
Tommi Tahvanainen

# Funktionaalinen ohjelmointi web-kehityksessä

Metropolia Ammattikorkeakoulu

Tieto- ja viestintätekniikan tutkinto-ohjelma

Menetelmäopinnot, Pelisovellukset

Tutkielma

16.12.2022

Sisällys

# 1 Johdanto

Tutkielman tarkoituksena oli esitellä funktionaalisen ohjelmoinnin paradigma ja sen käyttöä modernien web-teknologioiden kanssa selvittäen syitä funktionaalisten tekniikoiden käytölle palvelimien rakentamisessa web-alustoille.

Tutkielmassa verrattiin proseduraalisen, oliokeskeisen sekä funktionaalisen ohjelmoinnin eroja eristettyjen testiesimerkkien kautta. Testiesimerkit rakennettiin tutkielmaa varten havainnollistamaan eri tekniikoiden eroja.

Tutkielmassa tarkasteltiin rekrytointitehtävänä rakennettua palvelinta, jossa käytettiin funktionaalisia ohjelmointitekniikoita. Palvelimen tarkoitus oli toimia taustajärjestelmänä selainpohjaiselle sovellukselle, jolla käyttäjä voi lisätä ja poistaa kirjoja henkilökohtaiseen kirjastoonsa.

# 2 Funktionaalisen ohjelmoinnin historia

Funktionaalisten kielten ja ohjelmoinnin edeltäjänä pidetään normaalisti lambdakalkyyliä (Aaby 1998), joka on Turing-täydellinen laskennan malli, eli sillä voidaan ilmaista mitä tahansa matemaattisia laskennan ongelmia. Funktionaaliselle ohjelmoinnille ominaista on sen deklaratiivisuus, eli operaatioita ei järjestetä Turing-laitteelle tai imperatiiviselle ohjelmoinnille tyypillisellä vaihekaavalla, vaan jokainen operaatio voidaan purkaa binääripuuksi. Ongelmaa siis kuvataan tietojenkäsittelytieteen näkökulmasta tietorakenteena.

Tietojenkäsittelytieteistä löytyi jo alkuvaiheissa useampi koulukunta, joista tutkielmalle relevantteina voi pitää lambdakalkyyliä suosivaan kuntaa ja Alan Turingin seuraajia. Turingin hengessä ajattelevien mielestä tietokoneiden ja sittemmin ohjelmien tulisi mallintaa fyysisiä laitteita ja maailmaa lambdakalkyylin sijaan. Tästä maailmaa mallintavasta tavasta tulisi myöhemmin myös tietojenkäsittelytieteen ja ohjelmoinnin kaupallisen kehityksen perusta.

Lambdakalkyylia suosivan koulukunnan työn seurauksena funktionaalinen ohjelmointi on edelleen käytössä tietojenkäsittelytieteen koulukunnassa esimerkiksi ohjelmoinnin ja ohjelmointikielien teorian tutkimuksessa, kuin myös kaupallisessa ohjelmoinnissa.

Ensimmäinen lambdakalkyylin toteuttava ohjelmointikieli on vuonna 1958 alkunsa saanut Lisp (Turner 2012), joka on vuosien varrella muodostunut kokonaiseksi kielikunnaksi. Nykypäivänä aktiivisessa käytössä on monia eri Lisp-dialekteja, joista tunnetuimmat ovat Common Lisp, Emacs Lisp ja Clojure. Lispin sekä muiden funktionaalisten kielien innoittamana funktionaalisen ohjelmoinnin konsepteja, kuten korkean tason funktiot, on myös otettu käyttöön proseduraalisiin ja olio-orientoituneisiin kieliin kuten JavaScriptiin sekä TypeScriptiin.

### 2.1 Lisp ja homoikonisuus

Lisp:n eri dialektit toteuttavat homoikonisuuden, eli dialektit kohtelevat koodia datana ja päinvastoin. Homoikonisuutta kuvaa hyvin koodin rakenne, jonka voi purkaa binääripuuhun, joka on yleinen tietorakenne.

### 2.2 JavaScript omaksuu funktionaalisia tekniikoita

Tutkielmassa tarkasteltavissa esimerkeissä ja rekrytointitehtävässä käytettiin JavaScript-ohjelmointikieltä. JavaScriptin kuvailee ECMAScript standardi, josta on julkaistu monta versiota vuodesta 1997 lähtien. Vaikka JavaScript on pohjimmiltaan proseduraalinen ja imperatiivinen kieli, se on kuitenkin tukenut funktionaalisia tekniikoita sen julkaisusta asti.

## 3 Määritelmät

Funktionaalisessa ohjelmoinnissa on sääntöjä, jotka varmistavat ohjelman toiminnan tavalla, josta funktionaalisen ohjelmoinnin mahdolliset hyödyt saadaan irti. Tutkielmaa varten kiteytettiin säännöt viiteen pääsääntöön;

muuttumattomuus, funktioiden ensiluokkaisuus, tilattomuus, sivuvaikutuksettomuus sekä laskennallinen loogisuus. Lisäksi määriteltiin deklaratiivisuus sekä puhtaat funktiot, jotka ovat myös tärkeitä määritelmiä funktionaalisessa ohjelmoinnissa.

## 3.1 Muuttumattomuus

Muuttumattomuudella tarkoitetaan sitä, että ohjelman sisällä luotuja olioita tai muuttujia tai niiden tilaa ei muuteta luonnin jälkeen (Macdonald 2021). Muuttumattomuuden takia funktionaalisessa ohjelmoinnissa luodaan usein kopioita olioista ja muuttujista, jotta alkuperäinen data säilyy koskemattomana muistissa. Muuttumattomuudella vältetään epäselvyyttä siitä, mikä olion/muuttujan tila tai sisältö on milläkin hetkellä ohjelman kulkiessa, sillä epäselvyys voi johtaa odottamattomiin lopputuloksiin ja vaikeuttaa vian löytämistä.

## 3.2 Ensiluokkaiset funktiot

Ensiluokkaiset funktiot tai korkeamman tason funktiot toimivat funktionaalisen ohjelman perusyksikköinä. Ensiluokkaisuus tulee siitä, että funktioita kohdellaan kuten muuttujia ja olioita; funktioita voi käyttää parametreina toisille funktioille ja niitä voidaan palauttaa arvoina funktio-operaatioista (Katz 2018). Ensiluokkaisten funktioiden käyttöä kuvaa hyvin aiemmin mainittu binääripuu, jossa operaatiot ja arvot jaetaan tietorakenteeseen.

## 3.3 Tilattomuus

Kuten muuttumattomuus, tilattomuus liittyy ohjelman suorituksen aikana tapahtuviin operaatioihin ja niiden kohteina oleviin muuttujiin ja olioihin. Tilattomuudella tarkoitetaan että ohjelman ei tulisi ylläpitää minkäänlaista olio-tilaa suorituksen aikana. Tällöin jokainen funktio operoi eristyksissä muista ohjelman funktioista ja on luotettavampi sekä jokaisen funktion

uudelleenkäytettävyys säilyy, koska ne eivät ole riippuvaisia minkään olion tilasta.

## 3.4 Sivuvaikutuksettomuus

Sivuvaikutuksella tarkoitetaan jotain muutosta sovelluksen tilassa, kuten globaalin muuttujan muutosta, josta funktio ei suoranaisesti riipu eli sitä ei sijoiteta parametrina funktioon (Kereki 2020). Jos funktion tulee muuttaa ulkopuolista tilaa, se tulisi tehdä kopioinnin kautta.

Funktionaalisissa ohjelmissa funktioiden ei tulisi aiheuttaa sivuvaikutuksia. Sivuvaikutuksettomuus useasti muodostuu luonnostaan jos muuttumattomuuden ja tilattomuuden sääntöjä noudatetaan, sillä tyypillisimmät sivuvaikutukset ovat muutoksia funktion ulkopuoliseen tilaan tai muutoksia viittauksella muistiin kun ”muuttuva” olio on asetettu funktion parametriksi.

Syöttö- sekä ulostulo-operaatiot ovat myös sivuvaikutuksia. Täydellistä sivuvaikutuksettomuutta on täten vaikea säilyttää kielissä, joissa ei ole implementoitu jotain erillistä tapaa hoitaa ohjelman vuorovaikutusta ulkoisen maailman kanssa. Haskell on esimerkki ohjelmointikielestä, jossa on implementoitu mekanismi nimeltä ”monadi”, jolla tämänkaltainen vuorovaikutus säilyy sivuvaikutuksettomana.

## 3.5 Laskennallinen loogisuus

Funktionaalisessa ohjelmoinnissa pyritään siihen, että jokainen funktio toimisi laskennallisesti johdonmukaisesti; jos kerroin funktiolle annetaan parametreina kokonaisluku 2 ja muuttuja X tulisi ulostulon olla aina 2*X. Tätä logiikkaa sovelletaan monimutkaisempiin operaatioihin, joissa esimerkiksi iteroidaan isoa tietorakennetta.

## 3.6 Puhtaat funktiot

Kun funktio täyttää sivuvaikutuksettomuuden sekä laskennallisen loogisuuden säännöt, kyseessä on puhdas funktio. Koska voidaan luottaa että funktio palauttaa deterministisen arvon, voidaan olettaa että funktio ei vaikuta sovelluksen muihin osiin tilan kautta. (Jansen 2019)

Puhtaiden funktioiden käyttö ensisijaisesti ohjelman perusyksikköinä tilattomuuden ja muuttumattomuuden kanssa johtaa täysin funktionaaliseen ohjelmaan.

## 3.7 Deklaratiivisuus

Deklaratiivinen ohjelma ilmaisee laskennan logiikkaa kuvailematta sen toiminnan kulkua.

Deklaratiiviselle ohjelmoinnille on tyypillistä, että ohjelman ja aliohjelmien lopputulosta kuvaillaan, kun taas imperatiivisessa ohjelmoinnissa keskitytään ohjelman kulun kuvailuun tai prosessin välivaiheiden määrittelyyn. Imperatiivisen ohjelmoinnin laskennan pääväline on arvojen muuttujiin sijoittaminen, kun taas deklaratiivisessa ohjelmoinnissa päävälineenä toimii argumenttien sijoittaminen funktioihin (Day 2013). Vastakkainasettelua paradigmojen välillä kuvaillaan monesti miten/mitä-asettelulla, jossa imperatiivinen keskittyy siihen miten jotain tehdään ja deklaratiivinen siihen mitä tehdään (Hammad 2021).

Funktionaalinen koodi on deklaratiivista, missä proseduraalinen koodi on imperatiivista. Web-ohjelmointi on useasti sekoitus molempia ohjelmointiparadigmoja.

# 4 Ohjelmakoodiesimerkkejä

## 4.1 Korkeamman tason funktiot

```
const toMax = (max, value) => Math.Max(max, value);
[1,2,3,4].reduce(toMax);
```

Esimerkkikoodi 1. JavaScript ohjelma, jossa on määritelty funktio toMax(), joka palauttaa korkeimman arvon siihen sijoitetuista arvoista (Katz 2018). Ohjelma on esimerkki funktioiden sijoittamisesta muuttujaan, ja kuinka funktioita voidaan käyttää parametreina toisille funktioille, tässä tapauksessa Array.reduce():lle.

## 4.2 Kompositio

```
const composeTwo = (fun1: Function, fun2: Function) => (value: any) =>
fun1(fun2(value));
const square = (value: number) => value * value;
const addEighth = (value: number) => value + (value * 0.125);

const squareAndMargin = composeTwo(addEighth, square);
console.log(squareAndMargin(3)); // 10.125
```

Esimerkkikoodi 2. TypeScript ohjelma, jossa korkeamman tason funktiot square() ja addEighth() muotoillaan yhdeksi funktioksi kompositiolla.

Kompositio mahdollistaa useamman yksinkertaisemman funktion yhdistämisen yhteen kompleksimpaan kuvailevaan funktioon (Jansen 2019). Isommissa koodikannoissa kompositio selkeyttää rakennetta ja vähentää bugeja, kun isoja määriä yksinkertaisia funktioita ketjutetaan.

## 4.3 Korkeamman tason funktioiden ja proseduraalisten toistorakenteiden vertailu

```
function getPrimes(array) {
     let primes = [];
     let flag = true;

     for (number in array) {
          flag = true;
          if (number == 0 || number == 1) flag = false;
          else if (number > 1) {
               for (let i = 2; i <= number / 2; ++i) {
                    if (number % i == 0) {
                         flag = false;
                         break;
                    }
               }
          }
          if (flag == true) primes.push(number);
     }
     console.log(`Prime numbers: ${primes}`);
     return primes;
}
```

Esimerkkikoodi 3. Proseduraalinen JavaScript aliohjelma, joka tulostaa ja palauttaa listan alkulukuja sille parametrina annetusta listasta. Funktio tarkistaa listan luvut proseduraalisesti, hyödyntäen perinteisiä toisto- ja valintarakenteita.

```
const isPrime = n =>
     n <= 1
          ? false
          : !Array.from(new Array(n), (el,i) => i + 1)
               .filter(x => x > 1 && x < n)
               .find(x => n % x === 0);

const array = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17];
const primes = array.filter(isPrime);
```

Esimerkkikoodi 4. Funktionaalinen JavaScript ohjelma, jossa on määritelty aliohjelma isPrime() joka määrittelee onko sille parametrina annettu luku alkuluku. Aliohjelma sijoitetaan Array.filter():lle parametrina, joka käydessään alkuperäistä listaa läpi tarkistaa onko kyseessä alkuluku isPrime():ssa määritellyllä tavalla ja palauttaa alkion uuteen listaan mikäli näin on.

Verrattaessa proseduraalista ja funktionaalista esimerkkiä huomataan proseduraalisen esimerkin sisältävän enemmän sisennystä sekä kontekstirajoja, joka tekee ohjelman lukemisesta sekä ymmärtämisestä

haastavaa. Funktionaalisessa esimerkissä käytetty valintarakenne (eng. ternary operator) sekä korkeamman tason funktioiden hyödyntäminen kuvailee tarkemmin mitä aliohjelma isPrime() tekee. Koodin luettavuus on monesti mielipidekysymys, mutta tässä tapauksessa korrektiuden kannalta deklaratiivisuus tuo selkeyttä koodiin.

Esimerkkikoodien 3. ja 4. välillä ei ole merkittävää eroa suoritusnopeudessa ellei alkioita ole todella suuria määriä. Tapauksissa, joissa tiedetään ettei funktiota käytetä suurien tietorakenteiden iterointiin, voidaan valita kehittäjille mielekkäämpi ja selkeämpi tapa toteuttaa funktio.

## 4.4 Kirjastopalvelin

Kirjastopalvelin kirjoitettiin rekrytointitehtävänä ja ammentaa sekä proseduraalisia että funktionaalisia tekniikoita. Esimerkeistä on hyvä huomata, että tietokannan operaatiot sekä yhteyden avaus ovat turvallisuus- sekä suorituskriittisiä osia mitä tahansa web-sovellusta, joten niiden yksinkertaistaminen tai suoraviivaistaminen mahdollisimman puhtaiksi funktioiksi on tärkeää. Näin vältytään sivuvaikutuksilta, joilla voi olla arvaamattomia seurauksia sovelluksen suoritusaikana tietokantayhteyden ollessa auki.

```
export const dbPromise = async () => open({
     filename: ’books.db’,
     driver: sqlite3.Database
});

const queryRun = async (sql: string, params: SqlParams) => {
     const promise = await dbPromise();
     return await promise.run(sql, params);
};
```

Esimerkkikoodi 5. Osa laajempaa TypeScript-moduulia, jossa aliohjelmat dbPromise() ja queryRun() muodostavat yhteyden tietokantaan ja ajavat esimerkkikoodissa 5. määriteltyä palvelinkoodia tietokantaa vasten. Moduulissa käytettiin ”semi-funktionaalista” koodia, jossa funktiot ovat sivuvaikutuksettomia ja kutsuvat muita funktioita, mutta eivät ole täysin puhtaita funktioita.

```
const create = async (book: SqlParams) => {
     const sql = `insert into
     books(title,author,year,publisher,description) values
     (?,?,?,?,?)`;
     return await db.queryRun(sql, book);
};

const searchAll = async (search: SqlParams) => {
     const sql = buildSearchQuery(search);
     return await db.queryAll(sql, search);
};

export const buildSearchQuery = (search: SqlParams) => {
     let sql = `select * from books where `;
     let first = true;
     if (search[0]) {
          sql = sql.concat(`author = ? `);
          first = false;
     }
     if (search[1]) {
          if (first) {
               sql = sql.concat(`year = ? `);
               first = false;
          }
          else sql = sql.concat(`and year = ?`);
     }
     if (search[2]) {
          if (first)
               sql = sql.concat(`publisher = ? `);
          else sql = sql.concat(`and publisher = ?`);
     }
     return sql;
}
```

Esimerkkikoodi 6. Osa laajempaa TypeScript-moduulia, jossa funktiot create() ja searchAll() muodostavat SQL-lausekkeita funktioiden parametrien perusteella ja sijoittavat SQL-lausekkeet tietokantamoduulin funktioihin. Moduulia varten muodostettiin apufunktio buildSearchQuery(), joka rakentaa SQL-lausekkeen parametreistaan riippuen. Apufunktiossa hyödynnettiin proseduraalista ohjelmointia, sillä merkkijonojen ketjutus parametrien olemassaolosta riippuen osoittautui hankalaksi puhtaasti funktionaalisella ratkaisulla.

## 5 Yhteenveto

Funktionaalinen ohjelmointi ei ole taianomainen ratkaisu web-ohjelmoinnin polttavimpiin kysymyksiin, kuten selain yhteensopivuuteen tai JavaScriptin skaalautuvuuteen kun käyttäjiä on paljon.

Funktionaalisia tekniikoita ja arkkitehtuuria suosimalla voidaan kuitenkin tehdä ohjelmakoodin ylläpitämisestä sekä ymmärtämisestä helpompaa sen kuvauksellisuuden takia. Tutkielmassa käsiteltyjen esimerkkien tapauksessa hyöty näkyisi enimmiltään siinä, että kirjastopalvelinta olisi helppoa laajentaa hyödyntämään pilvipalveluita ohjelmakomponenttien modulaarisuuden ja eriytyvyyden ansioista, jotka ovat funktionaalisten tekniikoiden ansiota. Samalla ohjelmakoodi pysyi selkeänä ja kehittäjän ei tarvitse jatkossa miettiä metafyysisempiä kysymyksiä moduulien olemuksesta laajentaessaan niitä, kuten luokkakeskeisessä olio-ohjelmoinnissa usein käy.

Funktionaaliset tekniikat tekevät ohjelmistokoodista helpommin testattavaa etenkin testiautomaatiokirjastoja käyttäessä. Puhtaiden funktioiden syöttö-ulostulo malli ei vaadi funktion ulkopuolisen tilan replikoimista testattaessa tehden testien kirjoittamisesta oikein helpompaa.

Sekoitus funktionaalisia sekä proseduraalisia tekniikoita on validi tapa kehittää ohjelmia ja yhden ongelman ratkaisu ei välttämättä taivu deklaratiiviseksi koodiksi yhtä helposti kuin toinen.

## Lähteet

Aaby, Anthony A. 1998. Functional Programming. Verkkoaineisto. Walla walla College. <https://www.cs.jhu.edu/~jason/465/readings/lambdacalc.html>. Luettu 28.11.2022.

Day, Laurence. 2013. Programming Paradigms – Computerphile. Verkkoaineisto. <https://www.youtube.com/watch?v=sqV3pL5x8PI>. Luettu 25.11.2022.

Hammad, Madhuri. 2021. Difference Between Imperative and Declarative Programming. Verkkoaineisto. <https://www.geeksforgeeks.org/difference-between-imperative-and-declarative-programming/>. Luettu 28.11.2022

Jansen, Remo H. 2019. Hands-On Functional Programming with TypeScript. O'Reilly.

Katz, Ben. 2018. Practical Functional Programming in Javascript. Verkkoaineisto. <https://www.youtube.com/watch?v=zeZOPB9uxdE&t=1835s>. Luettu 20.11.2022.

Kereki, Federico. 2020. Mastering JavaScript Functional Programming: Write Clean, Robust, and Maintainable Web and Server Code Using Functional JavaScript. O'Reilly.

Macdonald, Millie. 2021. Tiny Programming Principles: Immutability. Verkkoaineisto. <https://www.tiny.cloud/blog/mutable-vs-immutable-javascript/>. Päivitetty 26.1.2021. Luettu 28.11.2022.

Turner, D. A. 2012. Some History of Functional Programming Languages. PDF. <https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf>. University of Kent & Middlesex University. Luettu 10.12.2022