F#-ohjelmointikielellä tehdyn kirjaston kutsuminen

C#-ohjelmointikielellä tehdyssä sovelluksessa

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Visamäki, tietojenkäsittelyn koulutusohjelma

Syksy, 2017

Henri Korpela

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Visamäki

Tekijä Henri Korpela **Vuosi** 2017

Työn nimi F#-ohjelmointikielellä tehdyn kirjaston kutsuminen C#-ohjelmointikielellä tehdyssä sovelluksessa

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Tässä opinnäytetyössä käsitellään funktionaalista ohjelmointia F#-ohjelmointikielen avulla C#-ohjelmoijan näkökulmasta. Työn tavoitteena oli toteuttaa F#-ohjelmointikieltä käyttäen kirjasto .NET-ympäristöön, jota kutsutaan C#-ohjelmointikielellä toteutetussa sovelluksessa. Työn tilaaja oli Sovelluskontti Oy, ja F#-kirjaston toteutus koostuu toimeksiantajalta saadun tulosteen jäsentelyyn tarvittavasta toiminnallisuudesta.

Työ on jäsennelty seuraaviin kokonaisuuksiin: funktionaalinen ohjelmointi ja F#-ohjelmointi sekä F#-ohjelman ajaminen, F#-kirjaston toteuttaminen yleisesti ja tapauskohtaisen F#-kirjaston toteuttaminen tulosteen jäsentelyä varten. F#-ohjelmointikieltä ja sen ominaisuuksia ei käsitellä kattavasti, vaan pääasiassa niin, että tapauskohtaisen tehtävän toteuttaminen onnistuisi.

Johtopäätöksissä tuodaan esille, että C#-koodia toiminnallisuudeltaan vastaava F#-koodi on minimaalisempaa. Mikäli työssä olisi käytetty enemmän hyödyksi F#:n tukemia funktionaalisen paradigman ominaisuuksia, potentiaalisia hyötyjä olisi saavutettu enemmän.

Avainsanat F#, funktionaalinen ohjelmointi, kirjasto

Sivut 50 sivua, joista liitteitä 9 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Visamäki

Author Henri Korpela **Year** 2017

Subject Calling a library implemented in F# programming language in an application implemented in C# programming language

Supervisor Tommi Lahti

ABSTRACT

This thesis discusses functional programming in F# programming language through the viewpoint of a C# programmer. The goal of the work was to implement a library written in F# programming language to .NET environment. This library is called from an application written in C# also implemented in this work. The subscriber of the work was Sovelluskontti Inc. and implementation of the F# library consists of functionality demanded by the client to parse a printout received from the client.

The work is organized to the following sections: functional programming and F# programming together with the execution of an F# application, F# library implementation in general and the implementation of the case by case F# library to parse a printout. F# programming language nor its features are covered in-depth, but rather in a sense that the implementation of the case by case task could be achieved.

In conclusions, it is brought up that C# code equivalent to F# code in functionality is more minimalistic. Had the work used more features of functional paradigm as supported in F#, more potential benefits would have been achieved.

Keywords F#, functional programming, library

Pages 50 pages including appendices 9 pages

SANASTO

F#	Moniparadigmainen mutta ensisijaisesti funktionaalinen ohjelmointikieli, jonka kehitti Microsoft Research.
funktionaalinen ohjelmointi	Ohjelmointiparadigma, jonka keskeisinä konsepteina ovat ensimmäisen luokan funktiot, muuttumattomuus ja sivuvaikutuksettomuus.
Haskell	Puhtaasti funktionaalinen ohjelmointikieli.
imperatiivinen ohjelmointi	Ohjelmointiparadigma, jossa käytetään lausekkeita muuttamaan ohjelman tilaa.
Lisp	(List Processing, vaihtoehtoisesti LISP) on John McCarthyn kehittämä, vuonna 1958 julkaistu moniparadigmainen ohjelmointikieli, joka tukee funktionaalista ohjelmointia.
ML	Funktionaalinen ohjelmointikieli, jonka kanssa F#-ohjelmointikieli piti alun perin syntaksista yhteensopivuutta.
ohjelmointiparadigma	Lähestymistapa ohjelman rakentamiseen.

SISÄLLYS

Liitteet

1 JOHDANTO

Tämä opinnäytetyö perustuu työn asiakkaan ja tilaajan, Sovelluskontti Oy:n, ehdottamaan opinnäytetyöaiheeseen: toteuttaa F#-ohjelmointikieltä käyttäen kirjasto, jota käytetään C#-ohjelmointikielellä toteutetussa sovelluksessa. Aiheen valintaan ovat vaikuttaneet henkilökohtainen kiinnostus opetella F#-ohjelmointikieltä, mutta myös into täyttää asiakkaan tarve F#-ohjelmointikielen potentiaalisten etujen selvittämisessä.

Aiheen voi nähdä ajankohtaisena, koska funktionaalinen ohjelmointi paradigmana on vaikuttanut tekevän uutta tuloaan ja saaneen jalansijaa yritysmaailmassa. F#:n käytön voi taas nähdä modernina tapa ilmentää funktionaalista ohjelmointia .NET-sovelluskehitysympäristössä.

Opinnäytetyön tavoitteena on perehdyttää lukija funktionaaliseen ohjelmointiin F#-ohjelmointikieltä käyttäen. F#-ohjelmointikieli on moniparadigmainen ohjelmointikieli eli se tukee useita erilaisia lähestymistapoja rakentaa ohjelmia. Tästä huolimatta keskiössä tässä työssä on funktionaalinen paradigma, eivätkä muut paradigmat muuta kuin vertailua varten.

Lukijalta edellytetään ymmärrystä ohjelmoinnin perusteista ja olio-ohjelmoinnista. C#-ohjelmointitaito ei ole välttämätöntä, vaikka työssä F#:n ominaisuuksia lähestytäänkin vertaamalla niitä C#-ohjelmointikielen ominaisuuksiin. Mitä tulee työkalujen tuntemiseen, lukijan oletetaan pystyvän käyttämään Visual Studiota lievällä avustuksella.

Keskeisinä käsitteinä opinnäytetyössä ovat funktionaalinen ohjelmointi ja F#.

2 FUNKTIONAALINEN OHJELMOINTI

Petricekin ja Skeetin (2010) kirjoittamassa esittelyssä F#-ohjelmointikieleen ja funktionaaliseen ohjelmointiin todettiin, että täsmällistä määritelmää on vaikea antaa funktionaaliselle ohjelmoinnille. Lukuisista funktionaalisista kielistä ei löydy selkeätä ominaisuuksien joukkoa, joka jokaisessa funktionaalisessa kielessä on oltava.

Määritelmiä funktionaaliselle ohjelmoinnille on kuitenkin laadittu useita. Terkin (2012, 1) mukaan funktionaalinen ohjelmointi on ”ohjelmointiparadigma, jossa funktiot ovat keskeisessä asemassa”. Huttonin (2002) mukaan funktionaalinen ohjelmointi on ohjelmointityyli, joka korostaa ilmausten evaluointia käskyjen suorittamisen sijaan. Evaluoinnilla tarkoitetaan lausekkeiden arvojen laskemista. Ilmaukset näissä kielissä on muotoiltu käyttäen funktioita perusarvon yhdistämiseksi. Funktionaalinen kieli on kieli, joka tukee ja rohkaisee ohjelmointia funktionaalisella tyylillä. (Hutton 2002.)

Hutton (2002.) antaa esimerkin, jossa hän havainnollistaa imperatiivisen C-ohjelmointikielen ja funktionaalisen Haskell-ohjelmointikielen välistä eroa ohjelmalla, joka laskee yhteen luvut yhdestä kymmeneen. Alla olevassa Huttonin tekemässä esimerkissä, joka on toteutettu C-kielellä, toistorakenteen lopputulos kootaan muuttujaan `total` käyttämällä laskurimuuttujaa `i`:

```
total = 0;
for (i=1; <=10; ++i)
        total += i;
```

Funktionaalisessa kielessä sama ohjelma voidaan ilmaista ilman yhtäkään muuttujan päivittämistä. Esimerkiksi Haskellissa tulos voidaan laskea evaluoimalla seuraava ilmaus:

```
sum [1..10]
```

Funktionaalinen ohjelmointi juontaa juurensa lambdakalkyylistä, joka on 1930-luvulla kehitetty matemaattisen laskennan malli (Terkki 2012, 1). Lambdakalkyylissä on kyse abstraktin ohjelmointikielen mallin mallintamisesta, joka perustuu lambdalausekkeisiin. Mallissa näytetään, kuinka ongelmat voidaan ratkaista vain luomalla lambdalausekkeita ja kutsumalla niitä. Verrattaessa tätä Alan Turingin kehittämään Turingin koneeseen, jossa koneen tila muokkautuu annettujen käskyjen mukaan, Turingin koneessa ja lambdakalkyylissä toteutus eroaa, mutta saavutettava lopputulos on sama. (Myyrä 2011, 24.)

Lambdakalkyylillä tarkoitetaan matematiikassa logiikkaan perustavaa funktioiden esitysmuotoa. Esitysmuoto koostuu lambdalausekkeista, jotka kirjoitetaan prefix-muodossa ja joissa funktio ja sen parametrit kirjoitetaan

peräkkäin ilman sulkeita. Yleisesti ohjelmointikielissä ja normaalisti matematiikassa funktion kutsuminen kirjoitetaan esimerkiksi sin(x), mutta lambdakalkyylissä sin x. Funktion sisältäessä useamman parametrin ne lisätään peräkkäin. (Myyrä 2011, 24.) Esimerkiksi seuraava summa

```
5 + 10 + 15
```

esitetään lambdakalkyylissä seuraavassa muodossa:

```
+ 5 10 15
```

Matematiikassa funktiot on tapana esittää yhtälönä, mutta lambdakalkyylissä funktioita ei nimetä lainkaan (Myyrä 2011, 24). Lähestyttäessä funktionaalista paradigmaa F#:ssa tässä työssä lambdalausekkeita käsitellään tarkemmin.

2.1 **Funktionaalisen ohjelmoinnin konsepteja**

Funktionaalisen paradigmaan liittyy muiden paradigmojen tapaan ominaisia piirteitä, toisin sanoen keskeisiä konsepteja. Näihin lukeutuvat funktiot ensimmäisen luokan arvoina, sivuvaikutuksettomuus ja viittauksellinen läpinäkyvyys.

Fancherin (2014, 104) mukaan määrittelevä ja luultavasti tärkein piirre kaikissa funktionaalisissa kielissä on se, että funktioita käsitellään kuin mitä tahansa tietotyyppiä. Terkin (2012, 1-2.) mukaan imperatiivisessa ohjelmoinnissa funktioilla tarkoitetaan yleensä aliohjelmia, kun taas funktionaalisessa ohjelmoinnissa ne tarkoittavat funktiota sanan matemaattisessa merkityksessä. Funktionaalisessa ohjelmoinnissa funktion arvo on jokaisella kutsukerralla sama, kun samoja argumentteja käytetään.

Terkin mukaan ominaista funktionaaliselle ohjelmoinnille on sivuvaikutuksien välttäminen. Sivuvaikutuksella tarkoitetaan tilan muutosta. Imperatiivisessa ohjelmoinnissa ohjelman tila muodostuu muuttujien arvoista tietyllä hetkellä. Tilan muutoksen saa aikaan muuttujan arvon muuttuminen. Sivuvaikutuksien puuttuessa voidaan olla varmoja siitä, että ohjelma toimii aina tietyllä tavalla. (Terkki 2012, 1-2.)

Funktionaaliset ohjelmointikielet voivat sallia imperatiivisen ohjelmoinnin piirteitä tai olla täysin sivuvaikutuksettomia eli puhtaasti funktionaalisia kieliä. Puhtaasti funktionaalisissa kielissä ei ole tilaa. Niissä muuttujaan voi sitoa arvon vain kerran, eikä sitä sen jälkeen voi enää muuttaa. Funktion kutsumisen seurauksena ei voi tapahtua muuta kuin funktion arvon laskeminen. (Terkki 2012, 1-2.)

Sivuvaikutuksettomuuden ideaalista huolimatta Terkki mainitsee funktionaaliseen ohjelmointiin liittyvän kuitenkin tilanteita, joissa sivuvaikutuksilta ei voi välttyä. Tällaisia tilanteita ovat poikkeukset ja syötteen lukeminen. (Terkki 2012, 1-2.)

Viittauksellisella läpinäkyvyydellä (*referential transparency*) tarkoitetaan sitä, että ilmaisu voidaan korvata sen tuloksella muuttamatta ohjelman käyttäytymistä. Esimerkiksi korvattaessa seuraava lauseke, joka on kirjoitettuna F#:lla

```
let sum = add 5 10
```

tällä lausekkeella

```
let sum = 15
```

ilman, että ohjelman käyttäytyminen muuttuu, `add`-funktion sanotaan olevan viittauksellisesti läpinäkyvä. (Fancher 2014, 104.) Alla on esimerkkikoodi C#:lla, jonka kautta voidaan havainnollistaa, miten viittauksellinen läpinäkyvyys saattaa poistaa epäselvyyksiä ohjelman käyttäytymisessä:

```
if(getX() == getX()){
    // Jotain toiminnallisuutta
}
```

Tarkasteltuna ylläolevaa C#-esimerkkikoodia saattaa vaikuttaa siltä, että `if`-lausekkeen ehto täyttyisi eli sen arvoksi tulisi tosi. Näin ei kuitenkaan välttämättä ole, koska ohjelman tila saattaa muuttua joka kerta kutsuttaessa funktiota `getX`, riippuen funktion toteutuksesta. Jos funktion `getX` toteutus olisi osana esimerkiksi seuravaanlaista ohjelmakoodia, vertailuoperaation `getX() == getX()` tulos olisi epätosi:

```
private int x = 0;

public void getX(){
    x++;
    return x;
}
```

Yllä koodiesimerkissä jokaisella funktion getX kutsukerralla muuttujan x arvo lisääntyy yhdellä. Jos siis funktiota getX kutsutaan ensimmäisen kerran, x on arvoltaan 1 ja toisella kutsukerralla se on 2. Kun sitten näitä arvoja verrataan, tulos on epätosi. Tässä tapauksessa `getX` ei olisi viittauksellisesti läpinäkyvä, koska funktioiden `getX` käytön korvaaminen niiden arvoilla muuttaa ohjelman käyttäytymistä, eli x ei muuttuisikaan joka funktion getX kutsukerralla.

Tämänlainen esimerkkikoodi ei välttämättä tuo esille viittauksellisen läpinäkyvyyden varteenotettavia etuja, mutta suurikokoisemmissa ohjelmissa, joissa on satoja luokkia metodeineen, jotka muuttavat olioiden tilaa, viittauksellinen läpinäkyvyys saattaa helpottaa ohjelmakoodin ymmärrettävyyttä ja estää odottamattomia muuttujien arvojen muutoksia.

2.2 **Varhaiset funktionaaliset ohjelmointikielet**

Funktionaalisten ohjelmointikielten historia ulottuu monien vuosikymmenien taakse, oli kyseessä sitten puhtaasti funktionaalinen ohjelmointikieli tai ei. Terkin mukaan lambdakalkyylin ideoihin perustuvia funktionaalisia ohjelmointikieliä ovat Lisp, Haskell ja ML (Terkki 2012, 1).

2.2.1 Lisp

Lisp (historiallisesti LISP, tulee sanoista *List Processing*) kehitettiin tekoälyn käyttöä varten (Milner ym. 1997, Preface, para. 6). Lisp-kielen suunnittelijan, John McCarthyn (1996.) mukaan Lispin toteuttaminen alkoi syksyllä 1958. McCarthy kertoo, että yksi matemaattinen, huomioonotettava asia, joka vaikutti Lispiin, oli tapa ilmaista ohjelmia soveltavina lausekkeina, jotka on rakennettu muuttujista ja vakioista funktioita käyttäen. Hän piti tärkeänä tehdä näistä lausekkeista sellaisia, että ne noudattaisivat yleisiä matemaattisia lakeja, sallien lausekkeiden korvaamisen lausekkeilla antamalla saman arvon. Motiivina oli sallia ohjelman ominaisuuksien todistaminen matemaattisin keinoin. McCarthyn kuvaamassa motiivissa vaikuttaisi olevan kyse viittauksellisesta läpinäkyvyydestä ja siitä, että ohjelmat perustuisivat enemmän matematiikkaan eivätkä ketjuihin käskyjä.

Matemaattisiin lakeihin nojautuvasta motiivistaan huolimatta McCarthy (1996.) sanoo, että tämä todistaminen matemaattisin keinoin on mahdollista vain niin kauan kuin sivuvaikutuksilta voidaan välttyä. McCarthyn mukaan sivuvaikutukset ovat valitettavasti suuri mukavuus, kun laskentatehokkuus on tärkeää. Sivuvaikutukselliset ”funktiot” ovat olemassa Lispissä, mutta niin sanottu puhdas Lisp on sivuvaikutuksetonta. McCarthy antaa ymmärtää sivuvaikutuksettomuus-termin käytössään, että sivuvaikutuksettomuus puhtaassa Lispissä on rinnastettavassa sivuvaikutuksettomuuteen muissa puhtaasti funktionaalisissa ohjelmointikielissä.

Paul Graham (2002.) kiteyttää Lispistä, että tämä 1950-luvun ohjelmointikieli ei ole vanhentunut, koska se ei ollut teknologiaa vaan matematiikkaa, ja matematiikka ei väljähdä. Lisp oli Grahamin mukaan pala teoriaa, joka odottamattomasti muuntui ohjelmointikieleksi.

Grahamin (2002.) mukaan Lisp käsitti yhdeksän uutta ideaa, kun se aluksi kehitettiin. Osaa näistä ideoista pidetään nyt itsestäänselvyyksinä, muita nähdään ainoastaan kehittyneemmissä kielissä ja kaksi näistä ovat uniikkia Lispille:

1. **Ehdollisuudet** Ehdollisuus on if-then-else-rakenne. Näitä pidetään nyt itsestäänselvyyksinä, mutta Fortranissa näitä ei ollut.
2. **Funktiotyyppi** Lispissä funktiot ovat tietotyyppejä, kuten kokonaisluvut ja merkkijonot. Niillä on literaali esitysmuoto, niitä voidaan varastoida muuttujiin, voidaan viedä funktioihin argumentteina ja niin edelleen.
3. **Rekursio** Lisp oli ensimmäinen kieli, joka tuki rekursiota.
4. **Dynaaminen tyypitys** Kaikki muuttujat ovat käytännössä katsoen osoittimia. Arvoilla on tyypit, ei muuttujilla, ja asettamalla tai sitomalla muuttujia tarkoitetaan osoittimien kopioimista eikä sitä, mihin osoittimet osoittavat.
5. **Roskienkeruu**
6. **Ohjelmien koostuminen ilmaisuista** Lisp-ohjelmat ovat puurakenteita ilmaisuista (*expressions*), joista jokainen palauttaa arvon. Lisp eroaa Fortranista ja useimmista sitä seuranneista kielistä, joissa tehdään ero ilmaisujen ja lausekkeiden (*statements*) välille.
7. **Symbolityyppi** Symbolit ovat käytännössä katsoen osoittimia merkkijonoihin, jotka on varastoitu hash-taulukkoon. Näin yhtäsuuruutta voi testata vertaamalla osoittimia sen sijaan, että vertaa jokaista merkkiä erikseen.
8. **Merkintätapa eli notaatio koodiin symboleiden ja vakioista koostuvien puurakenteiden käyttämiseksi**
9. **Koko kieli olemassa koko ajan** Lispissä ei ole todellista erottelua lukuajan, käännösajan ja ajoajan välillä. Koodia voi kääntää ja ajaa lukemisen aikana, lukea tai ajaa kääntämisen aikana ja lukea ja kääntää ajon aikana. Koodin ajaminen lukuaikana antaa käyttäjien uudelleenohjelmoida Lispin syntaksia. Koodin ajaminen käännösaikana on perustana makroille. Kääntäminen ajon aikana on perustana Lispin käytölle laajennuskielenä ohjelmissa kuten Emacs.

Grahamin (2002.) mukaan nämä ideat Lispin ilmaantuessa olivat kaukana sen aikaisesta ohjelmointikäytännöstä, joka määräytyi suuresti 1950-luvun lopulla olemassa olevan laitteiston mukaan.

Nykyään Lispistä on olemassa lukuisia murteita, esimerkiksi Clojure, Common Lisp, Emacs Lisp ja Scheme.

2.2.2 Haskell

Haskell on yleiskäyttöinen ohjelmointikieli, joka tarjoaa korkeamman tason funktiot, staattista polymorfista tyypitystä, käyttäjän määrittämiä algebratyyppejä, hahmontunnistuksen (*pattern matching*), moduulijärjestelmän ja muita ominaisuuksia (Marlow, 2010A). Se on myös puhtaasti funktionaalinen ohjelmointikieli (Fancher 2014, 27; Marlow 2010B; Terkki 2012, 2).

Haskell sai alkunsa vuonna 1987 Portlandissa, Oregonissa pidetyssä konferenssissa nimeltä Functional Programming Languages and Computer Architecture. Marlowin mukaan tapaamisessa oli vahva yhteisymmärrys siitä, että laajamittaisempaa tuon luokan funktionaalisten kielten käyttöä vaikeutti yhteisen kielen puute. Päätettiin, että komitea muodostettaisiin suunnittelemaan tällainen kieli, tarjoten nopeamman uusien ideoiden kommunikoinnin, vakaan perustan todellisten ohjelmien kehittämiselle ja kulkuneuvon, jonka kautta muita rohkaistaisiin käyttämään funktionaalisia kieliä. Haskell-komitean työlle loogisen perustan tarjoaa loogikko Haskell B. Curryn työ. Hänen mukaansa kieli on nimettykin. (Marlow, 2010B.)

Vaikka Haskell on puhtaasti funktionaalinen kieli, siinä on mahdollista varautua tilan sisältävän toiminnan toteuttamiseen. Myyrän (2011, 27.) mukaan Haskell-ohjelmointikielessä ratkaisu syötteen ja tulosteen kanssa toimimiselle on kertoa kääntäjälle, mitkä ohjelmakoodin kohdat sisältävät sivuvaikutuksia. Vastaavasti kääntäjä voi siten tarkistaa, ettei sivuvaikutuksia koodissa ole sisällytetty merkittyjen alueiden ulkopuolelle.

2.2.3 ML

ML-ohjelmointikielellä on ollut vaikutusta F#-kielen kehitykseen. ML-kielen nimi on lyhenne sanoista *meta language* eli metakieli. ML-ohjelmointikielen suunnitteli Rob Milner. (A Functional Programming Language, n.d.) Metakieli on termi, jota loogikot käyttävät kielestä, jossa muista, muodollisista tai epämuodollisista kielistä, keskustellaan ja analysoidaan. Alun perin ML laadittiin välineeksi todisteiden etsimiseksi ja suorittamiseksi loogisessa kielessä. (Milner ym. 1997, Preface, para. 6.)

ML on vahvasti tyypitetty, mutta suurimman osan ajasta ohjelmoijan ei tarvitse kirjoittaa mitään tyyppiesittelyjä. Tämä tekee ML:stä paljon kompaktimman ja luettavamman kuin kielistä, jotka vaativat täsmälliset tyyppiesittelyt (Black 2007). F# seuraa ML-kieltä siinä mielessä, että F# on vahvasti tyypitetty, eikä tyyppiesittelyjä tarvita kuin harvoin.

2.3 **F# ja funktionaalinen ohjelmointi**

F# on ensisijaisesti funktionaalinen, moniparadigmainen ohjelmointikieli (Fancher 2014, 1). F# 4.0 spesifikaatio avaa tätä moniparadigmaisuutta sanomalla, että F# ei ole ainoastaan funktionaalinen vaan myös imperatiivinen ja oliopohjainen ohjelmointikieli (Syme ym. 2016, 11).

Funktionaalisesta puhtaudesta puhuttaessa F# ei Fancherin mukaan ole puhdas funktionaalinen ohjelmointikieli. F# tekee arvot oletuksena muuttumattomaksi. Tämä ei kuitenkaan tarkoita sitä, etteikö muuttujia voisi käyttää perinteisessä mielessä, vaan että arvon muuttamiseksi se täytyy erikseen sallia. (Fancher 2014, 27.)

F#:lla ja ML-ohjelmointikielellä on yhteyttä toisiinsa. Tämä ilmenee siitä, että aikaisin F#:n kehityksessä F#-tiimi yritti kovasti ylläpitää syntaktista yhteensopivuutta ML-kielen kanssa. Ajan myötä F# on kuitenkin poikennut tästä. (Fancher 2014, 1-2.)

3 F#-OHJELMOINTI

Fancherin mukaan F#:iin on kehittynyt ansaitsematon maine kielenä, joka on hyödyllinen ainoastaan akateemisessa ympäristössä tai korkeasti erikoistuneissa talousohjelmistoissa. Tästä johtuen se ei ole onnistunut saamaan laajaa hyväksyntää varsinkaan yritysohjelmistoissa. Tämä näyttäisi kuitenkin muuttuvan kehittäjien alkaessa ymmärtää funktionaalisten kielten hyveitä. (Fancher 2014, 2.)

F# on ensimmäisen luokan kieli monilla alustoilla, mukaan lukien Mac ja Linux, ja työkalutuen avulla Emacsissa, MonoDevelopissa ja Xamarin Studiossa (F# Software Foundation, n.d). Fancherin (2014, xxiv.) mukaan F# tuo funktionaalisen ohjelmoinnin .NET-kehitykseen. Vaikka C# sisältää joitakin funktionaalisia aspekteja, se on ensisijaisesti olio-ohjelmointikieli. C# on kiinnostunut pääasiallisesti käyttäytymisestä ja aina muuttuvasta järjestelmän tilasta. F# taas on ensisijaisesti funktionaalinen kieli, jolla kiinnostus on funktioiden soveltamisessa suhteessa dataan. Tällä erolla on dramaattinen vaikutus, ei ainoastaan koodin kirjoittamiseen, vaan myös siihen, miten koodin kirjoittamista ajatellaan (Fancher 2014, 103). Puhuessaan kielten kiinnostuksen kohteista Fancher viittannee kokonaisuuksiin, joiden avulla ongelmanratkaisua näissä kielissä lähestytään. Fancherin mukaan F#-kehityksessä tulisi suosia funktionaalisia lähestymistapoja, kun mahdollista, ja vaihtaa toisiin tyyleihin, kun tämä on soveliasta (Fancher 2014, 103).

Alun perin F# on kehitetty Microsoft Researchissä, Cambridgessä. Kieli juontaa juurensa vuoteen 2002, mutta ensimmäinen pääjulkaisu, versio 1.0, ilmaantui vuonna 2005. Aikaisin F#:n kehityksessä F#-tiimi yritti kovasti ylläpitää syntaktista yhteensopivuutta ML-kielen kanssa, mutta ajan myötä kieli on hieman poikennut siitä. Microsoft jatkaa vahvasti panostamista F#:iin, mutta kieltä itsessään hallinnoi itsenäinen F# Software Foundation. (Fancher 2014, 1-2.)

3.1 Avainkonseptit ja -rakenteet

F#:n ollessa funktionaalinen ohjelmointikieli se tukee ainakin osittain funktionaalisille ohjelmointikielille keskeisiä konsepteja. F# 4.0-spesifikaation (Sume ym. 2016, 13.) mukaan eräs avainkonsepti F#:ssa on muuttumattomuus (*immutability*). Useimmat asiat, kuten listat ja tuplet, ovat oletuksena muuttumattomia. Fancherin (2014, 26) mukaan haastavin ominaisuus mukautua F#:ssa on muuttumattomuus oletuksena, mikäli F#:iin tullaan pääosin olio-ohjelmointitaustasta. Muuttumattomuus tarkoittaa sitä, että kun arvo on luotu ja sille on annettu nimi, tätä nimeen yhdistettyä arvoa ei voi muuttaa. Muuttumattomuudella on useita etuja. Eräs näistä eduista on se, että muuttumaton data on luonnostaan säieturvallista, joka tekee parallellisoidun koodin prosessoinnin yksinkertaisemmaksi. (Syme ym. 2016, 13.)

Fancherin mukaan ohjelmat, jotka on kirjoitettu kielillä, joissa ei ole muuttumattomuutta oletuksena, voivat olla ennalta-arvaamattomia, koska järjestelmän tila, ohjelmadata, voi muuttua melkein koska tahansa. Näihin muutoksiin Fancher viittaa sivuvaikutuksina. (Fancher 2014, 26). Muuttumattomuudella oletuksena tuetaan siis sivuvaikutuksettomuutta.

3.2 **Ohjelman rakenne**

C#-ohjelmointikielestä poiketen F#:ssa hyödynnetään Python-ohjelmointikielen tapaan sisennystä. Fancherin mukaan F#:ssa kielen suunnittelijat päättivät tehdä välilyönnin merkitykselliseksi sen sijaan, että olisivat nojanneet syntaksisiin merkkeihin koodilohkojen merkitsemiseksi. Lohkon sisällä koodi täytyy sisentää kauemmaksi kuin koodilohkon avaava rivi. Sisennyksen määrällä riveissä ei ole väliä, kunhan ne ovat sisennettynä ja sisennyksen taso on johdonmukainen. (Fancher 2014, 5-6.)

Yksi erottavista tekijöistä F#:ssa on se, että kieli on ilmaisupohjainen. Tämä tarkoittaa sitä, että lähes kaikki, mikä evaluoidaan, palauttaa tuloksen. Tämä on jyrkkä kontrasti esimerkiksi C#:iin, jossa tyypillisesti vain metodit ja operaattorit palauttavat arvon. (Fancher 2014, 8.) Puhuttaessa tässä työssä enemmän F#:n funktioista sekä muista sen tarjoamista rakenteista ilmaisupohjaisuutta selitetään tarkemmin.

3.3 **Sidonnat**

Sidonnat ovat F#:n pääasiallinen tapa yksilöidä ajettavaa koodia tai arvoja. F#.ssa on kolme sidontatyyppiä: do, let ja use. Jokaisella näistä on oma, erityinen tarkoituksensa. (Fancher 2014, 28.)

let-sidonta tehdään käyttämällä avainsanaa let. let-sidonnat ovat kuten readonly-muuttujat C#:ssa. Ne eivät ole vakioita eli literaaleja, koska sidontojen arvot selvitetään ajon aikana, eikä niitä korvata rivikohtaisesti käännösaikana (Fancher 2014, 28.) Alla on esimerkki let-sidonnasta:

```
let age = 18
```

Fancherin mukaan let-sidonnan on mahdollista tehdä myös niin, että arvoa voi muuttaa alustamisen jälkeen. Tämä tapahtuu siten, että sidonnan yhteyteen sisällytetään mutable-avainsana. (Fancher 2014, 29.) Alla on esimerkki, jossa ensimmäisellä rivillä on muutettavissa oleva sidonta ja toisella rivillä uuden arvon asettaminen tähän sidontaan. Sidontaa varten käytetään operaattoria `<-`:

```
let mutable age = 18
age <- 25
```

F#:ssa on myös do-sidonta. Se määritellään käyttäen avainsanaa do. Toisin kuin muut sidontatyypit, do-sidonnat eivät yhdistä arvoja nimeen, vaan niitä käytetään, kun on tarve ajaa jotain koodia funktion tai arvon sidonnan kontekstin ulkopuolella. do-sidontaa käytetään yleisesti toistorakenteissa, sekvenssi-ilmaisuissa, luokkarakentajissa ja moduulialustuksessa. (Fancher 2014, 33.)

use-avainsanalla on muoto, joka muistuttaa let-sidontaa. Se tarjoaa saman toiminnallisuuden kuin let-sidonta, mutta lisää kutsun metodiin Dispose, kun use-avainsanalla sidottu arvo saavuttaa lohkon lopun (*goes out of scope*). (Carter ym. 2016.)

3.4 **Tyypitys ja tietotyypit**

F# Software Foundationin virallisen verkkosivuston mukaan F# on vahvasti tyypitetty ohjelmointikieli (F# Software Foundation, n.d). Koska F# on .NET-ympäristössä tuettu kieli, se tukee täyden valikoiman Common Language Infrastructure eli CLI-tyyppejä. Jokaisella ydinprimitiivillä ja joillakin monimutkaisemmilla tyypeillä, kuten System.String, on tyyppilyhennys, toisin sanoen alias olemassa olevalle tyypille. Näistä monilla on myös ylimääräistä syntaksitukea parantamaan tyyppipäättelyä (*type inference*). Tällä tarkoitetaan kääntäjän kykyä automaattisesti päätellä tietotyyppejä tai yksinkertaistaa niiden kanssa työskentelemistä. (Fancher 2014, 33.)

Alla on esimerkkinä let-sidontoja erityyppisin arvoin. Kommenttiin, joka on samalla rivillä, on merkittynä sidotun arvon tyyppi:

```
let a = ‘a’     // char
let b = 2.5     // float
let c = 50      // int
let d = “Hello” // string
```

Yleisesti jälkiliitteitä käytetään F#:ssa useammin kuin muissa .NET-kielissä, koska ne tarjoavat kääntäjälle kaiken tiedon tyypin päättelemiseksi oikein (Fancher 2014, 35). Alla on havainnollistettuna let-sidonta arvoon, jossa on jälkiliite. Samalla rivillä kommenttiin on merkittynä jälleen sidotun arvon tyyppi:

```
let a = 32u  // uint32
let b = 5.0f // float32
let c = 255I // bigint
```

F# 4.0 -spesifikaation mukaan F# voi tyypillisesti päätellä tyypit, mutta ajoittain täytyy tarjota täsmälliset tyyppiannotaatiot eli tyyppimerkinnät (Sume ym. 2016, 13.) Alla on esimerkkinä let-sidontoja, joissa on tyyppiannotaatiot määrittelemässä täsmälliset tyypit:

```
let a: char = ‘a’
let b: float = 2.5
```

```
let c: int = 50
let d: string = “Hello”
```

F#:ssa on myös unit-tyyppi, jolle ei ole vastaavaa .NET-tyyppiä (Cartel ja Wendel, 2016a). Tyyppi vastaa void-tyyppiä C++- ja C#-ohjelmointikielissä. (Cartel ja Wendel, 2016b.) Jokaisen F#-lausekkeen täytyy palauttaa arvo, ja sellaisissa lausekkeissa, joista ei palauteta arvoa, unit-tyyppi merkitsee tietyn arvon puuttumista. unit-tyypillä on vain yksi arvo, merkittynä ( ). Se toimii silloin, kun mitään muuta arvoa ei ole tai sitä ei tarvita. Alla on sidottu unit-tyyppinen arvo muuttujaan a:

```
let a = ()
```

3.5 **Funktiot**

Fancherin mukaan F#, C#:sta poiketen, ei tee eroa funktioiden, jotka palauttavat arvon, ja funktioiden, jotka eivät palauta arvoa, välillä. Itse asiassa jokainen funktio F#:ssa hyväksyy tasan yhden syötearvon ja palauttaa tasan yhden ulostuloarvon. Tämän toiminnallisuuden mahdollistaa unit-tyyppi. Kun funktio ei sisällä yhtään parametria, funktio todellisuudessa ottaa vastaan unit-tyyppisen arvon. Funktio, joka ei sisällä mitään täsmällistä ulostuloa, palauttaa arvon, joka on myös tyyppiä unit. (Fancher 2014, 104.)

Koska kaikki funktiot ovat ilmauksia, edellytetään, että kaikki funktiot palauttavat arvon. F#-kääntäjä olettaa viimeisen funktiossa olevan lausekkeen olevan funktion paluuarvo. Paluuarvoa ei tarvitse suoraan osoittaa käyttämällä avainsanaa, kuten return. (Fancher 2014, 9.)

C#:ssa on tyypillistä nähdä funktioita, jotka ottavat vastaan useampia parametreja. F#:ssa on näennäisesti funktioita useampien parametrien kera, mutta alla esitellään parametrit F#:ssa ikään kuin ne toimisivat samalla tavalla kuin C#:ssa. Tämä on tehty sen vuoksi, että tässä vaiheessa oleellista on vain käsitellä funktioiden syntaksia, eikä niinkään funktioiden funktionaalista ulottuvuutta F#:ssa.

Alla on näkyvissä funktio nimeltä multiply. Funktio multiply hyväksyy kaksi parametriä: x ja y. Funktiossa on lauseke, jossa kerrotaan nämä parametrit keskenään. Koska laskutoimitus on viimeinen ilmaus funktiossa, palautetaan tämän viimeisen operaation arvo funktiosta:

```
let multiply x y = x * y
```

Ylläolevassa esimerkissä funktion parametreille ei ole annettu tyyppiä. Tyypin voi kuitenkin määrittää samankaltaisesti kuin let-sidonnoissa tyyppiannotaatioin. Myös paluuarvon tyyppi on määriteltävissä. Esimerkkinä alla on funktio concat, joka ottaa parametreikseen muuttajat x ja y, jotka ovat tyyppiä string ja joka palauttaa arvon, tyypiltään string:

```
let concat (x: string) (y: string): string =
    x + y
```

Funktion kutsumisessa C#:sta poiketen ei käytetä sulkeita, vaan parametrit syötetään peräkkäin välilyönnin erottamana ilman pilkkua. Esimerkiksi funktiota concat kutsuttaisiin seuraavasti:

```
concat “Hello, ” “world!”
```

F#:ssa saatetaan törmätä myös let-sidontaan, jonka perässä on käytetty avainsanaa rec. Fancherin (2014, 109) mukaan F#:ssa let-sidotun funktion tekemiseksi rekursiiviseksi täytyy sisällyttää sen määrittelyyn avainsana rec. Rekursiivinen funktio kutsuu itseään joko suoraan tai epäsuoraan toisen funktion kautta. F#:ssa tyypin sisällä olevat metodit ovat implisiittisesti rekursiivisia, mutta ne funktiot, jotka ovat moduulin sisällä, eivät ole tällaisia. Fancher antaa ymmärtää, että ainakin moduulin sisällä olevat rekursiiviset, let-sidotut funktiot vaativat rec-avainsanan käyttöä.

Rekursion voi nähdä osana funktionaalista ohjelmointia ja sitä käsitellään työssä tarkemmin puhuttaessa enemmän funktionaalisesta ohjelmoinnista F#:ssa.

3.6 **Taulukot**

Taulukot F#:ssa ovat rakenteeltaan samanlaisia kuin perinteiset .NET-taulukot. Ne sisältävät kiinteän määrän arvoja, jokainen samaa tyyppiä. Taulukkosidonnat ovat muuttumattomia, mutta yksittäiset taulukon jäsenet ovat muutettavissa, joten täytyy olla varovainen, ettei tuo esiin ei-toivottuja sivuvaikutuksia. (Fancher 2014, 142.)

Fancherin (2014, 142.) mukaan F# tarjoaa lukuisia tapoja uusien taulukoiden luomiseksi. Yksi yleisimmistä tavoista luoda taulukko on taulukkoilmaus. Taulukkoilmaukset koostuvat puolipistein erotetusta listasta arvoja, merkintöjen [| ja |] ympäröimänä. Jos jokainen arvo sijoitetaan omalle rivilleen, puolipisteitä ei tarvita.

Huomionarvoista on merkitä putkimerkki | hakasulkeiden sisään. C#-taustasta ja monista muista ohjelmointikielistä tultaessa putkimerkki saattaa jäädä vanhasta tottumuksesta merkitsemättä. Sama pätee puolipisteen käyttöön arvojen erottajana pilkun sijaan.

Alla on vierekkäin kokonaislukutaulukko values C#:lla sekä samanniminen ja samoin arvoin kokonaislukutaulukko taulukkoilmauksella tehtynä F#:lla:

C#	F#
var values = new int[]{10, 20, 30};	let values = [|10; 20; 30|]

Tyhjän taulukon luomiseksi käytetään merkintöjä [| ja |] ilman sisältöä. Kontekstista riippuen saattaa tarvita tyyppimerkintää varmistamaan, että kääntäjä ei automaattisesti generalisoi taulukkoa. (Fancher 2014, 143.) Ilman tyyppimerkintää tyhjän taulukon voi määritellä seuraavasti:

```
let noValues = [| |]
```

Tyyppimäärittelyn kanssa tyhjän taulukon voi määritellä seuraavasti:

```
let noValues : int array = [| |]
```

Toinen tapa luoda tyhjä taulukko on käyttää funktiota Array.empty. Sitä kutsutaan ilman argumentteja taulukon, pituudeltaan nolla, luomiseksi. Funktion tyyppiparametrin voi ohittaa ja antaa kääntäjän automaattisesti generalisoida se. (Fancher 2014, 143.) Alla on kutsuttu tätä funktiota tyyppiparameterin int kanssa:

```
let noValues = Array.empty<int>
```

Taulukoiden kanssa saattaa olla tarve käsitellä taulukon jäseniä. Fancherin (2014, 144) mukaan yksittäisen taulukon jäsenet ovat avattavissa indeksoidun ominaisuuden (*property*) kautta. Alla on havainnollistettuna taulukon `values` kolmannen jäsenen arvon muuttaminen. C#:sta poiketen hakasulkeiden edessä käytetään pistettä:

```
values.[2] <- 10
```

3.7 Nimiavaruudet

Nimiavaruudet F#:ssa ovat samoja kuin C#:ssa siinä suhteessa, että ne sallivat ryhmittämään koodia nimen perusteella ja vähentämään nimeämiskonfliktin todennäköisyyttä. Nimiavaruudet voivat sisältää moduuleja ja tyyppimäärittelyjä, mutta eivät voi suoraan sisältää mitään arvoja tai funktioita. (Fancher 2014, 6.)

Nimiavaruuksia voi esitellä käyttäen avainsanaa namespace, jonka jälkeen annetaan nimiavaruuden tunniste. (Fancher 2014, 6.) Allaolevassa taulukossa on rinnakkain jo määritellyn System nimiavaruuden käyttö C#:ssa ja F#:ssa:

C#	F#
using System;	open System

Alla on sijoitettuna tyyppi Vehicle nimiavaruuteen Example ja molemmissa, sekä C#:ssa ja F#:ssa, tulostetaan ”Driving” funktiossa Drive:

C#

```
namespace Example{
    public class Vehicle{
        public void Drive(){
            Console.Writeline(“Driving”);
        }
    }
}
```

F#

```
namespace Example

type Vehicle() =
    member this.Drive =
        printfn “Driving”
```

3.8 **Poikkeuskäsittely**

C#:n tapaan F#:ssa on mahdollista varautua poikkeuksiin. F# rakentuu standardeihin .NET-poikkeusmekanismeihin, joihin on lisätty syntaktista tukea ja joilla voidaan F#:n kielenkäytössä nostaa ja käsitellä poikkeuksia (Fancher 2014, 53).

F#:ssa on kaksi rakennetta virhekäsittelyyn: try … finally ja try … with. Nämä rakenteet ovat toisistaan itsenäisiä, joten finally-lohkoa ei voi asettaa try … with -rakenteen yhteyteen, paitsi sisentämällä try … with -lohkon try … finally -lohkon sisään (Fancher 2014, 53.)

try … with -rakenteessa try-lohkon sisällä olevat lausekkeet ajetaan, ja poikkeuksen noston yhteydessä F#:n hahmontunnistusta (*pattern matching*) käytetään sopivan käsittelijän paikantamiseksi with-lohkossa. (Fancher 2014, 53.)

4 F#-OHJELMOINTI JA OLIO-OHJELMOINTI

Vaikka F# on ensisijaisesti funktionaalinen ohjelmointikieli, se tukee myös olio-ohjelmointia, C#:n tapaan muun muassa luokkia ja rajapintoja. Olio-ohjelmointi F#:ssa saattaa olla tarpeellista, esimerkiksi muiden .NET-kielten, kuten C#:n, yhteiskäytön mahdollistajana.

4.1 Luokat

Luokat ovat pääasiallinen tyyppikonsepti, joka tukee olio-ohjelmointia F#:ssa (Cartel, Latham ja Weizel, 2016). Luokat konseptina F#:ssa ovat identtisiä luokkiin muissa olio-ohjelmointikielissä siinä mielessä, että ne enkapsuloivat toisiinsa liittyvää dataa ja käyttäytymistä kenttinä, ominaisuuksina, metodeina ja tapahtumina (joita kutsutaan kollektiivisesti jäseniksi) mallintamaan reaalimaailman konsepteja ja olioita. C#:n tapaan F#:ssa luokat ovat viittaustyyppejä, jotka tukevat yksittäisperintää ja monen rajapinnan toteuttamista. (Fancher 2014, 64.)

F#:ssa esitellään luokka käyttäen avainsanaa type, kuten kaikkien muidenkin käyttäjän määrittelemien tyyppien kanssa. Sen sijaan, että jokainen tietotyyppi vaatisi eri avainsanan, kääntäjä päättelee type-avainsanalla määritellyn tyypin sille määritellyn rakenteen perusteella. (Fancher 2014, 64.) Alla on vertailun vuoksi C#:lla toteutettu luokka ja vastaava luokka toteutettuna F#:lla:

C#

```
class Printout{
    private string name;

    public Printout(string name){
        this.name = name;
    }
}
```

F#

```
type Printout(name: string) =
    member private this.name = name
```

4.2 Rakentajat

Rakentajat eli konstruktorit toimivat jokseenkin eri tavalla F#:ssa kuin muissa .NET-kielissä. F#:ssa luokalla on aina pääasiallinen rakentaja, jonka argumentit kuvaillaan tyypin nimen jälkeen. Rakentajan vartalo koostuu let ja let rec -sidonnoista luokan esittelyn alussa ja näitä seuraavista do-sidonnoista. (Cartel, Latham ja Weizel, 2016.)

F#:ssa pääasiallisen rakentajan lisäksi vaihtoehtoisia rakentajia lisätään käyttämällä new-avainsanaa. Vaihtoehtoisen rakentajan vartalossa täytyy kutsua pääasiallista rakentajaa. Alla on esimerkkikoodi, jossa on luokka Rectangle pääasiallisen ja vaihtoehtoisen, parametrittoman rakentajan kanssa:

```
type Rectangle(width: int, height: int) =
    let h = height
    let w = width

    new() = Rectangle(0, 0)
```

Rectangle-luokan rakentajaa voitaisiin kutsua esimerkiksi seuraavasti:

```
let rect = Rectangle()
```

4.3 **Viittaaminen itseensä**

Itseistunniste (*self identifier*) on nimi, joka esittää nykyistä ilmentymää eli instanssia. Itseistunnisteita muistuttaa this-avainsana C#- ja C++-ohjelmointikielissä. Muista .NET-kielistä poiketen F#:ssa itseistunnisteen voi nimetä kuinka haluaa. Nimiä ei ole rajoitettu esimerkiksi nimiin Me, self tai this. (Cartel, Latham ja Weizel, 2016a.)

F#:ssa voidaan määritellä itseistunniste kahdella tavalla: koko luokan määritelmän lohkossa tai pelkästään yksittäiselle metodille. Itseistunnisteen määritteleminen koko luokalle tapahtuu käyttämällä avainsanaa as. Avainsana sijoitetaan rakentimen parametrilistan jälkeen ja sen jälkeen annetaan tunnisteen nimi. (Cartel, Latham ja Weizel, 2016a.) Alla on esimerkki as-avainsanan käytöstä itseistunnisteen nimeltä self määrittelemisestä koko luokalle:

```
type Rectangle(width: int, height: int) as self =
    // Luokan muu toteutus tässä
```

Itseistunnistetta, joka on määritelty as-avainsanalla, ei ole alustettu ennen kuin let-sidonnat on suoritettu. Näin ollen sitä ei voida käyttää let-sidonnoissa, mutta do-sidonnoissa kyllä. (Cartel, Latham ja Weizel, 2016a.)

4.4 **Kentät ja ominaisuudet**

Kentät määrittelevät dataelementit, jotka ovat yhteydessä olioon. Yksi tapa F#:ssa luoda kenttiä on let-sidonnoin pääasiallisessa rakentajassa. Tällaiset kentät on alustettava pääasiallisessa rakentajassa, ja ovat aina yksityisiä luokalle. (Fancher 2014, 68.) Alla on esimerkki let-sidotusta kentästä address luokassa Location:

```
type Location(addr: string) =
```

```
    let address: string = addr
```

let-sidontojen, jotka vaativat alustuksen pääasiallisessa rakentajassa, sijaan voi käyttää myös täsmällistä kenttää. Kun halutaan enemmän kontrollia kentän suhteen tai pääasiallista rakentajaa ei ole, voidaan täsmällinen kenttä luoda val-avainsanaa käyttäen. Täsmällisiä kenttiä ei tarvitse alustaa välittömästi, mutta sen yhteydessä täytyy olla DefaultValue-attribuutti varmistamassa, että arvo asetetaan sopivaan ”nolla-arvoon”. (Fancher 2014, 68.) Alla on havainnollistettuna täsmällisen kentän address esittely luokassa Location:

```
type Location(addr: string) =
    [<DefaultValue>] val mutable address: string
```

Fancherin (2014, 69.) mukaan ominaisuudet (*properties*) esittävät kenttien tapaan dataa, joka on yhteydessä olioon. Kentistä poiketen ne kuitenkin tarjoavat enemmän kontrollia sen suhteen, kuinka dataa avataan ja muokataan. Datan käsittely tapahtuu get- ja set-funktioiden avulla.

F#:ssa voidaan määritellä ominaisuudet joko implisiittisesti tai ekspliittisesti. Eksplisiittiset ominaisuudet kontrolloivat tyypillisesti taustalla olevaa let-sidontaa ja toteuttavat itse get- ja set-funktioiden vartalot. Eksplisiittinen ominaisuus määritellään käyttämällä avainsanaa member, jota seuraavat itseistunniste, ominaisuuden nimi ja tyyppiannotaatio, mikäli kääntäjä ei pysty sitä päättelemään, sekä funktioiden vartalot. (Fancher 2014, 69.) Allaolevassa koodissa on määritelty ominaisuus Address luokkaan Location. Ominaisuus käyttää datan varastoimiseen let-sidottua merkkijonomuuttujaa address:

```
type Location() =
    let mutable address: string = “”

    member this.Address
        with get() = address
        and set(value) = address <- value
```

Fancherin (2014, 70.) mukaan voidaan käyttää and-avainsanan sijaan vaihtoehtoista syntaksia, jossa get- ja set-avaajat on määritelty erillisinä ominaisuuksina. Alla on esimerkkikoodi, jossa on käytetty tätä vaihtoehtoista syntaksia luokan Location ominaisuuden Address määrittelemisessä:

```
type Location() =
    let mutable address: string = “”

    member this.Address
        with get() = address
    member this.Address
        with set(value) = address <- value
```

4.5 **Perintä**

F#:ssa vain yksi kantaluokka on sallittu, eikä luokan toteuttamia rajapintoja lueta kantaluokiksi. Kantaluokan voi määritellä käyttäen avainsanaa inherit ja avainsanan jälkeen sulkeissa syötetään parametrit kantaluokalle. inherit-lauseke tunnistaa suoran kantaluokan, jos sellainen on. (Cartel, Latham ja Weizel, 2016a.) Alla on määriteltynä esimerkkiluokka Rectangle, joka kutsuu kantaluokkansa Shape parametritonta rakentajaa kantaluokan avulla ja parametreittä periäkseen tuosta luokasta:

```
type Rectangle(width: int, height: int) as self =
    inherit Shape()
```

Kantaluokan metodien ja ominaisuuksien avaamiseksi peritystä luokasta voidaan käyttää avainsanaa base (Cartel, Latham ja Weizel, 2016a), kuten C#:ssa.

4.6 **Rajapinnat**

F# lähestyy rajapintatoteutusta kuin C#, joka sallii epätäsmällisen (*implicit*) ja täsmällisen (*explicit*) toteutuksen. F#:ssa kaikki rajapintatoteutukset ovat täsmällisiä. Kun F#:ssa määritellään tyyppi ilman rakentajia ja jäsenet ovat pelkästään abstrakteja, F#-kääntäjä päättelee tyypin rajapinnaksi. (Fancher 2014, 92-93.) Alla on määritelty esimerkkirajapinta IDrawable abstraktilla metodilla Draw:

```
type IDrawable =
    abstract member Draw : unit -> unit
```

Rajapinnan toteuttaminen on samankaltaista kuin luokan periminen, paitsi että siinä käytetään avainsanaa interface (Fancher 2014, 92-93.) Alla on esimerkki rajapinnan IDrawable toteuttamisesta luokassa Rectangle:

```
type Rectangle() =
    interface IDrawable with
        member x.Draw() =
            printfn “Drawing rectangle”
```

Luokkien tapaan rajapinnat voivat periä toisensa. Ja kuten luokkien perintä, myös rajapintojen perintä saavutetaan käyttämällä avainsanaa inherit (Fancher 2014, 94.) Alla on havainnollistettuna rajapinnan IDrawable perintä rajapinnassa ITransparentDrawable:

```
type ITransparentDrawable =
    inherit IDrawable
    abstract member Transparent: bool
        with get, set
```

Huomionarvoista on, että minkä tahansa tyypin, joka toteuttaa rajapinnan ITransparentDrawable, täytyy toteuttaa sekä IDrawable-rajapinnan jäsenet että ITransparentDrawable-rajapinnan jäsenet.

5 F# JA FUNKTIONAALINEN OHJELMOINTI

Erityisesti F#:n tapauksessa funktionaalinen ohjelmointi eroaa imperatiivisesta ohjelmoinnista ajattelumalliltaan siinä, että se käsittelee funktioita ja niihin liittyviä rakenteita eri tavalla. Osa imperatiiviselle ohjelmoinnille ominaisista piirteistä, kuten ehtolauseet ja toistorakenteet, vaativat eri tapaa lähestyä funktionaalista ohjelmointia puhtaimmillaan.

5.1 Automaattinen funktioiden ketjuttaminen

F#:ssa funktiot toimivat hieman eri tavalla kuin mihin on totuttu ajateltaessa funktion parametreja. F#:ssa jokainen funktio ottaa vastaan yhden parametrin ja palauttaa tasan yhden arvon (Fancher 2014, 106). Alla on funktio multiply, joka ottaa vastaan näennäisesti kaksi parametria

```
let multiply x y = x * y
```

– näennäisesti, koska Fancherin mukaan F#:ssa funktiot eivät toimi näin. Fancherin esimerkkiä mukaillen funktio multiply on sidottu funktioon, joka ottaa parametrikseen kokonaisluvun x ja palauttaa funktion. Palautettu funktio taas vastaanottaa kokonaisluvun y ja palauttaa kokonaisluvun. Fancher kutsuu tätä automaattiseksi funktioiden ketjuttamiseksi (*currying*). Tämä ketjutus on olennaista F#:n käyttämiseksi tehokkaasti, koska se tuo mukanaan lukuisia muita ominaisuuksia, jotka vaikuttavat funktioiden suunnittelemiseen. (Fancher 2014, 106.)

5.2 Lambda-ilmaukset

Lambda-ilmaukset, tai lambdafunktiot, saattavat olla tuttu ominaisuus C#:sta ja Javasta, vaikka kielet ovatkin enemmän imperatiivisia kuin funktionaalisia. Fancherin (2014, 112.) mukaan lambda-ilmauksia on käytetty laajasti funktionaalisessa ohjelmoinnissa. Lyhyesti ne tarjoavat tavan määritellä yksinkertaisia, kertakäyttöisiä, anonyymejä eli nimeämättömiä funktiota. Lambda-ilmauksia on tyypillisesti suosittu let-sidottujen funktioiden sijaan, kun funktio on merkittävä ainoastaan asiayhteydessään, kuten kokoelman filteröinnissä eli suodattamisessa.

F#:ssa lambda-ilmaus alkaa avainsanalla fun, ohittaa funktion nimen ja käyttää nuolimerkintää -> yhtäsuuruusmerkin sijaan (Fancher 2014, 112). Alla on havainnollistettuna lambda-ilmauksen käyttö funktion Array.iter parametrina. Alla oleva tulostaa väliaikaisen taulukon jäsenet 2 ja 4 eri riville:

```
[|2; 4|] |> Array.iter(fun x -> printfn “%d” x)
```

5.3 **Tuplet**

Tuplet lukeutuvat F#:n lisätietotyypeihin. Niitä kutsutaan myös monikoiksi. Tuplet on tyypillisesti yhdistetty funktionaaliseen ohjelmointiin, mutta myös kehitykseen, jossa paradigmat sekoittuvat. Tuplet on tapa ryhmitellä määrä arvoja yhteen rakenteeseen luomatta sille omaa tyyppiä. Tuplet ilmaistaan pilkuin erotettuina listoina, ja ovat joskus ne pm ympäröityinä sulkeisiin. (Fancher 2014, 113.) Tuple saattaa olla tuttu rakenne esimerkiksi Python-ohjelmointikielestä.

Tuple voidaan määritellä, aiemmin esiteltyjen ydinprimitiivien tapaan, ilman tyyppiannotaatiota tai sen kanssa. Alla esimerkkikoodissa on määritelty let-sidonnalla muuttuja vec1 ensimmäisellä rivillä ilman tyyppiannotaatiota ja toisella rivillä muuttuja vec2 tyyppiannotaation kanssa. Fancherin (2014, 113) mukaan tupletyypin annotaatiossa jokaisen arvon tyyppi on erotettuna asteriskimerkillä (*):

```
let vec1 = 2.0, 4.0, 16.0
let vec2 : float * float * float = 2.0, 4.0, 16.0
```

Syntaksisista samankaltaisuuksista huolimatta tuplet, erityisesti sulkeiden ympäröimänä, eivät ole kokoelmia, vaikka sisältävätkin monia arvoja. Tuplet eivät ole iteroitavissa for-toistorakenteessa. Ne ovat usein hyödyllisiä palauttaessa funktiosta monia arvoja tai lähettäessä monia arvoja funktioon ilman, että käytetään automaattista funktioiden ketjuttamista. (Fancher 2014, 113-114.)

F# tarjoaa pääsyn tuplen arvoihin funktioiden fst ja snd avulla. Funktio fst noutaa ensimmäisen arvon tuplesta ja funktio snd noutaa toisen arvon tuplesta. (Fancher 2014, 114.) Allaolevassa esimerkissä on käytetty näitä funktioita tuplen vec kanssa let-sidonnoissa:

```
let vec = 2.0, 4.0, 16.0

let x = fst vec
let y = snd vec
```

Jos yritetään käyttää kumpaakin funktiota triplan eli kolmen arvon sisältävän tuplen kanssa, saadaan aikaan tyyppien yhteensopimattomuus. (Fancher 2014, 114.) Funktiot fst ja snd rajaavat tuplen arvojen käyttämisen kahteen ensimmäiseen arvoon. Tämä lienee riittämätöntä monessa tilanteessa. Suurempi hyöty tuplejen kanssa toimimiseen ilmennee hahmontunnistuksen yhteydessä.

5.4 **Putkitus**

Eräs laajalti F#:ssa käytetty, automaattiseen funktioiden ketjuttamiseen liittyvä ominaisuus on putkitus (*pipelining*). Putkitus sallii omien funktioketjujen luomisen evaluoimalla yhden ilmauksen ja lähettämällä tuloksen toiseen funktioon viimeisenä parametrina. (Fancher 2014, 107.)

Putkituksia on erilaisia: eteenpäin (*forward pipelining*) ja taaksepäin putkitus (*backward pipelining*). Fancherin (2014, 107.) mukaan yleensä arvot lähetetään eteenpäin seuraavaan funktioon käyttämällä eteenpäin putkitus -operaattoria |>. Niin kauan kuin vastaanottavan funktion viimeinen parametri on yhteensopiva lähdefunktion paluutyypin kanssa, monimutkaisia funktioketjuja voidaan luoda.

Alla esimerkissä on taulukko values kokonaislukuineen. Taulukko putkitetaan funktion Array.filter kanssa, ja tämän funktion avulla suodatetaan taulukosta tavoitettavat arvot. Suodatuksen ehtona on, että kohdetaulukon arvot ovat alle 30. Tämän jälkeen putkitetaan taulukko vielä funktion Array.iter kautta, jossa tulostetaan iteroiden jokainen taulukon jäsen suodatuksen jälkeen:

```
let values = [|25; 29; 36; 42|]

values
    |> Array.filter(fun x -> x < 30)
    |> Array.iter(fun x -> printfn “%d” x)
```

Putkitukset voidaan sisällyttää vaihtoehtoisesti samalle riville ilman rivinvaihtoja. Ohjelman ajamisen tuloksena tulostetaan peräkkäin seuraavat arvot:

```
25
29
```

Jos funktion tuloksella ei tahdota tehdä mitään ja funktio palauttaa jonkun muun kuin unit-tyyppisen arvon, tuloksen voi putkittaa ignore-funktioon (Fancher 2014, 107). Alla on käytetty ignore-funktiota taulukon jäsenten summan ohittamiseksi. Summa on laskettu putkittamalla väliaikainen taulukko funktioon `Array.sum`:

```
[| 10; 20; 30 |] |> Array.sum |> ignore
```

Taaksepäin putkitus -operaattori <| toimii samalla tavalla kuin eteenpäin putkitus -operaattori lähettäessään ilmauksen tuloksen toiseen funktioon. Erona on kuitenkin se, että operaattori tekee sen oikealta vasemmalle. Ilmauksen etusijaisuuden muuttamisen vuoksi taaksepäin putkitus -operaattoria on joskus käytetty korvaamaan sulkeet. (Fancher 2014, 108.) Allaolevassa esimerkissä tulostetaan väliaikaisen taulukon summa merkkijonon ”Sum: ” jälkeen.

```
printfn “Sum: %d” <| Array.sum [|10; 20; 30|]
```

Ylläolevan esimerkin voisi toteuttaa myös käyttäen eteenpäin putkitus-operaattoria, jolloin funktiot printfn ja Array.sum olisivat sijoitettuna keskenään toisin päin seuraavalla tavalla:

```
Array.sum [|10; 20; 30|] |> printfn “Sum: %d”
```

Taaksepäin putkitus -operaattorin kanssa tulostamisilmauksen voidaan nähdä olevan luettavammassa muodossa, koska tällöin itse tulostaminen, se, mitä ollaan tekemässä, on korostettuna tulostettavan arvon sijaan.

Fancherin (2014, 108.) mukaan putkitus toimii myös niiden metodien kanssa, jotka ottavat vastaan yhden parametrin. Hän käyttää esimerkkinä TimeSpan-luokan staattista metodia FromSeconds ja Thread-luokan staattista metodia Sleep.

5.5 **Rekursio ja toistorakenteet**

Fancherin (2014, 109.) mukaan on olemassa tyypillisesti kolme toistorakennetta liittyen imperatiiviseen koodiin: for-silmukat, enumeroitavat for-silmukat ja while-silmukat. Koska nämä toistorakenteet nojaavat tilan muutokseen määritelläkseen, milloin poistumiskriteeri täyttyy, puhtaasti funktionaalisessa koodissa täytyy ottaa käyttöön erilainen lähestymistapa. Funktionaalisessa ohjelmoinnissa tällainen lähestymistapa toistorakenteisiin on rekursio.

Alla on Fancherin tarjoama esimerkki rekursiivisesta funktiosta. Kertomafunktio factorial luo ja kutsuu sitten sisennettyä rekursiivista funktiota fact eristääkseen toteutukseen liittyvät yksityiskohdat. Funktio fact ottaa vastaan sekä nykyisen iteraatioarvon c että edellisen iteraation lasketun tuloksen p. Funktio fact suorittaa rekursiivisen kutsun, kun iteraatioarvo ei ole nolla. Lopulta rekursion alustamiseksi funktio factorial kutsuu ensimmäistä fact-iteraatiota, vieden parametreiksi tarjotun arvon v ja arvon 1L (Fancher 2014, 110.):

```
let factorial v =
    let rec c p =
        match c with | 0L -> p
                     | _ -> fact <| c–1L <| c * p
    fact v 1L
```

Match-ilmaus hoitaa tässä esimerkkikoodissa imperatiiviselle ohjelmoinnille tutun `if`-lausekkeen virkaa. Match-ilmaukset liittyvät F#:n tukemaan ominaisuuteen, hahmontunnistukseen (*pattern matching*).

5.6 Hahmontunnistus

Fancherin mukaan yksi voimakkaimmista ominaisuuksista F#:ssa on hahmontunnistus (*pattern matching*) (Fancher 2014, 159). Hahmontunnistukseen F#:ssa lukeutuvat match-ilmaukset.

Match-ilmaukset ovat F#:n pääasiallinen haaroitusmekanismi, vaikka F# sallii imperatiivisen haaroitustyylin if-lausekkeiden kanssa. Pintapuolisesti monet match-ilmaukset muistuttavat C#:n switch-lausekkeita. C#:n switch-lausekkeet operoivat ainoastaan vasten muuttumattomia arvoja. Match-ilmaukset taas valitsevat ilmauksen evaluoitavaksi sen mukaan, mikä hahmo täsmää syötteen kanssa. Match-ilmauksia voidaan käyttää useiden tietotyyppien kanssa kuten numeroiden, merkkijonojen ja tuplejen kanssa (Fancher 2014, 159-160.) Alla on esimerkki match-lausekkeesta, jossa käytetään merkkijonoa str funktiossa CheckColor:

```
let CheckColor str =
    match str with
    | “blue” -> “Blue”
    | “green” -> “Green”
    | “red” -> “Red”
    | _ -> “Invalid color”
```

Koska match-ilmaukset palauttavat myös arvon, jokaisen tuloslausekkeen täytyy olla tyypiltään sama. Hahmot tunnistetaan peräkkäin, joten ne on organisoitava täsmällisemmästä vähiten täsmälliseen. Jos yleisempi hahmo on sijoitettu ennen tarkempia hahmoja, se estää minkä tahansa seuraavan hahmon evaluoinnin. (Fancher 2014, 160.)

Hahmoja F#:ssa on tuettu lukuisia erilaisia. Alla on koottuna taulukkoon hahmontunnistuksessa käytettävän hahmon nimi, kuvaus ja annettu esimerkkejä niiden käytöstä. Taulukkoon on poimittu vain osa F#:ssa tuetuista hahmoista, ei kattavasti kaikkia hahmoja:

Taulukko 1. F#.n tukemia hahmoja hahmontunnistuksessa (Cartel, Latham, Wenzel ja Marcio, 2016).

Nimi	Kuvaus	Esimerkit

Vakiohahmo	Mikä tahansa enumeraatio-vakio, merkki, merkkijono tai numero	`30`
		`1.0`
		`”test”`

AND-hahmo	hahmo1 & hahmo2	`(a, b) & (_, ”test”)`
OR-hahmo	hahmo1 | hahmo2	`([h] | [h; _])`
Taulukkohahmo	[|hahmo_1; …; hahmo_n|]	`[| a; b; c |]`
Tuplehahmo	(hahmo_1, …, hahmo_n)	`(a, b)`
Jokerimerkki-hahmo	_	_

null-hahmo	null	`null`

Vakiohahmot ovat enumeraatio-, numeerisia, merkki- ja merkkijonovakioita. Match-ilmausta, jolla on ainoastaan vakiohahmoja, voidaan verrata case-lausekkeisiin muissa kielissä. Syötettä verrataan literaaliarvoa vasten ja hahmo täsmää, jos arvot ovat yhtäsuuria. (Cartel, Latham, Wenzel ja Marcio, 2016.)

Alla olevassa esimerkkikoodissa on käytetty match-ilmauksessa numeerisia vakiohahmoja. Ensimmäisellä rivillä on sidottu arvo 5 muuttujaan x ja match-ilmauksessa on kolme hahmoa, johon muuttujaa x verrataan: 1, 5 ja jokerimerkki (*wildcard*). Jokerimerkkihahmo on esitetty alaviivan avulla. Fancherin esimerkkien perusteella jokerimerkkihahmo täsmää, mikäli mikään muu hahmo ei täsmää (Fancher 2014, 163-164). Ajettaessa alla oleva koodi muuttujaan y sidotaan merkkijono ”Five”, koska muuttujan x arvo 5 täsmää vakiohahmon 5 kanssa:

```
let x = 5
let y = match x with
        | 1 -> “One”
        | 5 -> “Five”
        | _ -> “Some other value”
```

Yhden hahmon täsmäämisen lisäksi voidaan yhdistellä hahmoja. AND-hahmot, joita joskus yhdistetyiksi hahmoiksi (*conjunctive patterns*) kutsutaan, voidaan täsmätä useampien yhteensopivien hahmojen kanssa et-merkillä (&). Jotta AND-hahmo täsmää, syötteen täytyy täsmätä jokaisen AND-hahmoon kuuluvan hahmon kanssa. Yleisesti AND-hahmot eivät ole niin hyödyllisiä perushahmontunnistusskenaariossa, koska ilmausukykyisemmät guard-ehdot soveltuvat paremmin tähän tehtävään. AND-hahmot ovat silti hyödyllisiä erottamaan arvoja esimerkiksi silloin, kun toinen hahmo AND-hahmossa täsmää. (Fancher 2014, 171-172.)

Jos useamman hahmon tulisi suorittaa sama koodi, voidaan ne yhdistää toisiinsa käyttämällä OR-hahmoa tai, toisella nimityksellä, vaihtoehdollista hahmoa (*disjunctive pattern*). OR-hahmossa yhdistetään useampi hahmo pystypäin olevalla putkimerkillä (|). Monella tapaa OR-hahmot ovat samankaltaisia C#:n läpiputoamistapauksien (*fall-through cases*) kanssa switch-lausekkeissa. (Fancher 2014, 172.)

Fancherin (2014, 166.) mukaan tuplen voi täsmätä ja hajottaa osatekijöihinsä tuplehahmon avulla. Match-lausekkeissa voidaan käyttää lukuisia tuplehahmoja suorittamaan haarauttamisen näiden arvojen perusteella. Fancher antaa tuplehahmoon liittyvän esimerkin, jossa on kaksiulotteinen tuplena esitetty piste. Tämä piste voidaan hajottaa itsenäisiin x- ja y-koordinaatteihin, ja sitä voidaan käyttää määrittelemään, onko se sijoitettuna origoon vai pitkin akselia, kuten alla olevassa esimerkissä on tehty match-ilmauksen avulla:

```
let locatePoint p =
    match p with
    | (0, 0) -> sprintf “%A is at the origin” p
    | (_, 0) -> sprintf “%A is on the x-axis” p
    | (0, _) -> sprintf “%A is on the y-axis” p
    | (x, y) -> sprintf “Point (%i, %i)” x y
```

Ylläolevassa esimerkissä on käytetty jokerimerkkihahmoa sallimaan minkä tahansa arvon sijoittamisen. Mitä tulee tuplen arvojen määrään, Fancher huomauttaa, että niiden täytyy täsmätä tuplehahmon kanssa (Fancher 2014, 167).

6 F#-OHJELMAN AJAMINEN

Oleellisena osana työtä oli F#:n käyttäminen Visual Studio -sovelluskehitysympäristössä. Tässä kappaleessa käydään läpi F#-ohjelman ajaminen ja tarvittavien työkalujen asentaminen, Microsoft Visual Studiota lukuunottamatta.

F# 4.0 spesifikaation mukaan F#-ohjelman ajamiseen on olemassa ainakin kolme erilaista vaihtoehtoa (Syme ym. 2016, 11):

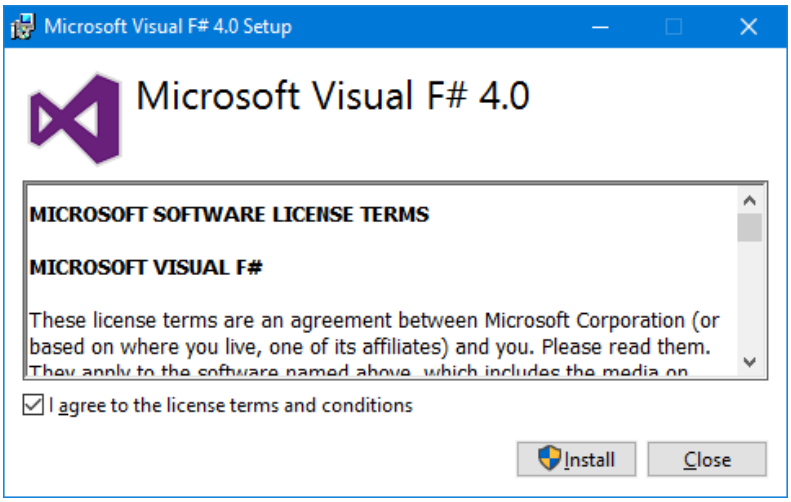
1. Käännetään ohjelma sovelluskehitysympäristössä, kuten Visual Studio
2. Suoritetaan manuaalisesti F#:n fsc.exe komentokehotteen kautta
3. Käytetään dynaamista kääntäjää nimeltä F# Interactive, joka on osa F#-jakoa

Tässä työssä käytetään sovelluskehitysympäristöä Visual Studio, eikä käsitellä näitä muita vaihtoehtoja.

6.1 Asentaminen

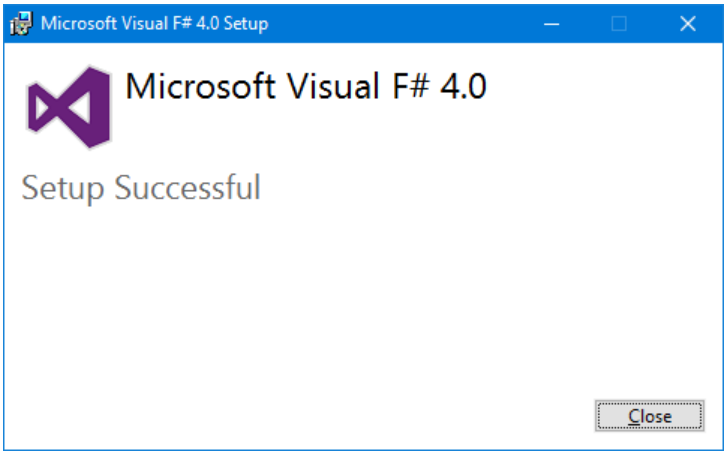
F# Software Foundationin verkkosivuilla kerrotaan, että F# on ajettavissa Androidilla, iOS:illa, Linuxilla, Mac OS X:llä, selaimissa ja GPU:issa (F# Software Foundation, n.d.). Tässä työssä asennus suoritettiin Windowsille. F#:n asentaminen on tehty tässä kappaleessa käyttäen Visual F# -työkaluja Microsoftilta.

Asennusohjelman avauduttua pyydettiin hyväksymään Microsoftin ohjelmiston käyttöehdot. Hyväksyttiin ehdot valintaruudun kautta ja painettiin sen jälkeen painiketta Install:



Kuva 1. Microsoftin ohjelmiston käyttöehdot, jotka on hyväksytty valintaruudun kautta.

Tämän jälkeen asennusohjelma asentaa Microsoft Visual F# 4.0:n työkalut käytettäväksi Visual Studioon. Asennuksen suorittamisen jälkeen ilmaantui seuraavanlainen näkymä, jossa painettiin painiketta Close asennusohjelman sulkemiseksi:

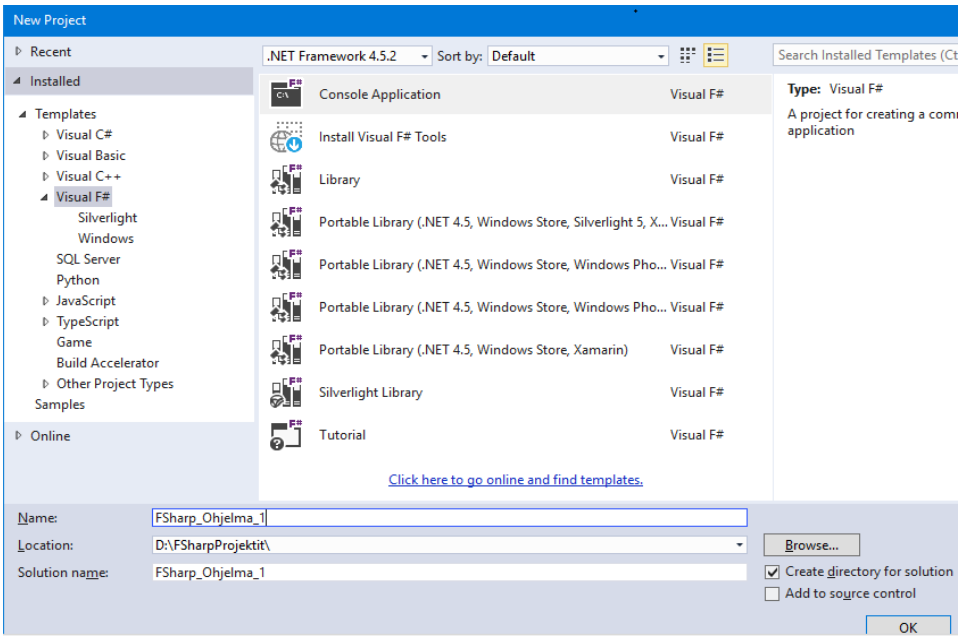


Kuva 2. Microsoft Visual F# 4.0 –asennusohjelma ilmoittamassa, että työkalut on asennettu onnistuneesti.

Asennuksen aikana oli minimaalisesti asetuksia, mistä johtuen asennus koettiin erittäin helpoksi.

6.2 Ympäristö

Microsoft Visual F# 4.0:n työkalujen asennuksen onnistuttua avattiin Visual Studio 2015 Community Edition ja siirryttiin projektimalleissa kohtaan Visual F# uutta projektia luotaessa.



Kuva 3. Visual F# -malliprojektivaihtoehdot, joista valittuna malliprojekti Console Application.

F#-projektimalleissa on useampi vaihtoehto. Yksi niistä on Install Visual F# Tools. Sen kautta on mahdollista asentaa Visual F# -työkalut. Valitettavasti opinnäytetyön suorittamisen aikana Visual Studio 2015 Community Editionissa työkalujen asennus tätä kautta ei toiminut lainkaan, eikä epäonnistuneesta asennusyrityksestä annettu minkäänlaista virheviestiä.

Tästä huolimatta Console Application -projektimallivaihtoehto toimi. Tämän mallin avulla luotiin uusi projekti nimellä FSharp_Ohjelma_1 paikalliseen hakemistoon.

6.3 **Projektin rakenne**

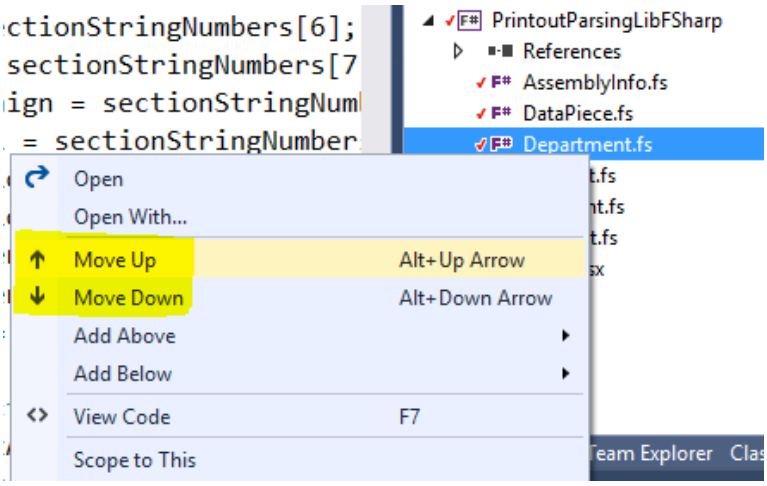
Fancherin (2014, 4.) mukaan F#-projekti on tietyiltä osin aivan kuin C#- tai Visual Basic –projekti. Esimerkiksi suoritettavissa olevat projektit voidaan suorittaa painamalla painiketta F5, Visual Studio –debuggeri voi kulkea läpi F#-koodin ja tiedostoja hallinnoidaan Solution Explorer –näkymän kautta. Projektin organisointi F#:ssa on kuitenkin todella erilainen perinteisiin .NET-kieliin verrattuna.

Perinteiset .NET-projektit noudattavat yleisesti käytäntöä, jossa yhdessä tiedostossa on yksi tyyppi, toisin sanoen yksittäiset tietotyypit ovat melkein aina varastoituina erillisiin tiedostoihin ja organisoituina kansiohierarkiaan, joka peilaa projektin nimiavaruuksia. Vankkoja sääntöjä siihen, miten ja milloin jokin voi ilmaantua projektissa, on harvassa, kehäassemblyviittausta lukuunottamatta. Tyypit ja jäsenet ovat vapaita viittaamaan toisiinsa ja jäseniinsä huolimatta siitä, missä ne ovat määriteltyinä projektissa. (Fancher 2014, 4.)

Fancherin mukaan F# sitä vastoin on äärettömän ohjaileva projektin organisoinnissa, sillä F#-koodi evaluoidaan ylhäältä alaspäin. Tämä tarkoittaa sitä, että esittelyiden järjestys yksittäisessä tiedostossa on merkityksellistä. Myös tiedostojen järjestyksellä projektissa on merkitystä. Kansioita ei ylhäältä alaspäin –evaluointijärjestyksen vuoksi sallita, eikä niitä ohjelmistokehitysympäristön puolelta ole mahdollista lisätä. Pääasiallinen etu tälle evaluontirakenteelle on se, että kääntäjä voi tehdä olettamuksia koodista ja antaa paremmat tyyppipäättelyvalmiudet. Tämä evaluointirakenne välttää tahattomia, rekursiivia määrittelyjä eli kahden tai useamman määrittelyn riippuvuutta toisistaan. (Fancher 2014, 4.)

Fancherin mukaan on yleistä uusille F#-ohjelmoijille lisätä uusi tiedosto F#-projektiin, lisätä tiedostoon määritelmiä ja sitten saada kääntäjävirhe, jossa kerrotaan, että uudet määritelmät puuttuvat. Tämä johtuu yleensä siitä, että ohjelmoija on unohtanut siirtää uuden, luodun tiedoston niiden tiedostojen yläpuolelle, jotka käyttävät näitä määritelmiä. Visual Studiossa on kontekstivalikon jäseniä ja pikanäppäimiä, joilla tiedostoja voidaan siirtää ylös ja alas. (Fancher 2014, 4.)

Alla on kuvankaappaus F#-projektista Visual Studiossa, jossa valitun tiedoston kontekstivalikko on avattuna ja jossa on korostettuna painikkeet, joiden avulla voidaan siirtää tiedostoa projektissa ylös tai alas:



Kuva 4. F#-projektissa valitun tiedoston kontekstivalikko, jossa korostettuna painikkeet, joilla siirretään tiedostoa ylös tai alas projektissa.

6.4 Ohjelman rakenne

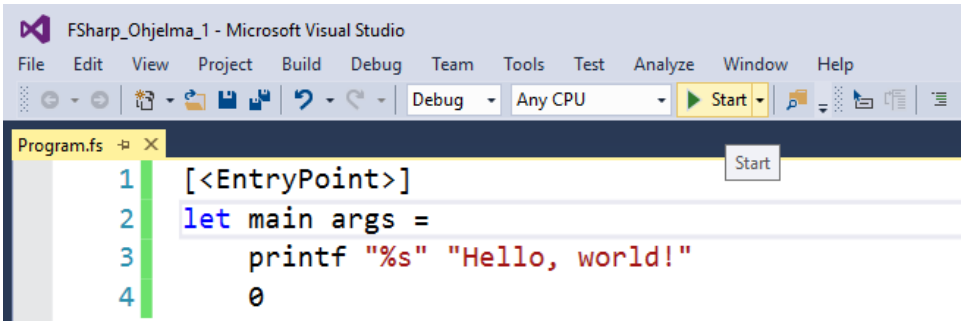
Fancherin (2014, 9) mukaan F#-applikaatiossa projektin viimeisessä tiedostossa määriteltyjä alustuksia käytetään oletuksena applikaation aloituspisteenä. Applikaation aloituspisteen voi myös asettaa määrittelemällä let-avainsanalla sidotun funktion ja asettamalla sille attribuutin EntryPoint. Tämä funktio on verrattavissa C#:n Main-metodiin. F#:ssa Entrypoint-attribuuttisen funktion nimi on mielivaltainen, mutta sillä on oltava parametrinaan merkkijonotaulukko ja paluutyyppinä kokonaisluku, kuten C#:ssa.

Alla on esimerkki applikaation aloituspisteestä, jossa tulostetaan ”Hello, world!”. Tulostus tehdään käyttämällä funktiota printf, joka muistuttaa toiminnallisuudeltaan C-kielen printf-funktiota. (Sume ym. 2016, 92) Koodissa funktioon printf syötetään kaksi parametria: ”%s” ja ”Hello, world!”. Ensimmäisellä parametrilla, formaattimerkkijonolla, määritellään, mikä on tulostettavan parametrin tyyppi, kuten C-kielessä. Formaattimerkkijonot analysoidaan käännösaikana (Sume ym. 2016, 92). ”%s” vastaa tyyppiä string (Sume ym. 2016, 93).

```
[<EntryPoint>]
let main args =
        printf “%s” ”Hello, world!”
        0
```

6.5 Ajaminen

F#-ohjelma on ajettavissa Visual Studiossa samalla tavalla kuin C#-ohjelma. Yksinkertaisen ”Hello, world!”-tekstin tulostamiseksi tehty demo käynnistettiin alla olevan kuvan mukaisesti painamalla Start-painiketta:



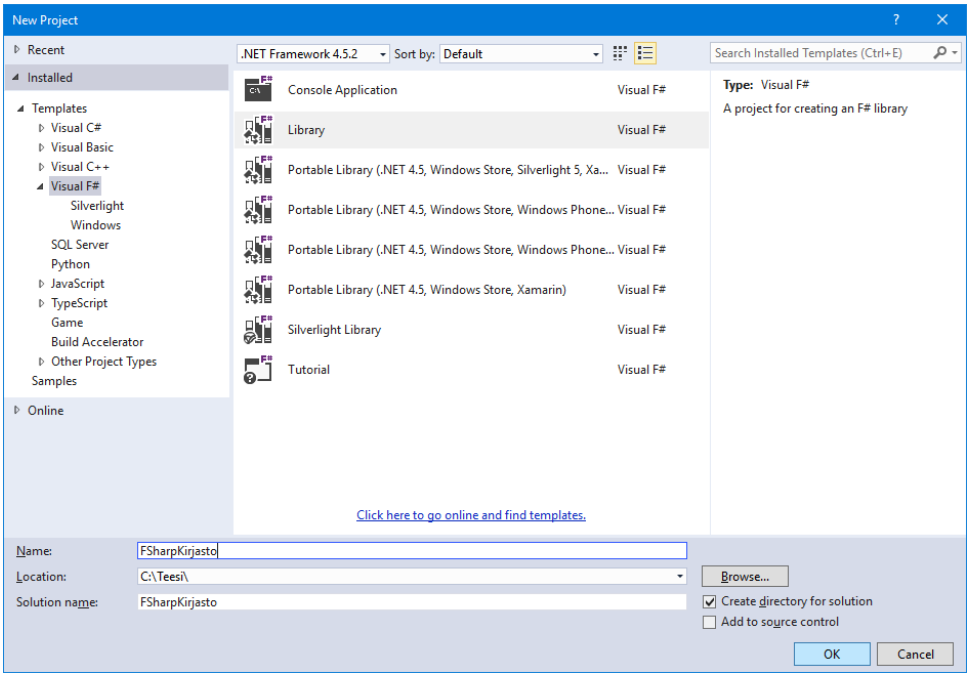
Kuva 5. Start-painike, jonka avulla F#-ohjelma voidaan käynnistää Visual Studiossa.

Ajetussa ohjelmassa tulostettiin avautuneeseen konsoli-ikkunaan ”Hello, world!”. Tämän jälkeen ohjelma sulkeutui. Ohjelmaan olisi voinut lisätä käyttäjän näppäinsyötteen lukemisen, jotta tulosteen ehtisi nähdä paremmin ennen ohjelman sulkeutumista, mutta työn muihin tavoitteisiin nähden asiaa pidetty merkityksellisenä. Oleellisena tässä vaiheessa pidettiin, että lyhyt F#-ohjelma onnistuttiin kirjoittamaan ja ajamaan edes jossakin ympäristössä.

7 F#-KIRJASTON TOTEUTUS .NET-YMPÄRISTÖÖN

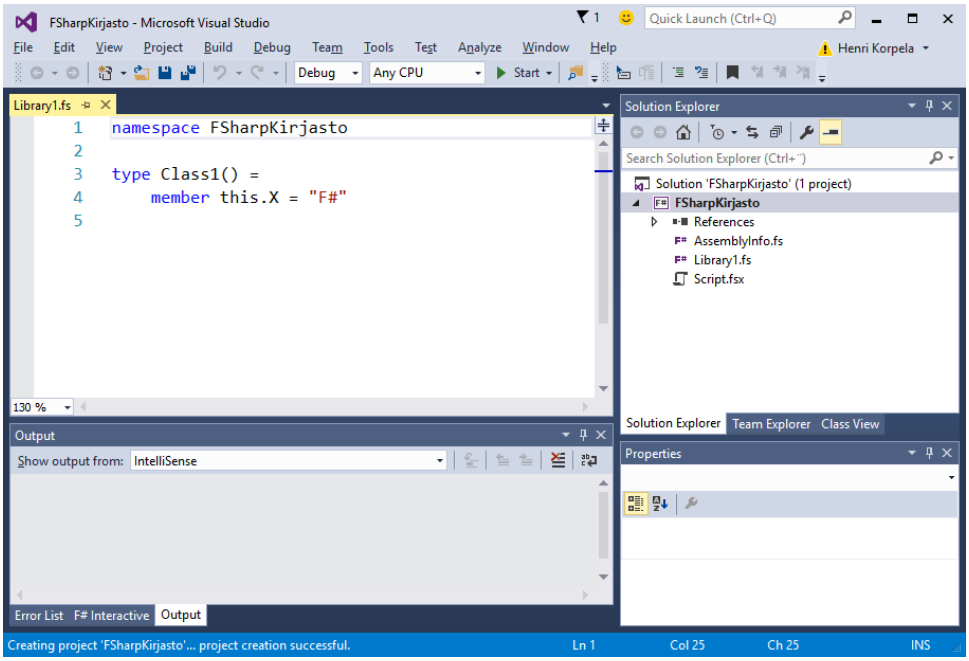
Visual Studio sisältää C#-ohjelmointikieltä varten projektimallin, jonka pohjalta voidaan toteuttaa ohjelmointikirjasto. Ajettavan F#-ohjelman rakenteessa oli mainitsemisen arvoisia eroja, johtuen muun muassa F#:n evaluointijärjestyksestä F#-tiedostojen suhteen. Mainitsemisen arvoisia eroja liittyy myös F#-ohjelmointikirjastoprojektiin, ja tässä kappaleessa läpikäydään F#-kirjaston toteutus johdatuksena käytännön osuuteen, jossa taitoja F#-kirjaston käyttämiseen C#-ohjelman kautta tarvittiin.

Esimerkkiohjelmointikirjastoa varten tehtiin uusi ratkaisu. Tämä tehtiin Visual Studion Templates -kohdan sisältä, kohdan ”Visual F#” alta. Avautuneesta listauksesta valittiin ”Library”. Projektin nimeksi annettiin FSharpKirjasto alla olevan kuvan mukaisesti:



Kuva 6. F#-ohjelmointikirjaston luominen nimellä FSharpKirjasto.

Oletuksena ohjelmointikirjastoprojektiin sisältyi kolme F#-tiedostoa: AssemblyInfo.fs, Library1.fs ja Script.fsx alla olevan kuvan mukaisesti:



Kuva 7. F#-ohjelmointikirjaston luomisen jälkeinen näkymä, jossa ohjelmakoodi on vasemmalla ja ratkaisun rakenne oikealla.

`#load`-direktiivi on eräs tapa ladata koodia FSI-sessioon. Direktiivi hyväksyy yhden tai useamman merkkijonoparametrin, joka sisältää absoluuttisen tai suhteellisen polun erillisiin lähdetiedostoihin. (Fancher 2014, 16.) Oletuksena kirjastoprojektin tiedostossa Script.fsx oli tällainen sisältö:

```
// Learn more about F# at http://fsharp.org. See
the 'F# Tutorial' project
// for more guidance on F# programming.

#load "Library1.fs"
open FSharpKirjasto

// Define your library scripting code here
```

Tästä esimerkkiohjelmointikirjastosta saatiin käyttökelpoinen lisäämällä sinne muutamia luokkia funktioineen. Tiedostosta Library1.fs poistettiin luokka Class1 toteutuksineen, ja tilalle toteutettiin luokat Fruit ja FruitApi seuraavasti:

```
type Fruit(name: string) =
    let _name = name

    member this.Name with get() = _name

type FruitApi() =
    member this.GetFruits() =
        [|
           Fruit("Apple");
           Fruit("Banana");
           Fruit("Grape")
```

```
        |]
    member this.GetFruitsByName(name) =
        this.GetFruits()
        |> Array.filter(fun f -> f.Name = name)
```

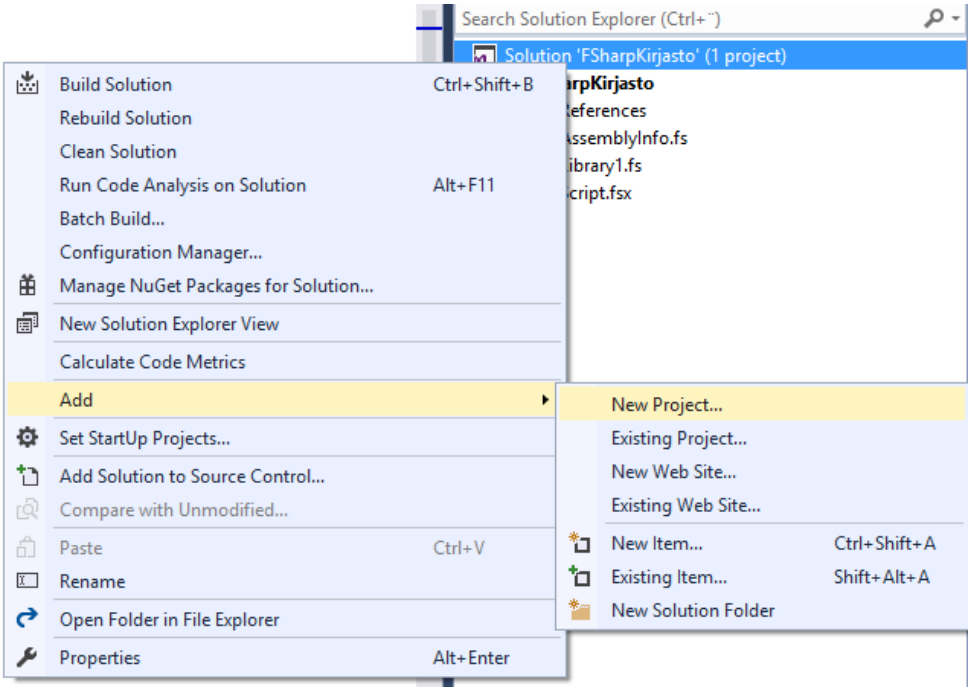
Luokka Fruit koostuu rakentajasta, johon syötetään parametrina merkkijonomuuttuja name. Tämä parametri sidotaan sitten luokan käyttäjältä piilotettuun arvoon _name. Alaviivaetuliite tälle arvolle lisättiin, jotta sen erottaisi julkisesti käytössä olevista kentistä, funktioista ja ominaisuuksista. Lisäksi luokassa on ominaisuus Name, jossa on ainoastaan get-funktio. Get-funktion toteutuksessa palautetaan kentän _name arvo. Ominaisuutta Name on tarkoitus hyödyntää luokassa FruitApi.

FruitApi-luokka on mallinnettu monenlaisten erilaisten, jo olemassaolevien API-luokkatoteutusten pohjalta. FruitApi on moniin tällaisiin toteutuksiin verrattuna yksinkertainen ja palvelee tässä yhteydessä ainoastaan kirjaston demoamistarkoitusta. Luokassa ei edes suoriteta mitään HTTP-pyyntöä REST API:iin, vaan käytettävä data on mockdataa eli testidataa.

Luokassa on metodi GetFruits. Metodin tarkoituksena on antaa käyttäjälle hedelmäolioita. Se ei ota vastaan parametreja (muuta kuin F#:n oletuksena sisään syötettävän unit-tyypin, koska muita parametreja ei ole annettu) ja palauttaa testidataa. Testidata koostuu taulukosta, jossa on kolme oliota instantioituina luokasta Fruit. Paluutyyppiä ei ole määritelty, koska F# päättelee paluutyypin paluuarvon perusteella.

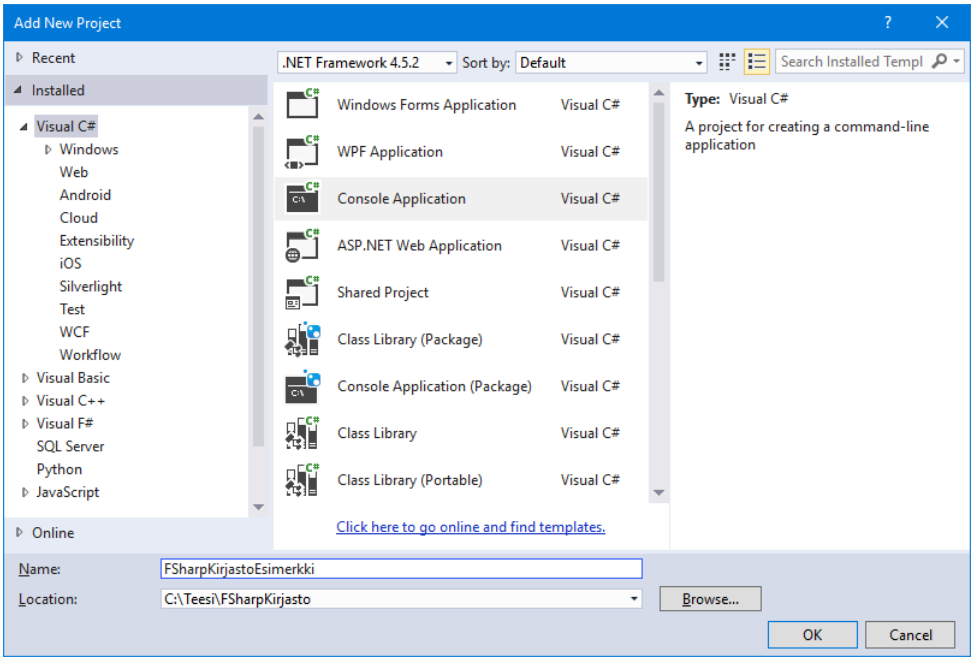
Vastaavaa paluuarvon päättelyä käyttää metodi GetFruitsByName. Metodista GetFruits poiketen se ottaa vastaan yhden parametrin name. Se on merkkijono, joka sisältää etsittävän hedelmän tai hedelmien nimen. Jälleen F# onnistuu päättelemään automaattisesti parametrin tyypin sen käytön perusteella. Metodin toteutuksessa hedelmien joukosta ”poimitaan” nimen mukaiset hedelmät. Tämän mahdollistaa metodi Array.filter. Suodatettava joukko muodostetaan käyttämällä metodin GetFruits paluuarvoa ja eteenpäin putkittamalla se Array.filter-metodin kanssa. Ehtona suodattamiselle on, että hedelmän nimi täsmää name-parametrin kanssa. Ehto on syötetty anonyymin funktion muodossa Array.filter-metodille.

Oli toteutuksen naiiviudesta mitä mieltä hyvänsä, toteutus toimii, mutta esimerkkikirjastoa voidaan hyödyntää vasta ajettavassa ohjelmassa. Sen vuoksi FSharpKirjasto-projektin rinnalle, samaan ratkaisuun, lisättiin FSharpKirjastoEsimerkki-projekti. Visual Studiossa navigoitiin ratkaisun FSharpKirjasto-ratkaisun kohdalla ja painettiin hiiren oikeata näppäintä kontekstivalikon avaamiseksi. Avautuneesta konstekstivalikosta valittiin Add ja sen alta New Project…. Kuvassa alla on nähtävissä New Project…-jäsen korostettuna:



Kuva 8. Korostettuna New Project…-jäsen nykyisen ratkaisun kontekstivalikon alla.

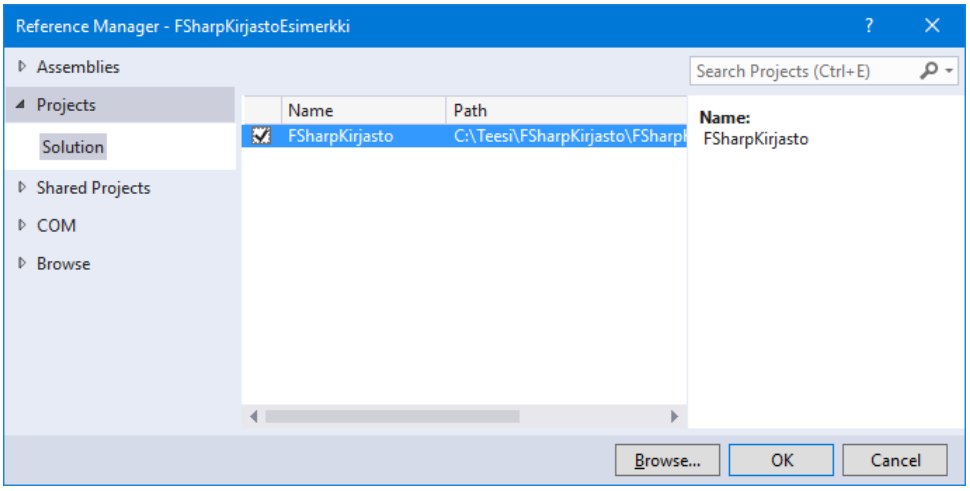
Esimerkkikirjaston testaamista varten valittiin C#-konsoliapplikaatioprojekti. Se löytyi vasemmalta asennettujen projektimallien hierarkiasta kohdasta Visual C# > Console Application. Alla olevassa kuvassa näkyy kyseinen projektimalli valittuna nimineen ja polkuineen.



Kuva 9. C#-konsoliapplikaatioprojektimalli valittuna ja nimeksi on annettu FSharpKirjastoEsimerkki.

Tämän jälkeen Solution Explorer -näkymässä oli nähtävissä uusi projekti FSharpKirjastoEsimerkki. C#-ohjelman ajamiseksi asetettiin tämä projekti

käynnistettäväksi projektiksi projektin kontekstivalikon kautta ja valitsemalla sieltä kohdan Set as StartUp Project. Jotta F#-esimerkkikirjastoa voitiin käyttää C#-ohjelmassa, lisättiin viittaus projektin References-kohdan kautta Solution Explorer -näkymässä. References-kohdan alta valittiin kontekstivalikon kautta Add Reference…. Avautuneesta ikkunasta valittiin tarjolla oleva vaihtoehto eli FSharpKirjasto. Oletuksena viittaukset rajattiin nykyisen ratkaisun projekteihin. Alla olevassa kuvassa näkyy FSharpKirjasto valittuna Refence Manager -näkymässä.



Kuva 10. FSharpKirjasto valittuna Reference Manager -näkymässä.

Painettiin OK. F#-kirjaston toiminnallisuuden varmistamiseksi rakennettiin F#-projekti, minkä jälkeen nimiavaruus FSharpProjekti oli käytettävissä C#-ohjelmassa.

Toiminnallisuuden demonstroimiseksi tehtiin testiohjelma C#-projektin Program.cs-tiedostoon (ks. liite 1). Pääpiirteittäin kuvattuna ohjelmassa tehdään uusi instanssi luokasta FruitApi ja kutsutaan instanssin metodia GetFruitsByName, parameterinaan ”Apple” eli palautetaan hedelmät, joiden nimi on Apple. C#-ohjelman toteutuksessa on käytetty C#:n LINQ-kirjaston metodia Select ilmentämään funktionaalista paradigmaa menetelmän mielenkiintoisuudesta ja lyhyydestä johtuen. Select-metodi muuntaa hedelmäoliotaulukon pelkästään niiden nimiksi sisältäväksi taulukoksi. Lopulta taulukon jäsenet yhdistetään merkkijonoksi staattisella metodilla String.join ja tulostetaan. Ohjelman ajamisen tuloksena tulostettiin konsoliin ”Apple”.

8 CASE TULOSTEEN JÄSENTELEMINEN

Opinnäytetyössä työn tilaajan tarpeisiin oli tarkoitus vastata jäsentelemällä tilaajalta saatua tulostetta. Jäsentelyllä tarkoitetaan tulosteessa esiintyvien tietoyksiköiden poimimista ja näiden tietojen kokoamista ohjelman tietorakenteisiin. Tulosteen jäsentelemistä varten laadittu terminologia selitetään kappaleen edetessä.

Tuloste on tekstimuodossa ja siinä on lähes kymmenentuhatta riviä, jossa on osastojen, segmenttien ja tuotteiden tietoja riveittäin. Näiden rivien seassa on sellaisia rivejä, joiden tarkoituksena on tarjota tulosteeseen ulkoasua. Ulkoasuun liittyvä tuloste ohitetaan lopullisessa ohjelmakoodin toteutuksessa C#:lla ja F#:lla, sillä tämä data on käyttökohteensa kannalta merkityksetöntä.

8.1 Tietorivit

Raportin jäsentelemisessä oli tarkoitus lukea tiedostosta ne rivit, joissa on joko osastojen, segmenttien tai tuotteiden tietoja. Osastoja, segmenttejä ja tulosteita kutsutaan yhteisellä nimellä tietoentiteetti. Näissä tietoentiteeteissä on olemassa hierarkia: osastot koostuvat segmenteistä ja segmentit koostuvat tuotteista.

Rivejä, joissa on jäsenneltävää tietoa, kutsutaan tietoriveiksi. Yhdellä tietorivillä on joko yhden osaston, yhden segmentin tai yhden tuotteen tiedot. Tietorivin rakenne on kuvailtu alla. Tietorivien rakenteen kuvailussa on käytetty nuolisulkeita osoittamaan tietoyksikköä. Tietoyksikkö on kuten olion attribuutti ohjelmakoodissa. Esimerkiksi jos tuote esitetään lopullisessa ohjelmakoodissa C#:lla ja F#:lla oliona, joka on instantioitu luokasta Tuote, niin tietoyksikkö ”tuotteen nimi”, joka on esiteltynä alla, on tämän olion attribuutti ”nimi”. Hakasulkeita on käytetty kuvaamaan tietorivin osiota, jossa on useampi tietoyksikkö.

Osaston tietoja sisältävä rivi on rakenteeltaan seuraavanlainen (ilman rivinvaihtoa):

> YHT. Os. <osaston tunniste> <osaston nimi> <luvut>
> TUOTE LKM <tuotelukumäärä>

Osaston tunnus on pituudeltaan 2 merkkiä ja sisältää ainoastaan numeerista tietoa.

Tulosteen jäsentelyssä osaston suhteen tarvitaan ainoastaan segmentin nimi ja tunniste, joten luvut-osio ja sitä seuraava data tällä tietorivillä ohitetaan kokonaan.

Segmentin tietoja sisältävä rivi on rakenteeltaan seuraavanlainen (myös ilman rivinvaihtoa):

YHT. SEGM. <segmentin tunniste> <segmentin nimi> <luvut>
TUOTE LKM <tuotelukumäärä>

Tulosteen jäsentelyssä segmentin suhteen tarvitaan ainoastaan segmentin nimi ja tunniste, joten luvut-osio ja sitä seuraava data tällä tietorivillä ohitetaan kokonaan.

Tuotteen tietoja sisältävä rivi on rakenteeltaan seuraavanlainen:

<tuotteen tunniste> <tuotteen nimi> [luvut]

Tuotteen tunniste on pituudeltaan 13 merkkiä. Joidenkin tuotteiden tunnisteessa saattaa olla edessä useampi nolla. Tuotteen tunniste voi olla esimerkiksi 3746385732635 tai useamman nollan kera 0000263356839.

Tuotteisiin liittyvät myös muut luvut osiossa nimeltä ”luvut”. Ne koostuvat 15 luvusta. Järjestyksessään ne vastaavat seuraavia tietoja:

1. Myynti kpl, kampanja
2. Myynti kpl, yhteensä
3. Myynti, kampanja
4. Myynti, yhteensä
5. Osuusprosentti, alaryhmä
6. Osuusprosentti, kampanja
7. Ale, rivi
8. Ale, yhteensä
9. Tuotto, kampanja
10. Tuotto, yhteensä
11. Tuotto-osuus, alaryhmä
12. Tuotto-osuus, norm
13. Tuottoprosentti, kampanja
14. Tuottoprosentti, yhteensä
15. Valikoimatunnus

Nämä numerot tallennetaan tuotteen tietoihin, mutta niitä ei muuten hyödynnetä ohjelmassa.

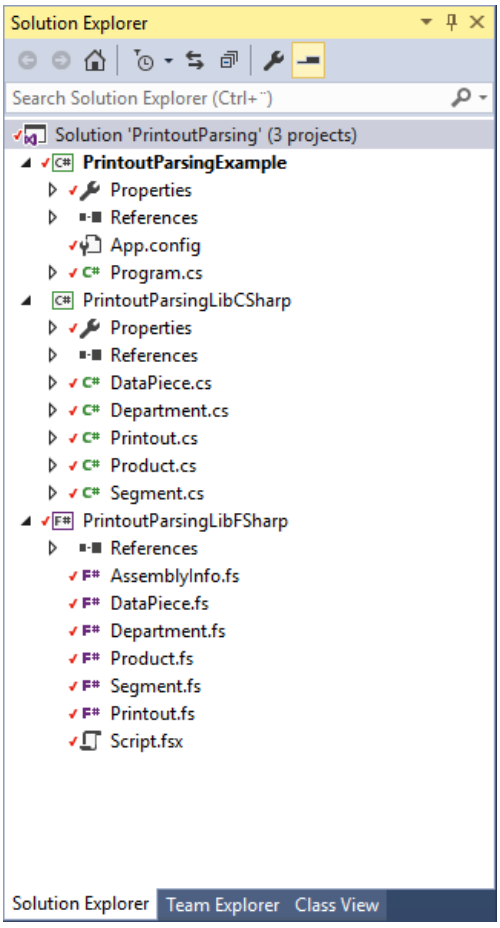
8.2 **Ratkaisun rakenne**

Jäsentelyssä poimittiin tiedot tietoriveiltä tietorakenteisiin. Tietoentiteetit asetettiin assosiatiivisiin taulukoihin, joissa avain oli entiteetin tunnus ja arvona oli entiteetti itsessään.

Tulosteen jäsentely toteutettiin Visual Studio 2015 Community -versiota käyttäen. Tehtävää varten tehtiin ratkaisu nimeltä PrintoutParsing, johon sisältyi kolme erilaista projektia:

- **PrintoutParsingExample** Suoritettava projekti. Tämä projekti on toteutettu C#:lla ja se on työn tavoitteissa C#-ohjelma, jolla on tarkoitus kutsua F#-ohjelmointikielellä toteutettua kirjastoa.
- **PrintoutParsingLibCSharp** .NET-kirjasto C#:lla toteutettuna, joka sisältää luokkia muuttujineen, metodeineen ja ominaisuuksineen, joita voidaan käyttää suoritettavassa projektissa. Kirjasto on olemassa vertailun vuoksi.
- **PrintoutParsingLibFSharp** .NET-kirjasto F#:lla toteutettuna, jonka toteutus vastaa toiminnallisuudeltaan ratkaisun projektia PrintoutParsingLibCSharp. Tämä projekti toimii työssä kirjastona, jota on tarkoitus kutsua C#-ohjelmointikielellä tehdyssä sovelluksessa.

.NET-kirjastojen osalta ratkaisussa on siis tehty sama toteutus C#:lla ja F#:lla. Alla olevassa kuvassa on Solution Explorer -näkymä, jossa kuvatut kolme projektia sijaitsevat:



Kuva 11. PrintoutParsing-niminen ratkaisu Solution Explorerissa, johon sisältyy kolme projektia.

8.3 Toteutuksen vaiheet

8.3.1 Prototyyppi

Toteutusta lähestyttiin aluksi siten, että tehtiin Javascriptillä NodeJS:ää käyttäen ohjelmakoodiprototyyppi, jossa lopullisen ratkaisun logiikka oli tarkoitus selvittää. Javascript valittiin sen heikon tyypityksen ja oliopohjaisuuden vuoksi. Dynaaminen tyypitys ei rajoita liiaksi olioiden koostamista, varsinkin, kun Javascriptissä assosiatiivisten taulukoiden ja olioiden käytössä on mahdollista käyttää samaa syntaksia. Prototyypin avulla ohjelman logiikan muotoileminen nähtiin käyvän nopeammin, kun tyypitykseen ei tuntunut rajoittavalta.

Alla on nähtävissä osa tästä Javascript-toteutuksesta Notepad++-tekstieditorissa:

```
"use strict" // Required by "let" command.

var fs = require('fs');

// Extensions to native types:
String.prototype.beginsWith = function(needle){
    return (this.indexOf(needle) === 0);
};

// Custom "classes":
/**
    Represents a department.
    @constructor
**/
var Department = function(options){
    options = options || {};
    this.name = options.name || null;
}
Department.idLength = 2;
Department.numberOfSpacesAfterSectionStartString = 4;
Department.sectionStartString = "YHT.Os.";

/**
    Represents a product of a store.
    @constructor
**/
var Product = function(options){
    options = options || {};
    this.name = options.name || null;
}
Product.idLength = 13;

/**
    Represents a product segment.
    @constructor
**/
var Segment = function(options){
    options = options || {};
    this.departmentId = null;
```

Kuva 12. Tulosteen jäsentely -ohjelman Javascript-toteutus osittain näkyvissä Notepad++-tekstieditorissa.

Tämän vaiheen jälkeen lähdettiin kääntämään Javascript- prototyyppikoodia C#-kirjastoksi lopulliseen ratkaisuun.

8.3.2 Tietoentiteettiluokat

Visual Studioon tehtiin uusi projekti nimeltä PrintoutParsing ja siihen lisättiin projekti PrintoutParsingLibCSharp C#-toteutusta varten ja projekti PrintoutParsingLibFSharp F#-toteutusta varten.

Molempiin projekteihin lisättiin kooditiedostot DataPiece, Department, Product ja Segment. Luokkien yhteydessä, ellei toisin mainita, pätee sama selostus, vaikka syntaksi on erilainen. DataPiece toimi kantaluokkana näille muille tietoentiteettiluokille ja sisältää tietoentiteeteillä keskeiset ominaisuudet Id ja Name. Alla on järjestyksessään luokan DataPiece C#- ja F#-toteutus:

C#

```
namespace PrintoutParsingCSharp{
    public class DataPiece{
        // Properties
        public int Id{
            get;
            protected set;
        }
        public string Name{
            get;
            protected set;
        }

        // Constructors and destructors
        public DataPiece(string name = null){
            this.Name = name;
        }
    }
}
```

F#

```
namespace PrintoutParsingFSharp

type DataPiece (name: string) =
    /// Properties
    member val Id = 0 with get, set
    member val Name = name with get, set
```

Toteutuksissa ovat ominaisuudet Id ja Name on get- ja set-funktion kera F#:n termein ilmaistuna. Rakentaja ottaa vastaan name-parametrin, joka on käytettävissä Name-ominaisuuden kautta. C#-toteutuksessa name on null, mikäli arvoa ei anneta. F#-toteutuksessa arvo on annettava.

Huomionarvoista on, kun vertaa C#- ja F#-toteutusta, F#-toteutus on paljon lyhyempi. Tämä on nähtävissä myös muiden luokkien yhteydessä.

Luokka Deparment, joka muiden tietoentiteettien tapaan perii luokasta DataPiece, sisältää yhden osaston tietoja. Se on toteutukseltaan seuraavanlainen:

C#

```
namespace PrintoutParsingCSharp{
    public sealed class Department : DataPiece{
        // Constructors & destructors:
        public Department(string name = null) : base(name){

        }

        // Static member data:
        public static int IdLength{
            get;
        } = 2;
        public static int NumberOfSpacesAfterSectStartString{
            get;
        } = 4;
        public static string SectionStartString {
            get;
        } = "YHT.Os.";
    }
}
```

F#

```
namespace PrintoutParsingFSharp

type Department (name: string) =
    inherit DataPiece(name)

    // Static member data
    static member IdLength = 2
    static member NumberOfSpacesAfterSectStartString = 4;
    static member SectionStartString = "YHT.Os.";
```

Department-luokka ei lisää luokan instanssin kautta käytettäviä jäseniä perittyjen jäsenten lisäksi, mutta sisältää staattisia ominaisuuksia, joita käytetään kuin vakioita. Tunnisteen pituus saadaan ominaisuuden IdLength kautta. Välilyöntien määrä tunnisteesta osaston numeroon saadaan ominaisuuden NumberOfSpacesAfterSectStartString kautta. Osaston sisältävän tietorivin alussa oleva merkkijono, jonka perusteella osaston tiedot sisältävä rivi voidaan tunnistaa, saadaan taas ominaisuuden SectionStartString kautta.

Product-luokka tehtiin seuraavasti:

C#

```
namespace PrintoutParsingCSharp{
    public sealed class Product : DataPiece{
```

```
        // Constructors & destructors:
        public Product(string name = null) : base(name){

        }

        // Properties:
        public float AleRow { get; set; }
        public float AleTotal { get; set; }
        public float SelectionId { get; set; }
        public float PortionSubgroup { get; set; }
        public float PortionCampaign { get; set; }
        public float ProfitCampaign { get; set; }
        public float ProfitTotal { get; set; }
        public float ProfitPercentCampaign { get; set; }
        public float ProfitPercentTotal { get; set; }
        public float ProfitPortionSubgroup { get; set; }
        public float ProfitPortionNorm { get; set; }
        public float SaleCampaign { get; set; }
        public float SaleTotal { get; set; }
        public float SaleUnitCampaign { get; set; }
        public int SegmentId{ get; set; }

        // Static member data:
        public static int IdLength{
            get;
           set;
        } = 13;
    }
}
```

F#

```
namespace PrintoutParsingFSharp

type Product (name: string) =
    inherit DataPiece(name)

    /// Member data
    member val AleRow = 0.0f with get, set
    member val AleTotal = 0.0f with get, set
    member val PortionSubgroup = 0.0f with get, set
    member val PortionCampaign = 0.0f with get, set
    member val ProfitCampaign = 0.0f with get, set
    member val ProfitTotal = 0.0f with get, set
    member val ProfitPercentCampaign = 0.0f with get, set
    member val ProfitPercentTotal = 0.0f with get, set
    member val ProfitPortionSubgroup = 0.0f with get, set
    member val ProfitPortionNorm = 0.0f with get, set
    member val SaleCampaign = 0.0f with get, set
    member val SaleTotal = 0.0f with get, set
    member val SaleUnitCampaign = 0.0f with get, set
    member val SelectionId = 0 with get, set

    // Static member data
    static member IdLength = 13
```

Kaikista C#- ja F#-kirjastoprojektin luokista Product-luokka sisältää eniten ominaisuuksia. Luokalla on samankaltaisuuksia Department-luokkaan perinnän ja staattisen ominaisuuden IdLength myötä. Tämä staattinen omi-

naisuus ajaa vastaavaa asiaa kuin Department-luokan IdLength. Muut Product-luokan ominaisuudet on tarkoitettu varastoimaan tuotetietorivin lukuja. Nämä luvut tarkoittivat niitä lukuja, jotka asetetaan luokassa Printout, mutta joiden arvoja ei ajettavassa projektissa PrintoutParsingExample käytetä.

Segmenttejä, joihin tuotteet sisällytetään, tehtiin vielä yksi entiteettiluokka Segment. C#:lla ja F#:lla toteutus näyttää tällaiselta:

C#

```
using System.Collections.Generic;

namespace PrintoutParsingCSharp{
    public sealed class Segment : DataPiece{
        // Constructors & destructors:
        public Segment(string name = null) : base(name){

        }

        // Properties:
        public int DepartmentId{
            get;
            set;
        }
        public List<Product> Products{
            get;
            set;
        } = new List<Product>();

        // Static member data:
        public static int IdLength{
            get;
        } = 4;
        public static string SectionStartString{
            get;
        } = "YHT.SEGM";
    }
}
```

F#

```
namespace PrintoutParsingFSharp

open System.Collections.Generic

type Segment (name: string) =
    inherit DataPiece(name)

    /// Member data
    member this.DepartmentId = 0
    member this.Products = new List<Product>();

    /// Static member data
    static member IdLength = 4
    static member SectionStartString = "YHT.SEGM"
```

Luokan instanssin tarvitsee tietää, mihin osastoon se kuuluu. Tätä varten on ominaisuus DepartmentId. Instanssi voi sisältää myös tuotteita, jotka

varastoidaan ominaisuuden Products kautta käytettävään listaan. Viimeisenä luokassa on staattisia ominaisuuksia. IdLength määrittelee segmentin tunnisteen pituuden tietorivillä ja SectionStartString määrittelee merkkijonon, jonka perusteella segmentin tiedot sisältävä rivi voidaan tunnistaa.

8.3.3 Printout-luokka

Tietoentiteetit toimivat dataa jäsentelevinä yksiköinä, mutta näiden hyödyntäminen kulminoituu luokkaan Printout. Luokan vastuulla on tulostetiedoston lukeminen muistiin ja datan läpikäyminen rivi riviltä, sijoittaen jäsennelty data tietoentiteetteihin. Työssä Printout-luokka muodosti suurimman ja haastavimman kokonaisuuden.

Luokassa rakentajaan syötetään tulostetiedoston nimi (ks. liite 2 ja 3). Tätä käytetään sitten julkisessa metodissa Parse, jossa tiedostosta luetaan rivit ja rivit syötetään parametrina yksityiseen metodiin ParseLine. Alla Parse-metodin toteutus F#:lla:

```
let lines = File.ReadAllLines(filename)
lines
  |> Seq.iter(fun line -> this.ParseLine(line))
  |> ignore
```

C#-toteutuksessa on for-silmukka, mutta F#-toteutuksessa käytetään Seq.iter-metodia. Nimensä mukaisesti se iteroi, tässä tapauksessa, taulukon lines. Molemmissa toteutuksissa jokaista riviä kohden kutsutaan yksityistä metodia ParseLine.

ParseLine-metodissa on merkittävimmät erot C#- ja F#-toteutuksen välillä. F#-toteutuksessa on otettu jopa vapauksia sitä mukaa, kun F#:n tarjoamia funktionaalisia rakenteita käytännön osuutta työstettäessä oivallettiin. Metodissa käytetään aluksi metodia String.Trim poistamaan parametrista line ylimääräiset alku- ja loppupään välilyönnit. Tämän jälkeen ero C#:n ja F#:n kanssa tulee haaroitusmekanismissa. C#:ssa käytettiin if-else-lausekketta. F#:ssa haaroitus perustui match-lausekkeeseen eli kyseessä oli hahmontunnistus, jossa hahmoina olivat järjestyksessään true ja false. Molemmissa menetelmissä ehto oli sama: rivin pituuden tuli olla enemmän kuin 0. Alla on F#-toteutuksesta ote, jossa on sidonnan line pohjalta muodostettu arvo lineTrimmed, ja match-ilmaus ehdon kanssa:

```
let lineTrimmed = line.Trim()

match lineTrimmed.Length > 0 with
| true -> …
| false -> …
```

Jos ehto ei täyty, kummassakaan toteutuksessa ei tehdä mitään. C#:ssa mennään eteenpäin, F#:ssa palautetaan unit-tyyppinen arvo, mutta sitä ei käytetä mihinkään. Ehdon täyttyessä siirrytään jälleen ehtorakenteisiin,

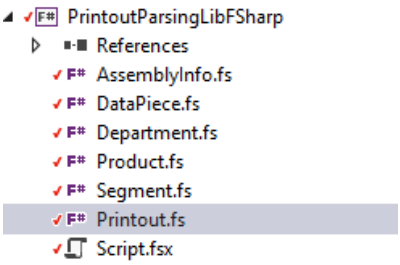
mutta tällä kertaa erot eivät ole merkittävät, koska F#:ssa käytettiin imperatiivista menetelmää eli if-lausekkeita haaroitusmekanismina. Syynä tähän oli menetelmän tuttuus ja epävarmuus siitä, mikä olisi ollut hyvä funktionaalinen tapa toteuttaa vastaava toiminnallisuus. Joka tapauksessa näillä mainituilla imperatiivisilla ehtorakenteilla tarkistetaan, mikä tietorivi on kyseessä. Jokaiseen näistä tietorivien käsittelyistä kuuluu moninaisia merkkijonomanipulaatioita sekä tyyppimuunnoksia. C#- ja F#-toteutuksessa on hyvinkin samanlaiset merkkijonomanipulaatiot, koska samoja .NET-sovelluskehityskehyksen metodeita käytettiin. Tyyppimuunnoksissa ja toistorakenteissa on kuitenkin eroja. Kun C#:ssa turvauduttiin tyyppimuunnoksissa muun muassa metodiin float.Parse, F#:ssa suosittiin putkituksia. Toistorakenteet F#:ssa eivät sisältäneet for-silmukoita, vaan metodeita Seq.iter ja Seq.map. Seq.map toimii kuten C#:n LINQ-kirjaston Select-metodi: se muuntaa taulukon datan jäsenkohtaisesti toisenlaiseksi. Tuotteen luvut-osion lukujen käsittelyssä tämä tulee ilmi (ks. liite 3). Luvut putkitetaan merkkijonotaulukkona metodiin Seq.map ja paluuarvona on uusi taulukko, joka sisältää numerot liukulukuina.

Tuotteeseen liittyvä tietorivin datan käsittely oli jälleen osuus, jossa C#- ja F#-toteutus eroavat merkittävästi. If-lauseke C#-toteutuksessa korvattiin match-ilmauksella. Numeroiden käsittely taas hyödynsi metodeita Seq.map ja Seq.toArray, mutta ei yhtäkään for-silmukkaa. Lisäksi F#-toteutuksessa ei ole yhtäkään muuttuvaa sidontaa.

Molemmissa toteutuksissa tietorakenteet ovat samalla tavoin käsittelyssä. Tietorivien löytyessä niiden data muunnetaan ja lisätään tietoentiteettikokoelmaan, joka on Dictionary-tyypin muodossa. Muotona tälle on avain-arvopari, jossa avaimena on entiteetin tunniste ja arvona itse entiteetti.

8.3.4 F#-kirjaston lopullinen rakenne

Mainittakoon luokasta Printout ja sen suhteesta projektiin sen verran, että luokka asetettiin projektissa viimeiseksi F#-kooditiedostoksi Script.fsx-tiedostoa lukuun ottamatta alla olevan kuvan mukaisesti:



Kuva 13. Printout.fs-tiedosto valittuna Solution Explorer -näkymässä ja asetettuna projektin viimeiseksi F#-kooditiedostoksi Script.fsx-tiedostoa lukuun ottamatta.

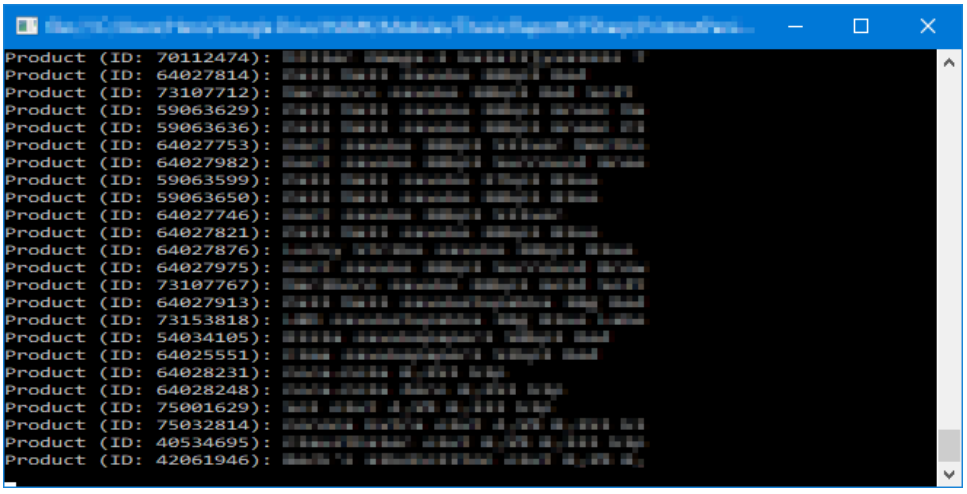
Syynä tähän oli aiemmin kuvattu F#:n evaluointijärjestys. Myös DataPiece.fs-tiedosto siirrettiin evaluontijärjestyksen vuoksi projektissa sitä käyttävien tietoentiteettiluokkien yläpuolelle. C#-projektissa ei tarvinnut tehdä mitään vastaavaa.

8.3.5 Testiohjelma C#:lla

Luvussa 7 kerrotaan, miten toteutettiin F#-kirjasto .NET-ympäristöön. Sama tehtiin tässäkin tapauskohtaisessa tehtävässä. Viimeisen tapauskohtaisen tehtävän suorittamiseksi lisättiin luvussa 7 esitettyyn tapaan C#-ohjelma, josta F#-kirjastoa kutsuttiin. Ratkaisuun lisättiin sitä varten C#-projekti PrintoutParsingExample. Jotta F#-kirjasto toimisi, lisättiin uuteen projektiin uusi viittaus PrintoutParsingLibFSharp-projektiin. Sitten rakennettiin F#-kirjastoprojekti.

Projektin PrintoutParsingExample mukana tulleeseen Program.cs-tiedostoon tehtiin tarvittavat muutokset, jotta F#-kirjasto saatiin testattua (ks. liite 4). Testiohjelmassa tehdään aluksi printout-niminen olio tyypiltään Printout. Tämä on siis luokka, joka F#-kirjastossa toteutettiin. Sen jälkeen kutsutaan olion metodia Parse, jotta osastot, segmentit ja tuotteet kerättäisiin olioon. Lopulta, pääpiirteittäin, tietoentiteetit tulostetaan entiteetti kerrallaan. Välissä odotetaan käyttäjän syötettä.

Ohjelmaa ei voi ajaa suoraan, vaan se tarvitsee komentojonoargumentin, johon on määritelty tulostetiedoston polku. Argumentti tarjottiin ja ohjelma ajettiin. Alla on kuva lopullisesta, ajetusta ohjelmasta (tietosuojasyistä otsikkopalkki ja osa tulosteesta on sumennettuna):



Kuva 14. C#-ohjelma, joka F#-kirjastoa käyttää, ajettuna konsolissa. Tulosteessa nähtävissä tuotteen tunniste ja tuotteen nimi.

9 TULOKSET

Työn tuloksena syntyi tehtävän toimeksiantajan mukaisen vaatimukseen vastannut kokonaisuus: C#-ohjelmointikielellä tehty ohjelma, joka käyttää F#-ohjelmointikielellä tehtyä kirjastoa. Tehtävän suorittaminen koettiin haastavaksi, paikoin jopa vaikeaksi, sillä tehtävän luonne vaatii uudelleenorientoitumista, mukautumista ajatusmalliin, jossa imperatiivinen, käskypohjainen menetelmä ratkoa ongelmia sivuutetaan, ainakin osittain, funktionaalisen, enemmän matemaattisiin todistuksiin perustuvan ongelmanratkaisumenetelmän tieltä. Mietittäessä sivuvaikutuksettomuutta, viittauksellista läpinäkyvyyttä, funktioita ensimmäisen luokan arvoina ja muita erilaisia funktionaalisen ohjelmoinnin konsepteja, yhden asian nähtiin kristallisoituvan: funktionaalisella ohjelmoinnilla on merkitystä ja sen on parasta tulla jäädäkseen.

Vaikka kokonaisuus vastasi tavoitteita, työn edetessä kävi selväksi, että F#-toteutuksessa on parantamisen varaa. Koodia luokan Printout kohdalla olisi voinut jakaa useampaan pienempään funktioon ja nimenomaan hyödyntää periaatetta, jossa funktio tekee vain ja ainoastaan yhden asian, toki kohtuuden rajoissa, ja mielellään niin, että viittauksellinen läpinäkyvyys pääsisi oikeuksiinsa. Hahmontunnistus koettiin hyödyllisenä rakenteena, mutta sen käyttökohteiden oivaltaminen jäi suppeaksi. Taitamattomuuden vuoksi suosittiin liikaa imperatiivisia rakenteita.

Toimeksiantajan kanssa käydyt keskustelut toivat myös lisävalaistusta parempien ratkaisujen äärelle, joita tässä työssä ei ehditty toteuttamaan. Esimerkkinä tästä oli tietorivien jäsentely. Ilmeni, että rivissä olevilla tiedoilla oli ennaltamäärätty maksimipituus, jonka pohjalta olisi todennäköisesti saatu aikaiseksi lyhyempi ja luettavampi C#- ja F#-toteutus.

Funktionaalisen ohjelmoinnin ilmentäminen F#:lla herätti monia näkemyksiä. F#:n kautta työssä tutustuttiin moniin funktionaalisiin rakenteisiin ja kieli täyttää odotuksia kompaktimpana ja helpommin ymmärrettävänä kielenä erityisesti siinä suhteessa, kuinka kieli pyrkii välttämään sivuvaikutuksia. Toisaalta syntaksi jätti monesti toivomisen varaa. Kyseessä saattoi olla oudot päätökset kielen suunnittelussa syntaksin suhteen, esimerkiksi taulukoissa, tai henkilökohtainen kokemattomuus ja ymmärtämättömyys F#:n kokonaiskuvasta ohjelmointikielenä. Oli aihe moitteeseen mikä hyvänsä, kielellä ohjelmointi kärsi tällaisista vivahteista, kun täytyi ajoittain pysähtyä virheeseen ja muistella, kuinka syntaksi menikään.

Loppukaneettina voidaan todeta, että funktionaalisessa ohjelmoinnissa nähtiin olevan potentiaalia, mutta sen ilmentämistä F#-ohjelmointikielellä olisi aihetta miettiä. Työn tuloksena ollut ratkaisu osoitti F#:n ja funktionaalisen ohjelmoinnin tuomia etuja, mutta ei niin runsaasti kuin oli odotettu, johtuen pääosin puutteellisesta funktionaalisen paradigman hyödyntämisestä.

10 YHTEENVETO

Funktionaaliselle ohjelmoinnille, jota suurimmalta osin F#-ohjelmointikielen kautta työssä läpikäytiin, ei löydy tarkkaa määritelmää, mutta sen merkitystä on mahdollista avata sen konseptien avulla, muun muassa funktiot ensimmäisen luokan arvoina, sivuvaikutuksettomuus ja viittauksellinen läpinäkyvyys. Työssä esiteltiin funktionaalisen ohjelmoinnin perustaa lambdakalkyylissä, ja varhaisimpien funktionaalisten ohjelmointikielten kautta johdateltiin lukija F#-ohjelmointikielen pariin.

F# kuvailtiin ensisijaisesti funktionaalisena ohjelmointikielenä, joka sallii myös muiden paradigmojen käytön, kuten olio-ohjelmointi. Työssä F#:n käyttöä .NET-ympäristössä pohjustettiin aluksi perusrakenteiden kautta, kuten sidonnat, tyypitys, funktiot ja taulukot. Sitten siirryttiin olio-ohjelmointiin F#:lla luokkineen ja rajapintoineen, sekä sen jälkeen funktionaalisen paradigman kattavaan osuuteen, jossa esiteltiin muun muassa lambdafunktiot, putkitus ja hahmontunnistus. Kaikissa näissä vaiheissa pyrittiin perehdyttämään lukija, toimeksiantajan toiveesta, asiaan C#-ohjelmoijan näkökulmasta.

Työn muihin osioihin nähden laajan teoriapohjan vastapainona oli käytännön osuus, jossa selostettiin F#-ohjelmointikielen käyttöönotto Visual Studiossa. Ohjelma toimii muutenkin hallitsevana työkaluna esimerkkikirjastototeutuksessa ja sitä seuraavassa case Tulosteen jäsentely -kokonaisuudessa. Esimerkkikirjastototeutuksessa tehtiin F#-kirjasto luokkineen, joita sitten kirjaston kylkiäisenä tehdyssä C#-ohjelmassa käytettiin. Tämä toteutus muodosti perustan case Tulosteen jäsentely -kokonaisuudelle, jossa vastaavalla tavalla toteutettiin F#-kirjasto ja kirjastoa käyttävä C#-ohjelma.

Tapauskohtainen tehtävä Tulosteen jäsentely raportoitiin ja samalla vertailtiin C#- ja F#-toteutusten eroja. Potentiaalisia kehittämisen kohteita ei tässä kohtaa tuotu esille, mutta niitä käsiteltiin työn lopuksi tuloksien yhteydessä. Tulokset kiteytettiin loppukaneettiin, että funktionaalisessa ohjelmoinnissa nähtiin potentiaalia, mutta että sen ilmentämistä F#-ohjelmointikielellä olisi aihetta miettiä. Työn tulos osoitti F#:n ja funktionaalisen ohjelmoinnin tuomia etuja, mutta ei niin runsaasti kuin odotettiin. Tämä johtui pääosin puutteellisesta funktionaalisen paradigman hyödyntämisestä.

LÄHTEET

A Functional Programming Language: ML (n.d.). Oppimateriaali. The University of Texas at Dallas. Haettu 26.3.2017 osoitteesta http://www.utdallas.edu/~gupta/courses/apl/ML-1.html

Carter, P. ja Wenzel, M. (2016a). Primitive Types (F#) | Microsoft Docs. Haettu 30.1.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/primitive-types

Carter, P. ja Wenzel, M. (2016b). Unit Type (F#) | Microsoft Docs. Haettu 21.3.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/unit-type

Carter, P., Latham, L. ja Wenzel, M. (2016a). Classes (F#) | Microsoft Docs. Haettu 21.5.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/classes

Carter, P., Latham, L. ja Wenzel, M. (2016b). Classes (F#) | Microsoft Docs. Haettu 5.7.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/interfaces

Cartel, P., Latham, L., Wenzel, M. ja Marcio, R. (2016). Pattern Matching (F#) | Microsoft Docs. Haettu 24.7.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching

Cartel, P., Latham, L., Wenzel, M. ja Onderka, P. (2016). Resource Management: The use keyword (F#) | Microsoft Docs. Haettu 20.6.2017 osoitteesta https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/resource-management-the-use-keyword

Fancher, D. (2014). The Book of F#: Breaking Free with Managed Functional Programming.

F# Software Foundation (n.d.). F# Software Foundation. Haettu 18.1.2017 osoitteesta http://fsharp.org/

F# Software Foundation (n.d.). About F# | The F# Software Foundation. Haettu 20.3.2017 http://fsharp.org/about/index.html#documentation

Graham, P. (2002). Revenge of the Nerds. Haettu 10.5.2017 osoitteesta www.paulgraham.com/icad.html

Hutton, G. (2002). FAQ for comp.lang.functional. Haettu 10.4.2017 osoitteesta http://www.cs.nott.ac.uk/~pszgmh/faq.html

Introduction - HaskellWiki. (n.d.). Haettu 28.1.2017 osoitteesta https://wiki.haskell.org/Introduction

Introduction to Functional Programming. Haettu 18.1.2017 osoitteesta http://tocs.ulb.tu-darmstadt.de/4649869.pdf

Marlow, S. (2010a). 1 Introduction. Haettu 21.5.2017 osoitteesta https://www.haskell.org/onlinereport/haskell2010/haskellch1.html#x6-90001

Marlow, S. (2010b). Preface. Haettu 21.5.2017 osoitteesta https://www.haskell.org/onlinereport/haskell2010/haskellli2.html#x3-2000

McCarthy, J. (1996). Introduction. Haettu 2.3.2017 osoitteesta http://www-formal.stanford.edu/jmc/history/lisp/node1.html#SECTION00010000000000000000

Milner, R. Tofte, M., Harper, R., MacQueen, D. (1997). The Definition of Standard ML (Revised). Haettu 26.3.2017 osoitteesta http://sml-family.org/sml97-defn.pdf

Myyrä, T. (2011). Tietokoneshakki funktionaalisella ohjelmoinnilla. Haettu 26.2.2017 osoitteesta https://www.theseus.fi/bitstream/handle/10024/29841/tietokoneshakki.pdf?sequence=2

Petricek, T. ja Skeet, J. (2010). Overview: Introducing F# and Functional Programming. Haettu 26.3.2017 osoitteesta https://msdn.microsoft.com/en-us/library/hh297121(v=vs.100).aspx

Syme, D., Alimov A., Battocchi, K., Fisher, J., Hale, M., Hu, J., Hoban, L., Liu, T., Lomov, D., Margetson, J., McNamara, B., Pamer, J., Orwick, P., Quirk, D., Ransom, K., Smith, C., Taveggia, M., Malayeri, D., Chae, W., Matsveyeu, U., Atkinson, L., ym. (2016). The F# 4.0 Language Specification. Haettu 18.1.2017 osoitteesta http://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-latest.pdf

Terkki, E. (2012). *Funktionaalinen ohjelmointi.* LuK-tutkielma. Tietojenkäsittelytieteiden tutkinto-ohjelma. Helsingin yliopisto. Haettu 29.3.2017 osoitteesta https://www.cs.helsinki.fi/u/eaterkki/kandi.pdf

Liite 1

C#-OHJELMA F#-ESIMERKKIOHJELMOINTIKIRJASTON KÄYTTÖÖN

```
using System;
using System.Linq;

using FSharpKirjasto;

namespace FSharpKirjastoEsimerkki
{
    class Program
    {
        static void Main(string[] args)
        {
            var fruits = new FruitApi().GetFruitsByName("Apple");
            var fruitNames = fruits.Select(f => f.Name).ToArray();
            Console.WriteLine(string.Join(", ", fruitNames));
            Console.ReadKey();
        }
    }
}
```

Liite 2

PRINTOUT-LUOKKA – C#-TOTEUTUS

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;

namespace PrintoutParsingCSharp{
    public class Printout{
        // Member data
        private Segment currentSegment = new Segment();
        private string filename;

        // Constructors and destructors
        public Printout(string filename = null){
            this.filename = filename;
        }

        // Properties
        public Dictionary<int, Department> Departments{
            get; set;
        } = new Dictionary<int, Department>();

        public Dictionary<int, Product> Products{
            get; set;
        } = new Dictionary<int, Product>();

        public Dictionary<int, Segment> Segments{
            get; set;
        } = new Dictionary<int, Segment>();

        // Member functions (methods)
        private void ParseLine(string line){
            string lineTrimmed = line.Trim();
            int lineLength = lineTrimmed.Length;

            // If this line is not empty:
            if (lineTrimmed.Length > 0){
                // If a line with department information was found:
                if (lineTrimmed.StartsWith(Department.SectionStartString)){
                    int indexOfDepartmentNameBegin =
                        Department.SectionStartString.Length +
                        Department.NumberOfSpacesAfterSectStartString +
                        Department.IdLength + 1;
                    string sectionStringEnd = lineTrimmed.Substring(
                        indexOfDepartmentNameBegin, lineLength - indexOf-
DepartmentNameBegin);
                    // Finding index of the first 4 spaces:
                    int indexOfDepartmentNameEnd = sectionStringEnd.IndexOf("
");

                    int departmentId = int.Parse(lineTrimmed.Substring(
                        Department.SectionStartString.Length + Depart-
ment.NumberOfSpacesAfterSectStartString,
                        Department.SectionStartString.Length + Depart-
ment.NumberOfSpacesAfterSectStartString + Department.IdLength -
                            (Department.SectionStartString.Length + Depart-
ment.NumberOfSpacesAfterSectStartString)
                    ));
```

```
                    string departmentName = sectionStringEnd.Substring(0, in-
dexOfDepartmentNameEnd);

                    var department = new Department(departmentName);
                    Departments.Add(departmentId, department);
                }
                else if (lineTrimmed.StartsWith(Segment.SectionStartString))
                {
                    int indexOfSegmentNameEnd = lineTrimmed.IndexOf("    ");

                    int segmentId = int.Parse(lineTrimmed.Substring(
                        Segment.SectionStartString.Length + 1,
                        Segment.SectionStartString.Length + 1 + Seg-
ment.IdLength -
                            (Segment.SectionStartString.Length + 1)
                    ));
                    string segmentName = lineTrimmed.Substring(
                        Segment.SectionStartString.Length + 2 + Seg-
ment.IdLength,
                        indexOfSegmentNameEnd - (Segment.SectionStart-
String.Length + 2 + Segment.IdLength)
                    );

                    // For each product of the current segment:
                    foreach(var product in currentSegment.Products){
                        product.SegmentId = segmentId;
                    }

                    var segment = new Segment(segmentName);
                    Segments.Add(segmentId, segment);
                    currentSegment = new Segment();
                }
                else if (lineLength >= Product.IdLength)
                {
                    int productId;
                    bool productIdParsedSuccessfully = int.Try-
Parse(lineTrimmed.Substring(0, Product.IdLength), out productId);

                    // If product ID was parsed successfully, then we are
dealing with a product line:
                    if (productIdParsedSuccessfully)
                    {
                        Console.WriteLine("Product ID:" + productId);

                        int indexOfProductNameEnd = lineTrimmed.IndexOf("
");
                        string sectionStringEnd = lineTrimmed.Substring(in-
dexOfProductNameEnd, lineLength - indexOfProductNameEnd);
                        string[] sectionStringEndSplitBySpace = sectionStrin-
gEnd.Split(' ');

                        // Initializing an array of numbers, which are gath-
ered from string section end:
                        var sectionStringNumbers = new float[15];

                        int indexCounter = 0;
                        foreach(var str in sectionStringEndSplitBySpace){
                            var numberAsString = str.Trim();
                            if (numberAsString.Length != 0){
                                // Converting number as a string to float:
```

```
                                var number = float.Parse(numberAsString, Num-
berStyles.Any, CultureInfo.InvariantCulture);

                                // Assigning the converted number to an array
containing all numbers
                                // except product ID associated with the cur-
rent product:
                                sectionStringNumbers[indexCounter++] = num-
ber;
                            }
                        }

                        string productName = lineTrimmed.Substring(
                            Product.IdLength + 1, indexOfProductNameEnd -
(Product.IdLength + 1));

                        // Storing numbers gathered from the line into varia-
bles,
                        // which are then used in initialization of a product
object:
                        float productSaleUnitCampaign = sectionString-
Numbers[0];
                        float productSaleUnitTotal = sectionStringNumbers[1];
                        float productSaleCampaign = sectionStringNumbers[2];
                        float productSaleTotal = sectionStringNumbers[3];
                        float productPortionPercentAlar = sectionString-
Numbers[4];
                        float productPortionPercentCampaign = sectionString-
Numbers[5];
                        float productAleRow = sectionStringNumbers[6];
                        float productAleTotal = sectionStringNumbers[7];
                        float productProfitCampaign = sectionString-
Numbers[8];
                        float productProfitTotal = sectionStringNumbers[9];
                        float productProfitPortionAlar = sectionString-
Numbers[10];
                        float productProfitPortionNorm = sectionString-
Numbers[11];
                        float productProfitPercentCampaign = sectionString-
Numbers[12];
                        float productProfitPercentTotal = sectionString-
Numbers[13];
                        int productSelectionId = (int)sectionString-
Numbers[14];

                        var product = new Product(productName);
                        product.AleRow = productAleRow;
                        product.AleTotal = productAleTotal;
                        product.PortionSubgroup = productPortionPercentAlar;
                        product.PortionCampaign = productPortionPercentCam-
paign;
                        product.ProfitCampaign = productProfitCampaign;
                        product.ProfitTotal = productProfitTotal;
                        product.ProfitPercentCampaign = productProfitPer-
centCampaign;
                        product.ProfitPercentTotal = productProfitPercentTo-
tal;
                        product.ProfitPortionSubgroup = productProfitPortion-
Alar;
                        product.ProfitPortionNorm = productProfitPortionNorm;
```

```
                        product.SaleCampaign = productSaleCampaign;
                        product.SaleTotal = productSaleTotal;
                        product.SaleUnitCampaign = productSaleUnitCampaign;
                        product.SaleTotal = productSaleUnitTotal;
                        product.SelectionId = productSelectionId;

                        currentSegment.Products.Add(product);
                        Products.Add(productId, product);
                    }
                }
            }
        }
        public void Parse(){
            var lines = File.ReadAllLines(filename);
            foreach(var line in lines){
                ParseLine(line);
            }
        }
    }
}
```

Liite 3

PRINTOUT-LUOKKA – F#-TOTEUTUS

```
namespace PrintoutParsingFSharp

open System.Collections.Generic
open System.IO

type Printout (filename: string) =
    /// Member data
    let mutable currentSegment = new Segment(null)
    let filename = filename

    /// Properties
    member val Departments = new Dictionary<int, Department>() with get, set
    member val Products = new Dictionary<int, Product>() with get, set
    member val Segments = new Dictionary<int, Segment>() with get, set

    member private this.GetNumberFromNumberString(number: string) =
        if number.Length > 0
            then number |> float32
            else 0.0f

    /// Member functions (methods)
    member private this.ParseLine (line : string) =
        let lineTrimmed = line.Trim()

        // Match expression has been utilized here, even though
        // imperative if statement would work here also
        match lineTrimmed.Length > 0 with
        // If this line is not empty
        | true ->
            // If a line with department information was found
            if lineTrimmed.StartsWith(Department.SectionStartString) then
                let indexOfDepartmentNameBegin =
                    Department.SectionStartString.Length +
                    Department.NumberOfSpacesAfterSectStartString +
                    Department.IdLength + 1
                let  sectionStringEnd  =  lineTrimmed.Substring(indexOfDepart-
mentNameBegin, lineTrimmed.Length - indexOfDepartmentNameBegin)

                // Finding index of the first 4 spaces
                let indexOfDepartmentNameEnd = sectionStringEnd.IndexOf("   ")

                let departmentIdBegin =
                    Department.SectionStartString.Length + Department.Number-
OfSpacesAfterSectStartString
                let departmentIdEnd =
                    Department.SectionStartString.Length + Department.Number-
OfSpacesAfterSectStartString + Department.IdLength - departmentIdBegin

                let  departmentId  =  lineTrimmed.Substring(departmentIdBegin,
departmentIdEnd) |> int
                let  departmentName  =  sectionStringEnd.Substring(0,  indexOf-
DepartmentNameEnd)
```

```
                this.Departments.Add(departmentId, new Department(department-
Name))

            elif lineTrimmed.StartsWith(Segment.SectionStartString) then
                // Finding index of the first 4 spaces
                let indexOfSegmentNameEnd = lineTrimmed.IndexOf("    ");

                let segmentIdBegin = Segment.SectionStartString.Length + 1
                let segmentIdEnd = Segment.SectionStartString.Length + 1 + Seg-
ment.IdLength - segmentIdBegin
                let segmentId = lineTrimmed.Substring(segmentIdBegin, segmen-
tIdEnd) |> int;

                let segmentNameBegin = Segment.SectionStartString.Length + 2 +
Segment.IdLength
                let segmentNameEnd = indexOfSegmentNameEnd - (Segment.Section-
StartString.Length + 2 + Segment.IdLength)
                let segmentName = lineTrimmed.Substring(segmentNameBegin, seg-
mentNameEnd)

                // For each product of the current segment
                currentSegment.Products |> Seq.iter(fun product -> product.Se-
lectionId <- segmentId )

                this.Segments.Add(segmentId, new Segment(segmentName))
                currentSegment <- new Segment(null)

            elif lineTrimmed.Length >= Product.IdLength then
                let     couldParse,     productId     =     System.Int32.Try-
Parse(lineTrimmed.Substring(0, Product.IdLength))

                match couldParse with
                // If product ID was parsed successfully, then we are dealing
with a product line
                | true ->
                    let indexOfProductNameEnd = lineTrimmed.IndexOf("    ")
                    let sectionStringEnd =
                        lineTrimmed.Substring(indexOfProductNameEnd,
lineTrimmed.Length - indexOfProductNameEnd)
                    let    sectionStringEndSplitBySpace    =    sectionStrin-
gEnd.Split(' ')

                    // Initializing an array of numbers, which are gathered
from string section end
                    let sectionStringNumbers = sectionStringEndSplitBySpace |>
Seq.map(this.GetNumberFromNumberString) |> Seq.toArray

                    let productName = lineTrimmed.Substring(Product.IdLength +
1, indexOfProductNameEnd - (Product.IdLength + 1))

                    // Storing numbers gathered from the line into variables,
                    // which are then used in initialization of a product
object:
                    let productSaleUnitCampaign = sectionStringNumbers.[0]
                    let productSaleUnitTotal = sectionStringNumbers.[1]
                    let productSaleCampaign = sectionStringNumbers.[2]
                    let productSaleTotal = sectionStringNumbers.[3]
                    let productPortionPercentAlar = sectionStringNumbers.[4]
                    let   productPortionPercentCampaign   =   sectionString-
Numbers.[5]
```

```
                    let productAleRow = sectionStringNumbers.[6]
                    let productAleTotal = sectionStringNumbers.[7]
                    let productProfitCampaign = sectionStringNumbers.[8]
                    let productProfitTotal = sectionStringNumbers.[9]
                    let productProfitPortionAlar = sectionStringNumbers.[10]
                    let productProfitPortionNorm = sectionStringNumbers.[11]
                    let    productProfitPercentCampaign    =    sectionString-
Numbers.[12]
                    let productProfitPercentTotal = sectionStringNumbers.[13]
                    let productSelectionId = sectionStringNumbers.[14] |> int

                    let product = new Product(productName)
                    product.AleRow <- productAleRow
                    product.AleTotal <- productAleTotal
                    product.PortionSubgroup <- productPortionPercentAlar
                    product.PortionCampaign <- productPortionPercentCampaign
                    product.ProfitCampaign <- productProfitCampaign
                    product.ProfitTotal <- productProfitTotal
                    product.ProfitPercentCampaign <- productProfitPercentCam-
paign
                    product.ProfitPercentTotal <- productProfitPercentTotal
                    product.ProfitPortionSubgroup <- productProfitPortionAlar
                    product.ProfitPortionNorm <- productProfitPortionNorm
                    product.SaleCampaign <- productSaleCampaign
                    product.SaleTotal <- productSaleTotal
                    product.SaleUnitCampaign <- productSaleUnitCampaign
                    product.SaleTotal <- productSaleUnitTotal
                    product.SelectionId <- productSelectionId

                    currentSegment.Products.Add(product)
                    this.Products.Add(productId, product)
                // Otherwisely
                | false -> ()
        // If this line is empty
        | false -> ()

    member this.Parse () =
        let lines = File.ReadAllLines(filename)
        lines |> Seq.iter(fun line -> this.ParseLine(line)) |> ignore
```

Liite 4

C#-OHJELMA KÄYTTÄEN F#-KIRJASTOA TULOSTEEN JÄSENTELYYN

```
using System;

namespace PrintoutParsingExample{
    class Program{
        static void Main(string[] args){
            if(args.Length >= 1){
                var printout = new PrintoutParsingFSharp.Printout(args[0]);
                printout.Parse();

                // For each department in the printout parsed
                Console.WriteLine("Printing departments...");
                foreach (var department in printout.Departments){
                    Console.WriteLine("Department (ID: {0}): {1}", depart-
ment.Key, department.Value.Name);
                }
                Console.ReadKey();

                // For each segment in the printout parsed
                Console.WriteLine("Printing segments...");
                foreach (var segment in printout.Segments){
                    Console.WriteLine("Segment (ID: {0}): {1}", segment.Key,
segment.Value.Name);
                }
                Console.ReadKey();

                // For each product in the printout parsed
                Console.WriteLine("Printing products...");
                foreach (var product in printout.Products){
                    Console.WriteLine("Product (ID: {0}): {1}", product.Key,
product.Value.Name);
                }
                Console.ReadKey();
            }
            else{
                Console.Error.WriteLine("Report file was not given.");
                Console.ReadKey();
            }
        }
    }
}
```