Funktionell programmering

En jämförelse med imperativ programmering.

Åbo Akademi
Skribent: Henrik Söderberg
Handledare: Marina Waldén
2018

Innehållsförteckning

Abstrakt

I dagens läge när datamängden och förväntningarna för mjukvarukvalitet ökar är det mer och mer viktigt att program är välstrukturerade och pålitliga. Det är bevisligen möjligt att skriva pålitliga program med språk som baserar sig på andra än den funktionella paradigmen, i vissa fall är det även rekommenderat. Problemet är inte tekniskt, det är visserligen möjligt att skriva mjukvara som håller sig till höga standarder på diverse olika språk och metoder, problemet uppstår då program blir större och komplexa. Vi som människor har svårt att hålla reda på stora mängder information åt gången och behöver abstraktioner och struktur. Funktionellt skrivna program är ofta klara och flexibla, ofta klarare än motsvarande program skrivna med imperativa programmeringsspråk. Funktionella paradigmen erbjuder lösningar till problem som råddig kod, flexibilitet och att köra program parallellt.

1 Inledning

Inom mjukvaruproduktion används flera olika programmeringsspråk. Enligt Tiobe indexen som följer användningen av programmeringsspråk inom industrin, är de vanligaste språken för tillfället är Java, C/C++ och Python [3]. Dessa språk faller in i flera paradigm, nämligen objektorienterade och imperativa paradigmen. Det är vanligt för moderna språk att inte vara avgränsade till något specifikt paradigm utan använder sig av koncept från flera olika paradigm [4]. Så gott som alla språk, funktionella eller inte, koncept från funktionella programmeringsspråk i sig. Däremot har språken ofta något huvudsakligt paradigm som de i stort sett följer. Exempelvis Java är objektorienterad, att skriva program i java utan att använda sig av objekt är nästan omöjligt.

Namnet funktionell programmering härstammar från sättet program uppbyggs.
I jämförelse till imperativa program som är uppbyggda av en sekvens kommandon är funktionella program uppbyggda enbart med funktioner. Specifika skillnaden mellan kommandon, funktioner och dylikt som skiljer paradigmen redogörs senare i tredje kapitlet där funktionell programmering förklaras i detalj. Det finns ingen exakt definition på vad ett funktionellt språk egentligen är. Däremot finns det kännetecknande egenskaper. Fokusen i avhandlingen ligger på vissa egenskaper vanliga till funktionella språk och hur de skiljer sig till imperativa och objektorienterade metoder. Egenskaperna som tas fram i avhandlingen är definierade värden istället för variabel tilldelning, rena funktioner dvs. funktioner utan sidoeffekter, högre ordningens funktioner, lat evaluering och matchning av mönster.

Syftet med avhandlingen är att förklara i detalj vad funktionell programmering är och varför det är relevant. Varför skulle en programmerare bry sig om funktionell programmering och när kunde det vara bra att använda sig av koncept från funktionella paradigmen. Avhandlingen behandlar också kort paradigm i allmänhet för att vidga läsarens syn på programmeringsspråk. I avhandlingen antas det att läsaren är bekant med programmering och kan programmera i åtminstone ett programmeringsspråk på en grundläggande nivå. Exempelkod i avhandlingen är

skrivna i JavaScript med ES6 standarden och Haskell som är ett funktionellt språk [5] [10].

2 Programmerings paradigm

Programmeringsparadigm är ett sätt att klassificera språk efter tankesätt och koncept centrala till språken. De vanligaste programmeringsparadigmen enligt populära språk listade i Tiobe Indexet är imperativ programmering, objektorienterad programmering, funktionell programmering [3]. Många paradigm överlappar varandra. Exempelvis funktionell programmering är en delmängd av deklarativa programmeringsspråk, däremot är alla deklarativa språk inte funktionella. Skillnaderna mellan paradigm kan alltså vara minimala, däremot kan väldigt små olikheter vid implementering av program ha en stor inverkan på resultatet. I detta kapitlet introduceras imperativ programmering, deklarativ programmering och objektorienterad programmering. De vanligaste paradigmen introduceras för att lyfta fram vad som menas med funktionell programmering och största skillnaderna med olika typer av programmeringsspråk.

2.1 Imperativ programmering

Imperativ programmering baserar sig på att definiera kommandon som modifierar programmets tillstånd på något sätt. Dessa kommandon kallas på efter varandra. På grund av att imperativa språk har tilldelning dvs. variabler kan byta värde, är ordningen på kommandon väldigt viktig. Att kalla på kommando A och sedan B är inte ekvivalent med att kalla kommando B och sedan A [4]. Någonting kännetecknande för imperativa språk är hur program skrivs, de skrivs på ett sätt som förklarar i detalj hur någonting skall göras. Programmeraren definierar inte bara vad programmet göra utan också hur den ska göra det.

```
const timesTwo = (numbers) => {
  let ret = []
  for(let i = 0; i < numbers.length; i++){
          ret.push(numbers[i] * 2)
  }
  return ret
}
```

I programmet ovanför definieras en funktion ”timesTwo” som tar en lista av siffror som argument. För att iterera över elementen i listan måste programmeraren definiera hur iterationen sker med en loop och sedan definiera vad funktionen egentligen gör, i detta fall multiplicerar siffrorna i listan med två. Programmet måste också förändra på programtillståndet för att kunna iterera över listan. I en loop introduceras en variabel, i detta fall variabeln ***i***, vars tillstånd programmet måste förändra för att hämta element från listan.

2.2 Deklarativ programmering

Deklarativ programmering är ett paradigm där instruktionerna grundar sig på att beskriva vad programmet ska göra istället för att ge instruktioner hur programmet ska göra det [7]. Detta ses snabbt från ett exempel. Funktionen använder sig av en vanlig funktion inom funktionella språk, ‘map’ som itererar igenom alla element i en lista och applicerar en funktion på elementen varefter en modifierad lista med lika antal element returneras. I exemplet nedan multipliceras varje element med två.

```
const timesTwo = (numbers) => {
  return numbers.map(el => el*2)
}
```

I jämförelse till imperativa version visar exemplet tydligt hur programmeraren inte behöver bry sig om hur listan itereras utan bara om vad som ska göras. Deklarativ

programmering är en abstraktion. Funktionaliteten är definierad i förväg för att underlätta arbetet. Fördelen med att abstrahera implementeringen är att göra det lättare för programmerare att resonera över ett problem. När programmerare tittar på det imperativa exemplet måste hen fundera över två saker. Vad gör programmet, och hur den gör det. Programmeraren kan inte vara säker på att loopen fungerar som den ska, detta kan exempelvis leda till programfel i form av indexeringsfel. Däremot i deklarativa implementeringen är detta inte ett problem, iterationen är färdigt definierad och programmeraren behöver bara fundera på implementeringen av själva funktionaliteten.

Ett bekant exempel på ett deklarativt språk är Structured Query Language (SQL). I SQL definierar man vad man vill söka från sin databas, inte hur man gör det [16].

```
SELECT Name
FROM Users
WHERE Age > 18
```

Ett exempel på SQL kod. När koden körs hämtar programmet namnet för alla personer som är över 18 år.

2.3 Objektorienterad programmering

Ett objekt i programmering är en instans av en klass. Ett exempel för att förklara klass och objekt är jämförelse till ritningar och byggnader. Ritningen är klassen och byggnaden som byggs på basis av ritningen är objektet [6]. Objektorientering skiljer sig från vanliga dataabstraktion såsom C språkets “struct” med att implementera funktionalitet som är relaterat till klassen.
. En definition av objektorienterad programmering är att skriva så kallade klasser för data som objekt innehåller. Klasserna definierar funktionaliteten som objekt ska ha. I objektorienterad programmering är det också möjligt att ärva funktionalitet från andra klasser. Exempelvis kunde programmeraren definiera en klass ***cirkel*** som en subklass av klassen ***form***. Exemplet nedan visar hur ett sådant program implementeras i JavaScript. Cirkeln ***c*** som skapas kommer alltså att vara ett objekt av

klassen ***Cirkel***, som också implementerar funktionalitet från klassen ***Form***. När programmet exekveras kommer c.getType() evalueras till ”cirkel” och c.describe() evalueras till ”Röd cirkel” på grund av att det är namnet som ges som argument då cirkel objektet skapas.

```
class Form {
  constructor(name, type){
    this.name = name
    this.type = type
  }
  getType(){
    return this.type
  }
}

class Cirkel extends Form {
  constructor(name){
    super(name, 'cirkel')
  }
  describe(){
    return this.getType() + 'n är rund'
  }
}


let c = new Cirkel("Röd cirkel")
c.describe()
c.getType()
```

3 Funktionell programmering

Idén för implementering av funktionella programmeringsspråk började redan på 1950 talet då intresse för artificiell intelligens ökade. Behovet för ett nytt språk för artificiell intelligens härstammar från brist på effektiv behandling av listor. Det behövdes ett sätt att effektivt behandla information i dynamiska länkade listor istället för räckor. Första egentliga funktionella språket LISP (List Processing) skapades år 1958 av John McCarthy vid MIT [7]. Funktionella programmeringsspråk baserar sig på definiering och exekvering av funktioner istället för att modifiera programmets tillstånd.
I detta kapitlet redogörs egenskaper vanliga för funktionella språk och lambda kalkylen som ligger som teoretiska basen.

3.1 Lambda kalkyl

Lambda kalkylen utvecklades av Alonzo Church år 1936 på samma tid men oberoende av Alan Turing för att lösa avgörbarhetsproblemet [8]. Kalkylen är teoretiska grunden till funktionell programmering och är därmed relevant att introducera [12].
Kalkylen är definierad med λ (lambda) termer, generalisering av uttryck med variabler och reducering av uttryck. För att evaluera uttrycken anges variablerna värden. Lambda uttryck är abstraktioner, i programmering är uttrycken ekvivalenta med funktioner. Lambda kalkylen är numera också använt i flera språk utanför funktionella paradigmen för att definiera så kallade anonyma funktioner. Anonyma funktioner i programmering är ofta lambda uttryck som evalueras, i flera språk som till exempel C# och Python används även nyckelordet ”lambda” för att konstruera sådana funktioner [14] [15].

I Lambda kalkylen är det vanligt att inte namnge funktioner. För att underlätta definitionerna används notationen F = (funktion definition). Att applicera funktioner beskrivs då med F A, som betyder att funktionen F appliceras på A.
Kalkylen i sig själv är litet men är tillräckligt kraftig för att representera ett godtyckligt programmeringsspråk. Kalkylen kan med andra ord uttrycka alla

beräkningsbara funktioner, detta betyder att lambda kalkylen är ekvivalent med Turing maskinen [2].
Logiska sanningsvärdena definieras med ett val. Sant och falskt kan tänkas som två olika funktioner där två värden tas som input och ett värde väljs. I lambda kalkylen är sant och falsk definierat på följande sätt.

Falskt = (λxy.y)
Sant = (λxy.x)

Ett simpelt exempel för att förstå lambda notationen är identitetsfunktionen, i lambda kalkylen är identitetsfunktionen definierad på följande sätt.

funktion = **λ**<namn>.<funktion definition>
λx.x

Namnet efter λ-tecknet är funktionsargument, expressionen efter punkten är funktionens definition. Enligt notationen definieras identitetsfunktionen som I = (**λx.x**).

Evaluering av funktionen med substituering.

I y = (λx.x) y

x substitueras med argumentet y.

= **(λy.y)**

funktionen evalueras till y

= y

Identitetsfunktionen beter sig förväntat och evalueras till givna argumentet.

Kalkylen är oberoende av typer. Givet en funktion **F** är det då möjligt att applicera den på sig själv **(F F).** Detta möjliggör rekursion som är en väsentlig del av lambda kalkylen [8]. Kalkylen skulle inte vara tillräckligt kraftig att representera vilken som helst beräkningsbar funktion utan rekursion.
Rekursion i lambda kalkylen är definierat med funktionen definierad nedanför, vanligtvis betecknat **Y**.

$$\mathbf{Y = \lambda f.(\ \lambda x.f(x\ x))(\ \lambda x.f(x\ x))}$$

En funktion **e** applicerat på **Y** evalueras då på följande sätt.

$$\mathbf{(Y\ e) = (\lambda x.e(x\ x))\ (\lambda x.e(x\ x))}$$

Namnet **x** substitueras med funktionen **(λx.e(x x)).**

$$\mathbf{= e\ ((\lambda x.e(x\ x))\ (\ \lambda x.e(x\ x)))}$$

Funktionen som ges som argument till **e** är nu **Y**.

$$\mathbf{= e\ (Y\ e)}$$

Funktionen som ges som argument till funktionen **Y** kommer att returnera funktionen själv som kallar på funktionen **Y** med sig själv som parameter [12].

3.5 Pattern matching

Pattern matching är en egenskap kännetecknande till funktionella språk. Pattern matching betyder att programmet hittar mönster i till exempel funktionsdefinition och argument och använder sedan definitionen som matchar argument [2]. I Haskell är det möjligt att definiera flera implementeringar av samma funktion. Exempelvis en funktion som räknar ut fibonacci talserien skulle definieras på följande sätt.

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Programmet väljer vilken definition av funktionen som används på basis av argumenten som ges till funktionen. Imperativa versionen av samma funktion kan programmeras på följande sätt.

```
const fibonacci = (n) => {
  if(n == 0){ return 0 }
  if(n == 1){ return 1 }
  else {
    return fibonacci(n-1) + fibonacci(n-2)
```

```
 }
}
```

Att hitta mönster i argument är väldigt kraftfullt. Detta tillåter definieringen av funktioner som använder sig av mönstret (head:tail) för listor som förklaras i nästa kapitel. Haskell kan använda sig av mönster för nästan alla datatyper. Exempelvis en funktion som byter position på element i en tupel kunde definieras på följande sätt.

```
swap (a,b) = (b,a)
```

3.2 Oföränderlig data

En av de största skillnaderna mellan funktionella och imperativa språk är hantering av tillstånd. Som förklarat i kapitlet 3.1 Imperativ programmering, hanteras tillstånd med att tilldela värden till variabler och på detta sätt manipulera tillstånd. Rent funktionella språk har ingen variabeltilldelning på samma sätt som imperativa språk [9]. Variabler namnges och är sedan oföränderliga som betyder att deras värde inte kan förändras. Detta förverkligas av implementationen av programmeringsspråket, självklart är variablerna de facto i minnet någonstans, skillnaden är att språket förhindrar programmeraren att förändra värdet. Vanligtvis måste värdet förändras i något skede, exempelvis listor som är dynamiska måste ju vara föränderliga. I Haskell är problemet löst med använda gamla värdet till godo när en förändring vill göras. Istället för att förändra på en existerande variabel kommer programmet att skapa en ny lista med önskade förändringen [9]. Lösningen låter väldigt ineffektivt, att skapa nya variabler när man vill förändra ett värde låter som slöseri av minne. Hur detta inte är ett problem kan visas med ett exempel. Listor i Haskell är inte som räckor från andra språk. De är rekursivt definierade på följande sätt.

Första elementet i listan, kallas för huvud ”head” på engelska. Andra elementet är en lista på resten av elementen som kallas för svansen ”tail” på engelska [11]. I exemplen kommer engelska namnen head och tail användas för att detta är en konvention i programmering. Om svansen inte innehåller element representeras den som en tom lista.

head:tail

En lista [1,2,3] är byggt upp i Haskell som 1:2:3:[]. På detta sätt kan programmerare rekursivt iterera över listor med att definiera funktioner som använder sig av huvudet och svansen som input parametrar. Exempelvis ”sum” funktionen som summerar elementen i listan skulle definieras på följande sätt.

```
sum [] = 0
sum (head:tail) = head + sum tail
```

Då tar listan formen av 1:tail där tail = [2,3]. Då evalueras sum på följande sätt.

```
sum [1,2,3]   =   1 + sum [2,3]
              =   1 + 2 + sum [3]
              =   1 + 2 + 3 + sum []
              =   1 + 2 + 3 + 0
              =   6
```

Det är möjligt att skapa en ny lista med att använda sig av gamla listor. Om en lista **lista1 = [2,3,4]** är definierad kan element tilläggas med kommandot **1:lista1**, det betyder alltså att 1 är huvudet och **lista1** är svansen. Detta kommer däremot inte att förändra **lista1** för att listat är naturligtvis oföränderligt. Hur detta inte slösar minnet är en följd av oföränderlighet. På grund av att **lista1** inte kan förändras, kan programmet använda sig av den när den skapar en ny lista. Detta betyder alltså att när **lista2** skapas som **lista2** = **1:lista1** kommer programmet inte att allokera minne för hela listan, utan bara för förändringen. **Lista2** kommer att evalueras som siffran 2 med en svans som pekar till minnet där **lista1** befinner sig. Oföränderliga data är enda orsaken till att detta är möjligt. Om **lista1** inte var oföränderlig kunde programmerare modifiera **lista1** och därmed kunde den inte pålitligt användas i **lista2**.

Att namnge variabler i funktionella språk är jämförbart med likhetstecknet från matematiken.
Givet en ekvation **x = y + 1** kan värdet på **x** hittas om värdet för **y** är känd. Detta gäller också för värdet för y, om värdet på **x** är känd kan värdet för **y** hittas. Ekvationen **x = y + 1** betyder alltså inte att **x** blir tilldelad värdet av **y + 1** utan kan tänkas som ett påstående om att värdet på vänstra sidan av likhetstecknet är lika med värdet på högra sidan [11]. En programmerare kan alltså inte påstå någonting som **x = x + 1** för att det inte är korrekt. Om **x = 2** kommer programmet att evalueras som 2 = 2 + 1 som naturligtvis är fel.
Skillnaden mellan variabeltilldelning och att namnge variabler är lätt att demonstrera i Haskell. Ett program som definierar **x = 5** kommer evaluera **5 = x** korrekt på grund av att båda då kommer att evalueras till 5 = 5 som är korrekt, imperativa programmeringsspråk kan inte evaluera detta för att programmet kommer att försöka tilldela **5 = x** istället för att evaluera uttrycket.

Rich Hickey, skaparen av programmeringsspråket Clojure beskrev oföränderliga data under sin presentation vid konferensen JaxConf 2012 “The Value of Values” [13]. Han tog ställning till vad programmerare tänker på när de tänker på värden. Det är vanligt att tänka på tilldelning som värden så som man tänker på dem i matematiken. I imperativa programmeringsspråk är variabler ofta någonting annat. Implementering av variabeltilldelning är adressering. Datorn associerar någon variabel med en minnesadress dvs. x = 5 betyder då inte “x är 5” utan att x är associerat med någon minnesadress som innehåller något värde. Eftersom variabler i imperativa programmeringsspråk ofta inte är oföränderliga argumenterar Hickey att programmerare aldrig kan vara säkra på om värdet sparat i en variabel under exekvering av ett program är korrekt [13].

Användning av oföränderliga data istället för variabler som är föränderliga referenser till minnet har sina för- och nackdelar. Fördelarna är för det mesta simplifieringar för programmerare. Det är simpelt att resonera vad något värde är om den är oföränderlig som leder till lättläst kod. Exempelvis att behandla listor blir enkelt att resonera över om man inte förändrar och försöker följa med tillståndet på en existerande lista, utan alltid introducerar en ny lista med förändringarna. Att data är

oföränderligt är också bra för processer som använder sig av samma data i olika processer. Om ett program till exempel itererar över och använder värden i en lista i process 1 och på samma tid förändrar värden i samma lista i process 2, kommer det att introducera problem. I imperativa språk måste man då introducera strukturer som låser upp tillgången till data under exekvering för att förhindra problem. Om data är oföränderligt är det däremot inte ett problem. På grund av att programmeraren själv inte behöver fundera på små detaljer som loopar och samtidigt exekverande process, är det lättare för programmerare att skriva program som är säkrare, simpla att resonera över och lättare att upprätthålla.

Nackdelen med att använda sig av oföränderliga data är effektivitet. Som tidigare nämnt är minnesanvändningen optimerad så gott som möjligt men är ändå inte tillräckligt bra till alla applikationer. I program där optimering av minne och tillstånd är viktigt, exempelvis i spelprogrammering eller inbyggda system är det inte alltid möjligt att programmera i funktionella språk på grund av bristen av manuell minneshantering.

3.3 Funktioner

Funktioner spelar en stor roll i funktionella språk. Programmerare som har programmerat i imperativa och objektorienterade språk har en uppfattning om funktioner. Funktioner i programmeringsspråk är kod som är namngett och definierat, någonting som enkelt kan återanvändas. I funktionella språk kan allting anses vara funktioner. I detta kapitlet redogörs två egenskaper vanliga för funktioner i funktionella programmeringsspråk, så kallade rena funktioner och högre ordningens funktioner.

Renhet. I imperativa och objektorienterade språk är det vanligt att förändra programmets tillstånd med att modifiera någon variabel som befinner sig utanför funktionen som körs. I funktionella programmeringsspråk är det vanligare att funktioner inte får innehålla så kallade sidoeffekter, detta betyder att funktionen inte kan förändra programmets tillstånd med att förändra någon variabel som befinner sig

utanför funktionen. Sådana funktioner kallas vanligtvis rena (”pure” på engelska) funktioner [9]. En ren funktion är liknande till funktioner från matematik där returvärdet inte är beroende på programtillstånd och kommer alltid att returnera samma resultat med lika parameter. Rena funktioner har alltså två egenskaper. Lika input producerar alltid lika output och de har inga sidoeffekter.
Ett bra sätt att reda ut om en funktion är ren är att alltid kontrollera om funktionen returnerar något värde, om funktionen inte returnerar ett värde är funktionen antin1gen onödig eller orsakar sidoeffekt.

En oren funktion som förändrar tillstånd utanför sig själv.

```
const plus_amount = (amount) => {
  x = x + amount
}
```

En ren funktion returnerar ett värde och förändrar inte tillstånd.

```
const plus_amount = (x, amount) => {
  return x + amount
}
```

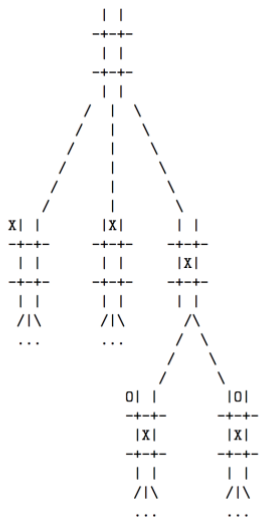
Programmering med rena funktioner är uppenbarligen någonting som kan göras i alla programmeringsspråk. Fördelen med att hålla funktionerna rena är att koden är läsbar och enkel att testa [9]. När en funktion är ren kan koden testas effektivare. När funktionen inte någonsin kommer att förändra programtillståndet och har samma output med lika variabler kan koden testas skilt oberoende av andra funktioner eller uttryck i programmet [1]. Om funktionen däremot inte är ren kan funktionens output påverkas av eller påverka programtillståndet, därmed är gjorda test inte lika pålitliga. Däremot behöver alla funktioner inte vara rena. Renhet är någonting som är väldigt användbart för att minimerar misstag men kan inte användas hela tiden. Exempelvis en funktion som generar ett slumpmässigt tal kan inte vara ren, funktionen har inte samma output med lika input.

Högre ordningens funktioner. En funktion som tar en annan funktion som argument eller returnerar en funktion kallas för en högre ordningens funktion. Alla andra funktioner är första ordningens funktioner [2]. Högre ordningens funktioner är numera implementerad i nästan alla av de allmänt använda programmeringsspråken. Högre ordningens funktioner är centrala till programmering i rent funktionella språk. Att kunna använda funktioner som argument i andra funktioner möjliggör flexibilitet i program [1]. På detta sätt kan små funktioner definieras och återanvändas. Utan högre ordningens funktioner skulle det vara omöjligt att skriva nyttiga program i funktionella programmeringsspråk.

3.4 Lat evaluering

Traditionellt när funktioner exekveras sparas resultatet från någon beräkning i minnet och används sedan vid behov. I funktionella språk kallas evaluering av uttryck lata på grund av att exekvering sker bara då det behövs. Exempelvis ett program som kallar på (**F Y)** F efter Y. Traditionellt om funktionen **Y** är definierad tidigare, som funktioner ofta måste vara för att kunna användas, sparas värdet i minnet och används sedan i exekveringen. Funktionella språk kommer att exekvera beräkningen av funktionen **Y** bara då funktionen **F** exekveras. Funktioner evalueras alltså vid behov [1]. Till exempel en simpel lista som innehåller alla naturliga tal [1,2,3…] kan med lat evaluering definieras utan att använda så mycket minne. I Haskell byggs ett sådant med definitionen **lista = [1..]**. Detta kommer att spara en oändlig lista i variabeln **lista**, som betyder att listan inte sparas i minnet utan evalueras då programmet behöver någonting från den.
Fördelen med lat exekvering är att inte behöva använda minnet i onödan och att behålla flexibilitet i program.

Ett exempel av en algoritm där fördelen med lathet är uppenbart är alfa-beta heuristik algoritmen för att estimera hur bra position en spelare har. Algoritmen använder sig av en trädstruktur som beskriver nuvarande positionen och alla möjliga steg i noder. Noden som programmet befinner sig är nuvarande positionen i spelet, och nodens löv beskriver de möjliga positionerna som kan nås från den nuvarande positionen.



Exempel på trädet i luffarschack [1].

Det blir klart att algoritmen är problematisk att implementera traditionellt. I ett spel som luffarschack där spelare försöker få vanligtvis tre kryss eller ringar i rad, är det möjligt att konstruera ett träd med att räkna ut alla möjliga positioner. Däremot om spelet blir större, som exempelvis schack, är det närapå omöjligt att räkna ut alla möjliga positioner. Det är därmed nödvändigt att bara räkna ut ett visst antal positioner framåt.

Traditionellt skulle programmet implementeras i en stor funktion på grund av att programmet inte troligtvis kan hålla tillräckligt stora mängder information i minnet för att representera N antal lager av trädet. Lathet möjliggör att funktionerna nödvändiga till algoritmen kan implementeras skilt och evalueras då programmet behöver dem. Därmed är funktionella programmet som använder lathet mycket mera flexibelt. Vid behov av förändringar behöver inte programmerare behandla programmet som ett stort program utan de kan göra förändringar i enskilda funktionerna som bygger upp algoritmen [1].

6 Slutsats

Syftet med avhandlingen var att undersöka egenskaper i funktionell programmering, egenskapernas inverkan i programmering, och jämföra det med imperativ programmering. Funktionell programmering är en abstraktion och ska behandlas som sådan. Att programmera på ett sätt som förhindrar att göra vissa operationer så som mutation av variabler och användning av rena funktioner kan tänkas vara ineffektivt. Att skriva bra mjukvarusystem som går att upprätthålla är väldigt svårt. Det finns ingen metod som magiskt fixar detta problem, på detta vis är funktionell programmering inget undantag. Idéerna har dock värde i mjukvaruproduktion, som tidigare kapitlen har tagit fram kan oföränderliga data, rena funktioner, högre ordningens funktioner komma med stora förenklingar för att skapa flexibel, läsbar kod som effektivt går att testa. Däremot är det inte ett krav att använda sig av ett rent funktionellt språk.

Personligen tycker jag att funktionell programmering är någonting som inte behövs göra i ett strikt funktionellt språk. En stor del av de mest använda programmeringsspråken stöder en funktionell programmeringsstil. Exempelvis Javascript ES6 har inbyggda funktioner vanliga för funktionella språk som map, filter och reduce. Andra koncept som oföränderlig data finns det stöd för i form av färdiga programbibliotek som Redux, Underscore och Immutable.js. Resten faller på programmeraren, att till exempel hålla sig till rena funktioner är någonting som programmeraren är ansvarig för. Funktionell programmering har nämnda fördelar och användning för vissa problem och skall behandlas som ett verktyg som kan vara väldigt effektivt.

[1] Why functional programming matters - John Hughes, Chalmers Tekniska Högskola
http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf

[2] Programming Languages: Principles and Paradigms
Maurizio Gabbrielli, Simone Martini
https://www.springer.com/la/book/9781848829138

[3] Tiobe Index, popularitet av programmeringsspråk - Hämtat: Februari 2018
https://www.tiobe.com/tiobe-index/

[4] Major programming paradigms
Gary T. Leavens, University of Central Florida
http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html

[5] Standard ECMA-262 6th edition
ECMAScript 2015 Language Specification
http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

[6] Lesson: Object-Oriented Programming Concepts
Oracle Java Documentation
https://docs.oracle.com/javase/tutorial/java/concepts/

[7] Concepts of programming langauges, Tenth Edition
Robert W. Sebesta
https://cs444pnu1.files.wordpress.com/2014/02/concepts-of-programming-languages-10th-sebesta.pdf

[8] Introduction to lambda calculus
Henk Barendregt, Erik Barendsen
https://pdfs.semanticscholar.org/322c/b8734965d5e7693248919f5d887e019f05b2.pdf

[9] Haskell wiki, Functional programming
Hämtat 27.02.2018
https://wiki.haskell.org/Functional_programming

[10] Haskell
Hämtat 23.3.2018
https://www.haskell.org/

[11] Learn You a Haskell for Great Good!
Miran Lipovača
http://learnyouahaskell.com

[12] Concepts, Evolution, and Application of Functional Programming Languages
Paul Hudak
http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf

[13] The Value of Values - Tal från JaxConf 2012
Rich Hickey
https://www.youtube.com/watch?v=-6BsiVyC1kM

[14] Lambda expressions (C# Programming Guide)
Microsoft
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressionsoperators/lambda-expressions

[15] Python Documentation – More Control Flow Tools
Docs Python3
https://docs.python.org/3/tutorial/controlflow.html

[16] Programming Paradigms for Dummies: What Every Programmer Should Know
Peter Van Roy - Professor of Computing Science and Engineering, Université catholique de Louvain
https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf