x)
  return x
}

tree
  .map(x => x * 10)
  .map(x => `The value of this node is ${x}.`)
  .map(x => x.toUpperCase())
  .map(print)
```

Esimerkkikoodi 63. Rekursiivinen funktori, joka säilöö puurakenteen.

Esimerkin 63 funktori pitää sisällään puurakenteen, jossa jokaisella solmulla on data-, left- ja right-arvot. Data on solmun sisältämä arvo, kun taas left pitää sisällään mahdollisen vasemman puoleisen lapsisolmun ja right taas oikean puoleisen lapsisolmun. Funktori saa konstruktorissa data-, left- ja right-arvot. Left ja right voivat olla undefined tai sitten ne voivat olla uusia Funtor-luokan ilmentymiä.



Kuva 12. Puurakenne, jossa jokaiselle solmulla on arvo ja jokaisella solmulla on mahdollista olla yksi vasemmanpuoleinen lapsi ja yksi oikeanpuoleinen lapsi.

Ohjelmassa luodaan puu, joka vastaa kuvan 12 rakennetta. Koska puurakenne on rekursiivinen, on sen map-metodi toteutettava myös rekursiiviseksi. Mapille annettu funktio on suoritettava kaikille puun solmuille. Map tarkistaa erikseen molemmat left- ja right-lapset ja antaa saamansa funktion eteenpäin lapsen map-metodille, mikäli lapsi on olemassa.

Kun map-metodi pitää huolen siitä, että funktio suoritetaan kaikille solmuille, ei mapille annettujen funktioiden tarvitse välittää puun rakenteesta. Ne voivat olla hyvin yksinkertaisia funktioita, jotka sopivat moneen tilanteeseen ja suorittavat vain jonkin operaation.

Ohjelman lopussa käytetään epäpuhdasta print-funktiota, jotta solmujen arvot voidaan tarkistaa. Funktio tulostaa konsoliin jokaisen solmun silloisen data-arvon ja palauttaa funktorin alkuperäisessä muodossa.

```
THE VALUE OF THIS NODE IS 10.
THE VALUE OF THIS NODE IS 20.
THE VALUE OF THIS NODE IS 30.
THE VALUE OF THIS NODE IS 40.
THE VALUE OF THIS NODE IS 50.
THE VALUE OF THIS NODE IS 60.
THE VALUE OF THIS NODE IS 70.
THE VALUE OF THIS NODE IS 80.
```

Esimerkkikoodi 64. Esimerkkikoodin 63 konsoliin tulostamat tulokset.

Esimerkissä 64 näkyy koodin 63 print-funktion tulostamat arvot. Jokaisen solmun arvo on kerrottu kymmenellä, sitten solmun arvon perusteella on muodostettu lause ”The value of this node is ${x}.” ja lopuksi kaikki lauseen kirjaimet on muutettu suuriksi kirjaimiksi.

Monimutkaisemmassa rakenteessa, kuten esimerkin 63 puurakenteessa, funktori helpottaa funktioiden ketjuttamista huomattavasti. Funktioita ei tarvitse räätälöidä tietyn rakenteen mukaan, vaan funktori pitää huolen siitä, että mapilla saatu funktio suoritetaan halutulla tavalla sen säilömälle datalle.

7.4 Monadi

Monadi on funktorista pidemmälle viety versio. Monadin toteutuksia on monenlaisia eri tilanteisiin. Niiden avulla voidaan haarauttaa ohjelman logiikkaa funktiosekvensseissä, mikä mahdollistaa esimerkiksi virhe- tai poikkeustilanteita funktionaalisen käsittelyn katkaisematta funktioketjua imperatiivisten ehtolauseiden vuoksi. [Fantasyland: Apply, Applicative, Chain, Monad; Kereki 2017: Building Better Containers – Functional Data Types. Containers. Monad, Building Better Containers – Functional Data Types. Containers. Monads. Adding operations.]

Monadin on toteutettava funktorin map-metodin lisäksi metodit ap, of ja chain. Niiden avulla monadia voi käyttää paljon monipuolisemmin funktoriin nähden. Staattinen of-metodi mahdollistaa uuden instanssin luomisen kyseisestä monadista. ap-metodi tulee sanasta apply, ja se mahdollistaa toisen monadin säilömän funktion suorittamisen sen monadin arvolle, jolle ap-metodia kutsuttiin. chain-metodilla voidaan suorittaa monadisia funktioita monadin arvolle. Monadinen funktio tarkoittaa funktiota, joka palauttaa monadin paluuarvona. Chain-metodi ei kääri paluuarvoa enää uuteen säiliöön, koska monadisen funktion palauttama paluuarvo on jo säilötty, eikä sitä haluta kääriä kahteen säiliöön. [Fantasyland: Apply, Applicative, Chain, Monad; Kereki 2017: Building Better Containers – Functional Data Types. Containers. Monad, Building Better Containers – Functional Data Types. Containers. Monads. Adding operations.]

7.4.1 Toteutus

Seuraavassa esimerkissä havainnollistetaan monadin toteutus yksinkertaisimmillaan. Monad-luokalla on staattinen of-metodi sekä map-, ap- ja chain-metodit.

```
class Monad {
  constructor(value) {
    this.value = value
  }

  static of(value) {
    return new Monad(value)
  }

  map(fn) {
    return new Monad(fn(this.value))
```

```
  }

  ap(m) {
    return this.map(m.value)
  }

  chain(fn) {
    return fn(this.value)
  }
}
```

Esimerkkikoodi 65. Monadin toteutus

Esimerkissä 65 on toteutettu monadin metodit of, map, ap ja chain. Yhdessä näillä yksinkertaisilla, metodeilla monadia pystyy ketjuttamaan ja monipuolisesti erilaisissa tilanteissa. Sen map-metodille voi antaa normaalin funktion, joka ei kääri paluuarvoa uuteen monadiin vaan map-metodi käärii funktion paluuarvon uuteen monadiin ennen sen palautusta. Sen ap-metodille voidaan antaa parametrina monadi, jonka säiliössä on funktio. Tämän parametrina saadun monadin funktio suoritetaan sen monadin arvolle, jolle ap-metodia kutsuttiin. Monadin chain-metodille voidaan antaa funktio, joka palauttaa itsessään uuden monadin, jolloin paluuarvoa ei haluta kääriä uuteen säiliöön. Sen staattisella of-metodilla voidaan luoda uusia monadeja.

```
const m1 = Monad.of(1)

// chain
const monadicFunction = x => Monad.of(x)

// Monad { value: Monad { value: 1 } }
const res1 = m1.map(monadicFunction)
// Monad { value: 1 }
const res2 = m1.chain(monadicFunction)
```

Esimerkkikoodi 66. Ohjelmassa havainnollistetaan map- ja chain-metodien ero

Esimerkkikoodissa 66 luodaan uusi monadi m1 esimerkkikoodin 65 Monad-toteutuksen pohjalta. Funktio monadicFunction palauttaa parametrina saadun arvon monadiin säilöttynä. Jos tämä funktio annetaan m1-monadille mapissa, lopputuloksena on Monad { value: Monad { value: 1 } }, eli kahteen säiliöön käärityn arvo. Tarkoitus on pitää arvo vain yhdessä säiliössä, joten chain-metodi auttaa tässä tilanteessa. Kun m1-monadin chainille annetaan monadicFunction, saadaan arvo vain yhdessä säiliössä, jonka monadinen funktio palauttaa.

```
const m1 = Monad.of(1)
const m2 = Monad.of(2)

// ap
const sum = (x, y) => x + y
const curriedSum = x => y => x + y
// Monad { value: 3 }
const res1 = m1.map(m2.map(curriedSum).chain(x => x))
// Monad { value: 3 }
const res2 = m1.ap(m2.map(curriedSum))
```

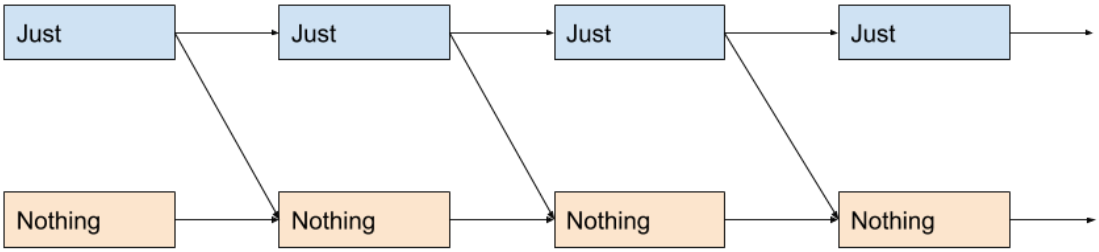
Esimerkkikoodi 67. Ohjelmassa havainnollistetaan ap-metodin käyttöä

Esimerkkikoodissa 67 luodaan kaksi monadia esimerkin 65 Monad-toteutuksen pohjalta. Tässä ohjelmassa halutaan luoda monadi, joka pitää sisällään kahden monadin arvojen summan. Ohjelman sum-funktio täytyy muuttaa yksiparametrisiksi funktioiksi currying-tekniikan avulla, jotta se sopii monadin metodeille ja ketjutukseen, jossa funktion tulos annetaan suoraan seuraavan funktion syötteeksi. Kahden monadin arvojen summa on mahdollista toteuttaa map- ja chain-metodien yhteispelillä, mutta ap-metodi vähentää syntaksia ja helpottaa tällaisen tilanteen toteuttamista. Ensin summa suoritetaan ilman map-metodia ja tallennetaan res1-muuttujaan. Koska m1-monadin map-metodi odottaa funktiota, täytyy m2-monadille antaa ensin curriedSum funktio map-metodiin, jolloin monadin arvoksi syntyy funktio y => 2 + y ja tämä täytyy ajaa chain:in läpi identiteettifunktiolla, jolloin saadaan monadin käärimä arvo ulos monadista. Sitten se voidaan antaa m1-monadin mapille ja saadaan tulokseksi uusi monadi Monad { value: 3 }. Seuraavalla rivillä, jossa tulos tallennetaan res2-muuttujaan, päästään samaan lopputulokseen lyhyemmällä syntaksilla, kun käytetään ap-metodia. Monadi m1:n ap-metodille annetaan parametrina monadi, joka syntyy, kun m2-monadille annetaan map-metodiin curriedSum-funktio. Monadi pitää taas sisällään funktion y => 2 + y. Kun se annetaan ap-metodille, monadi osaa ottaa funktion ulos säiliöstä ja suorittaa sen säilömälleen arvolle.

7.4.2 Esimerkki: Maybe

Maybe on monaditoteutus, jolla voidaan ketjuttaa funktioita, vaikka säilötty data saattaa muuttua jossain vaiheessa ketjua null- tai undefined-arvoksi. Tyypillisesti imperatiivisessa ohjelmassa suoritetaan if-tarkistuksia null- ja undefined-arvojen varalta, jotta vältytään virhetilanteilta. Funktionaalisessa ohjelmoinnissa ei haluta keskeyttää

funktioketjuja tarkistuksien vuoksi, joten null- tai undefined-virhetilanteet voidaan estää Maybe-toteutuksella käyttäen kahta erilaista monadia, usein niitä kutsutaan Just- ja Nothing-monadeiksi. Muut, kuin null- ja undefined-arvot, kääritään Just-monadiin, joka suorittaa saamansa funktiot normaalisti käärimälleen arvolle. Null- ja undefined-arvot kääritään Nothing-monadiin, joka pitää huolen, että sen saamia operaatioita ei yritetä suorittaa arvolle. Se palauttaa vain itsensä ja antaa siten funktioketjun edetä aiheuttamatta virhetilanteita. Koska molemmat monadit tarjoavat samannimiset metodit, voidaan funktioketjua jatkaa, vaikka metodien toteutukset ovat erilaiset Justissa ja Nothingissa.



Kuva 13. Kuvassa esitetään Just- ja Nothing-monadien kulku ketjussa. Just-monadille suoritetut operaatiot saattavat palauttaa Just- tai Nothing-monadin.

Kuvassa 13 havainnollistetaan ketjua, jossa arvo saattaa olla oikea arvo, eli se on säilötty Just-monadiin, tai se saattaa olla null tai undefined, jolloin se on kääritty Nothing-monadiin. Just-monadin säilömälle arvolle suoritetut funktiot ovat monadisia funktioita ja saattavat palauttaa uuden Just-monadin tai palauttaa arvon Nothing-monadissa, jos arvo on null tai undefined. Kun arvo säilötään Nothing-monadiin, ei se voi enää muuttua takaisin Just-monadiksi, vaan se kulkee ketjun läpi Nothing-säiliössä eikä suorita funktioita arvolleen.

```
class Just {
  constructor(value) {
    this.value = value
  }

  static of(value) {
    return new Just(value)
  }

  map(fn) {
    return new Just(fn(this.value))
  }
```

```
  ap(m) {
    return this.map(m.value)
  }

  chain(fn) {
    return fn(this.value)
  }
}

class Nothing {
  constructor() {
    this.value = null
  }

  static of() {
    return new Nothing()
  }

  map(fn) {
    return this
  }

  ap(m) {
    return this
  }

  chain(fn) {
    return this
  }
}

const people = [
  {
    id: 0,
    contact_information: {
      phone: "045 1234 5678",
      address: {
        zip_code: "00100",
        street: "Mannerheimintie",
        house: "1",
      },
    },
  },
  {
    id: 1,
    contact_information: null,
  },
]

const isNothing = x => x === undefined || x === null

const maybe = val => (isNothing(val) ? Nothing.of() : Just.of(val))

const getPersonById = id => {
  return people[id]
}

const getProp = prop => obj => {
  return isNothing(obj) || isNothing(obj[prop])
    ? Nothing.of()
    : Just.of(obj[prop])
```

```
}

// Just { value: '00100' }
const result = maybe(getPersonById(0))
  .chain(getProp("contact_information"))
  .chain(getProp("address"))
  .chain(getProp("zip_code"))

// Nothing { value: null }
const result2 = maybe(getPersonById(1))
  .chain(getProp("contact_information"))
  .chain(getProp("address"))
  .chain(getProp("zip_code"))

// Nothing { value: null }
const result3 = maybe(getPersonById(2))
  .chain(getProp("contact_information"))
  .chain(getProp("address"))
  .chain(getProp("zip_code"))
```

Esimerkkikoodi 68. Just- ja Nothing-monadien toteutus.

Esimerkkikoodissa 68 luodaan kaksi monadia, Just ja Nothing, sekä people-taulukko, jolla on kaksi oliota alkioina. Toisella oliolla on contact_information-arvo ja toisella se on null. Ohjelmassa on maybe-funktio, jonka avulla voidaan palauttaa Just- tai Nothing-instanssi riippuen arvosta. getPersonById-funktiolla haetaan henkilö people-taulukosta. Tässä tapauksessa taulukon indeksi vastaa henkilön id:tä, joten henkilö haetaan suoraan id:tä vastaavalla indeksillä. Henkilö annetaan maybe-funktiolle, jotta siitä saadaan joko Just- tai Nothing-monadi. Ohjelmassa yritetään hakea henkilön henkilötiedoista postinumero, eli zip_code-attribuutin arvo käyttämällä ketjua, joka hyödyntää getProp-funktiota. getProp-funktio palauttaa Just- tai Nothing-monadin, joten funktioiden ketjuttamiseen käytetään chain-metodia, joka ei kääri arvoa enää uuteen monadiin. Henkilö id:llä 0 saa ketjusta tulokseksi Just { value: '00100' }, eli postinumeron, joka on käärityt Just-monadiin. Ohjelman aikana ei tullut null- tai undefined-arvoja, joten ketju pystyi suorittamaan operaatiot alusta loppuun. Henkilö id:llä 1 saa sen sijaan null arvon, kun sen contact_information-arvo haetaan. Tässä vaiheessa getProp-funktio palauttaa Nothing-monadin, ja ketjun kaikki seuraavat operaatiot ohitetaan. Henkilöä id:llä 2 ei ole olemassa, joten maybe-funktio palauttaa jo alkujaan Nothing-monadin, eikä ketjun muita operaatioita enää suoriteta. Tällä tavoin monadeja hyödyntämällä voidaan haarauttaa ohjelman logiikkaa luopumatta funktionaalisesta tavasta yhdistellä funktiokutsuja niin, että edellisen funktion tuloste tulee seuraavan funktion syötteeksi. Ketjun loputtua voidaan tarkistaa, onko tulos Just- vai Nothing-säiliössä ja toimia sen perusteella eri tavoin.

8 Johtopäätökset

JavaScript taipuu hyvin funktionaaliseen ohjelmointiin, sillä sen käyttö ei ole sidottu mihinkään tiettyyn ohjelmointiparadigmaan. Toisaalta paradigman puuttuminen aiheuttaa myös haasteita pysytellä tietyn paradigman periaatteissa.

Muuttumattomuuden toteutus on mahdollista JavaScriptillä. Kieli itsessään ei sitä kuitenkaan vaadi, mikä saattaa tehdä sen toteuttamisesta virhealtista. Jos halutaan taata muuttumattomuus estämällä alkuperäisen datan manipulointi, täytyy data jäädyttää. Sisäkkäiset oliorakenteet täytyy vieläpä jäädyttää rekursiivisesti. JavaScriptille löytyy myös kirjastoja, jotka tarjoavat muuttumattomia tietorakenteita.

JavaScript käsittelee funktioita ensimmäisen luokan kansalaisina eli funktioita voidaan käyttää kuin JavaScriptin primitiiviarvoja. Niitä voidaan nimittää muuttujille, antaa parametreina toisille funktioille ja palauttaa paluuarvoina funktioista. Tämä mahdollistaa yksinkertaisten funktioiden käyttämisen yhdessä luoden monimutkaisempia ohjelmalogiikoita sekä ohjelman osien joustavan hyödyntämisen monissa tilanteissa. JavaScriptin Function-olion bind- ja apply-metodit osoittautuvat myös hyödyllisiksi tilanteissa, jossa luodaan funktioita generoivia apufunktioita, kuten osittainsoveltamis- ja currying-esimerkeissä.

JavaScript tarjoaa jo valmiiksi funktionaaliselle ohjelmoinnille tutut map-, filter- ja reduce-funktiot Array:n metodeina, joten taulukoita on helppo käsitellä funktionaalisesti suoraan JavaScriptin natiivisti tarjoamilla toiminnolla. Yhdessä nämä funktiot tarjoavat erittäin monipuolisen työkalun taulukoiden käsittelyyn ja niiden pohjalta koostettuihin laskutoimituksiin.

Rekursion avulla voidaan käsitellä monimutkaista dataa, kuten syviä puurakenteita, joiden läpikäynti voisi olla lähes mahdotonta perinteisillä for-silmukoilla. Rekursiota toteutettaessa on kuitenkin ymmärrettävä, miten JavaScriptin kutsupino toimii. Rekursio pitää toteuttaa niin, että vältetään pinon täyttyminen ja ohjelman kaatuminen. Trampoliinilla ja thunkilla voidaan estää pinon kasvaminen säilyttäen rekursiivinen ohjelmalogiikka. Eri JavaScript-moottorit saattavat myös käsitellä samaa ohjelmaa

erilailla, kuten häntärekursion optimoinnin yhteydessä, jossa optimoinnin toteutuminen riippuu käytetystä selaimesta.

Funktioita yhdistelemällä ketjuiksi JavaScript-ohjelmasta voidaan tehdä funktionaalista ja deklaratiivista. Ohjelman kulkua voidaan ohjata ketjuttamalla ja koostamalla pieniä funktioita peräkkäin. Funktoreiden avulla voidaan suorittaa funktioita säilötylle datalle säilyttäen datan rakenne ennallaan. Monadit mahdollistavat ohjelman logiikan haarauttamiseen ketjutetuissa funktiokutsuissa, jolloin ketjun kulkua ei tarvitse keskeyttää jokaisen funktiokutsun jälkeen esimerkiksi if-tarkistuksien vuoksi, mikä rikkoisi funktionaalisen deklaratiivisen ohjelmointitavan.

Vaikka täysin funktionaalisen ohjelman tekeminen voi olla mahdotonta, koska jo pelkkä tiedon hakemine API:sta tai käyttäjän syötteen lukeminen tekee ohjelmasta epäpuhtaan, voidaan funktionaalisia periaatteita toteuttaa osana epäpuhdasta ohjelmaa. Puhtaat osat voidaan pitää erillään epäpuhtaista funktioista, jolloin funktionaalisten periaatteiden tuomat hyödyt toteutuvat ainakin osassa ohjelmaa.

Funktionaaliset periaatteet tukevat hyvin yleisesti hyvinä pidettyjä ohjelmointitapoja. Funktionaalinen ohjelmointi on deklaratiivista. Pitkien ohjelmalogiikoiden sijasta ohjelmoidaan lopputulos mielessä. Hyvin nimetyt pienet funktiot, jotka tekevät yhden asian kerralla, sopivat hyvin osaksi funktionaalista ohjelmaa. Kun funktiot ovat puhtaita, ne eivät ole riippuvaisia ympäristöstään, niitä on helppo testata sekä käyttää uudelleen eri tilanteissa, eikä niillä ole sivuvaikutuksia. Kaikki nämä ominaisuudet ovat toivottavia myös muissa ohjelmointiparadigmoissa.

9 Yhteenveto

Työn tavoitteena oli käydä läpi funktionaalisen ohjelmoinnin periaatteet ja selvittää, miten näitä periaatteita voidaan soveltaa JavaScriptissä. Samalla tarkoituksena oli ymmärtää, mitä hyötyä funktionaalisesta ohjelmoinnista on ja syventää osaamista myös JavaScriptiin kielenä.

Työssä selvitettiin funktionaalisen ohjelmoinnin perusperiaatteet käymällä läpi sen teoriaa ja funktionaaliselle ohjelmoinnilla tyypillisiä ominaisuuksia ja toimintoja. JavaScriptiin perehdyttiin kielenä, jotta voitiin ymmärtää miltä osin soveltuu funktionaalisen paradigman toteutukseen. Työssä tehtiin lyhyitä JavaScript-koodiesimerkkejä, joilla havainnollistettiin erilaisten funktionaalisten toimintojen toteutus JavaScriptillä.

Funktionaalinen ohjelmointi perustuu puhtaisiin sivuvaikutuksettomiin funktioihin, funktioiden kohteluun ensimmäisen luokan kansalaisina, ohjelmalogiikan kirjoittamiseen funktiosekvensseinä, jossa edellisen funktion tuloste annetaan seuraavan funktion syötteeksi, rekursioon sekä datan muuttumattomuuteen ja taulukoiden käsittelyyn map-, filter- ja reduce-funktioilla.

JavaScript sopii kielenä funktionaaliseen ohjelmointiin. Sillä pystyy toteuttamaan funktionaalisen ohjelmoinnin periaatteet. Kieli ei kuitenkaan ole kehitetty funktionaaliseksi kieleksi, joten paradigman toteutus vaatii tietoisia päätöksiä ohjelmoijalta. Avuksi voidaan tehdä apufunktioita tai käyttää valmiiksi olemassa olevia kirjastoja.

Funktionaalinen paradigma ohjaa tekemään ohjelmakoodista selkeää ja uudelleenkäytettävää. Puhtaiden funktioiden suosiminen tekee funktioista yksinkertaisia, uudelleenkäytettäviä ja helposti testattavia. Deklaratiivinen ohjelmointitapa tekee ohjelman lukemisesta ja logiikan ymmärtämisestä helpompaa.

Lähteet

Aravinth, Anto ja Machiraju, Srikanth. 2018. Beginning Functional JavaScript. Apress.

Barendsen, Erik ja Barendregt, Henk. 2000. Introduction to Lambda Calculus.

ClojureScript. Verkkoaineisto. <https://clojurescript.org/>. Luettu 1.4.2020.

Elm. Verkkoaineisto. <https://elm-lang.org/>. Luettu 1.4.2020.

Fantasyland. Fantasy Land Specification. <https://github.com/fantasyland/fantasy-land>. Luettu 23.4.2020.

Fogus, Michael. 2013. Functional JavaScript: Introducing Functional Programming with Undercore.js. O'Reilly.

Folktale. Verkkoaineisto. <https://folktale.origamitower.com/>. Luettu 1.4.2020.

Icepick. Verkkoaineisto. <https://github.com/aearly/icepick>. Luettu 1.4.2020.

Immutable: Immutable Collections for JavaScript. Verkkoaineisto. <https://immutable-js.github.io/immutable-js/>. Luettu 1.4.2020.

JavaScript Info. Polyfills. 10.11.2019. Verkkoaineisto. <https://javascript.info/polyfills>. Luettu 2.4.2020.

Kereki, Federiko. 2017. Mastering JavaScript Functional Programming. Mumbai. Packt Publishing.

Mantyla, Dan. 2015. Functional Programming in JavaScript. Packt Publishing.

Mantyla, Dan ja Timms, Simon ja Antani, Ved. 2016. JavaScript: Functional Programming for JavaScript Developers. Mumbai. Packt Publishing.

Martin, Robert C. 2009. Clean code. A Handbook of Agile Software Craftsmanship. United States of America. Pearson Education Inc.

Michaelson, Greg. 2011. Functional Programming Through Lambda Calculus. Mineola. Dover Publications.

Mozilla. Array.prototype.filter(). Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter>. Luettu 3.4.2020.

Mozilla. Array.prototype.map(). Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>. Luettu 3.4.2020.

Mozilla. Array.prototype.reduce(). Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce>. Luettu 3.4.2020.

Mozilla. Callback function. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Glossary/Callback_function>. Luettu 7.4.2020.

Mozilla. Function.length. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/length>. Luettu 3.5.2020.

Mozilla. Function.prototype.apply(). Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply>. Luettu 3.5.2020.

Mozilla. Function.prototype.bind(). Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind>. Luettu 3.5.2020.

Mozilla. Pipeline operator. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Pipeline_operator>. Luettu 3.4.2020.

Ramda. Verkkoaineisto. <https://ramdajs.com/>. Luettu 1.4.2020.

Ramda. Documentation, remove. Verkkoaineisto. <https://ramdajs.com/docs/#remove>. Luettu 2.4.2020.

Reactjs. Docs. Renderng Elements. Verkkoaineisto <https://reactjs.org/docs/rendering-elements.html>. Luettu 2.4.2020.

Reactjs. Docs. Higher order components. Verkkoaineisto. <https://reactjs.org/docs/higher-order-components.html>. Luettu 2.4.2020.

Reason. Verkkoaineisto. <https://reasonml.github.io/>. Luettu 1.4.2020.

Simpson, Kyle. 2017. Functional-light JavaScript: Pragmatic, balanced FP in JavaScript. GetiPub.

Scala.js. Verkkoaineisto. <https://www.scala-js.org/>. Luettu 1.4.2020.