# Programmeringsparadigm - en jämförelse

Henrik Bäärnhielm
d98-hba@d.kth.se

Maj 1999

**Sammanfattning**

I denna uppsats beskrivs och jämförs två programmeringsparadigm, imperativ programmering och funktionell programmering. Först beskrivs varje paradigm för sig, dess egenheter, fördelar och nackdelar, med belysande exempel. Därefter jämförs de två paradigmen ur olika aspekter, som abstraktionsnivå och modularisering. Slutligen beskrivs två tillämpningar där respektive paradigm lämpar sig bäst, och det motiveras varför i båda fallen.

# Innehåll



# 1 Inledning

I denna uppsats skall jag beskriva och jämföra två *programmeringsparadigm*, alltså två fundamentalt olika sätt att se på och tänka vid programmering. De två paradigm jag har valt är det *imperativa*, som är det vanligaste, representerat av de stora programmeringsspråken, bl.a. C och Ada, och det *funktionella*, där språk som Haskell och LISP-dialekten Scheme tillhör de mer kända.

Först kommer jag att beskriva det ena paradigmet, dess egenheter, för- och nackdelar, och sedan motsvarande för det andra. Jag kommer att ta med exempel från riktiga programmeringsspråk som belyser paradigmens egenskaper där jag så finner lämpligt. Jag börjar med att ta upp det imperativa paradigmet, för jag tror det är mest känt bland läsarna.

Den andra delen i uppsatsen består av en jämförelse mellan de två paradigmen. Jag kommer att beskriva två tillämpningar. En där det imperativa paradigmet passar bäst och en där det funktionella är mest naturligt att använda. Jag kommer att argumentera för varför varje paradigm passar bäst till respektive tillämpning.

Den sista delen av uppsatsen innehåller slutsatser om när de två paradigmen passar bäst samt en diskussion av om något av de två kan anses som bättre än det andra rent allmänt.

# 2 Det imperativa paradigmet

Det imperativa paradigmet är det som de flesta stora programmeringsspråken följer. Grunden för den imperativa modellen för uträkning och programmering är Turingmaskinen, som också är grundläggande för alla vanliga datorers arkitektur. Turingmaskinen är en s.k. beräkningsmodell, uppkallad efter matematikern Alan Turing som utvecklade den. Något förenklat bygger den på att det finns ett oändligt stort "uppdaterbart lager", som i en riktig dator är internminnet. I en verklig dator är förstås minnet inte oändligt stort, men det är en god approximation, eftersom man i princip alltid kan få mer minne genom att sätta in fler komponenter i datorn.

## 2.1 Variabler

Att det imperativa paradigmet bygger på Turingmaskinen speglas direkt i det faktum att det mest karakteristiska för paradigmet är användandet av variabler, som motsvarar det "uppdaterbara lagret". Variablerna har två uppgifter, dels markera ett tillstånd för något och dels vara temporära lagringsplatser under en uträkning. Att markera tillstånd är den viktigaste uppgiften som brukar utföras genom s.k. "globala" variabler, medan den temporära lagringen egentligen inte är nödvändig vid god programmering i ett bra konstruerat programmeringsspråk.

Anledningen till att det imperativa paradigmet kallas just imperativt är en annan av dess karakteristiska egenskaper, nämligen att imperativ programmering i grunden består av kommandon som uppdaterar variabler. Den mest grundläggande formen av kommandon är tilldelningar av värden till variabler, och just att programmen i grunden består av variabeltilldelningar är en av nyckelegenskaperna för det imperativa paradigmet. Att variablerna kan tilldelas värden är en följd av att de enligt ovan är uppdaterbara.

## 2.2 Kommandon och uttryck

Ett annat sätt att uttrycka vad som händer är att programmen består av kommandon, vars effekt är att variabler ändrar värde. Detta för att vi lättare skall kunna se skillnaden mot funktionell programmering som vi beskriver senare. Man skiljer på *kommandon* och *uttryck*, där uttryck kan beräknas och ge ett värde, som sedan kan tilldelas en variabel, medan kommandon utför något, som i princip alltid innefattar att variablers värden ändras.

Begreppet procedur eller *subrutin* finns i alla betydande imperativa programmeringsspråk, även om det kan ha andra namn. Man kan även få s.k. funktioner, som är grunden för det funktionella paradigmet, genom att låta procedurerna returnera värden. I exempelvis C kallas alla procedurer för funktioner oavsett om de returnerar värden eller ej, medan man i Pascal skiljer på procedurer, som är rena kommandon, och funktioner som returnerar ett värde och därmed är uttryck.

Det är nu man verkligen kan tala om att kommandona, i det här fallet procedurerna eller funktionerna, har en viss effekt, eller snarare sidoeffekt. En funktion i sin rätta bemärkelse skall nämligen inte göra något annat än att beräkna ett värde och returnera det. De imperativa funktionerna kan mycket väl utföra andra saker också, som att ändra någon global variabel eller ändra någon av sina parametrar. Procedurer som inte är funktioner måste arbeta så om de skall uträtta något bestående alls. Detta kallas att funktionerna har *sidoeffekter*.

## 2.3 Flöde

Karakteristiskt för det imperativa paradigmet är också att programmen utgörs av ett flöde av kommandon som skall utföras i en viss ordning. Det har alltså i de flesta fall betydelse i vilken ordning kommandona exekveras. De två C-programmen i figur 1 ger inte samma effekt. Att kodradernas ordningsföljd har betydelse i imperativa programmeringsspråk hänger samman med att de saknar

s.k. *begreppsmässig genomskinlighet* (referential transparency)[1]. Vi definierar detta begreppet mer i detalj i avsnitt 4.3.

```
/* Program 1 */
#include <stdio.h>

int main()
{
    printf("Hello ");
    printf("World!\n");
    return 0;
}


/* Program 2 */
#include <stdio.h>

int main()
{
    printf("World!\n");
    printf("Hello ");
    return 0;
}
```

Figur 1: Kodradernas ordning har betydelse i imperativt paradigm.

### 2.4 För- och nackdelar

Det imperativa paradigmets förhållandevis låga abstraktionsnivå medför både fördelar och nackdelar. En fördel är att programmen blir snabba, eftersom de är förhållandevis maskinnära. Det är lätt att översätta dem till maskinkod utan att behöva lägga till så mycket kod som gör det hela långsamt. En nackdel är att man inte på ett enkelt sätt kan göra avancerade konstruktioner och datastrukturer, vilket gör att det kan bli svårt att göra riktigt avancerade tillämpningar. Det går för det mesta, men det blir mycket kod. Att koden blir omfattande är karakteristiskt för imperativa program, och det gör att de blir svåra att överblicka.

En annan fördel hos det imperativa paradigmet är det faktum att variablerna markerar ett tillstånd för något. Program skapas ofta för att simulera processer i verkligheten eller för att manipulera objekt i verkligheten, och sådana processer eller objekt har ofta någon form av tillstånd som varierar, med tiden e.d. Att använda variabler är ett naturligt sätt att hantera dessa objekt, och därför simulerar imperativa program på ett naturligt sätt sådana processer i verkligheten. Att man kan ändra variablernas värde, innebär att man får ett naturligt sätt att ändra objektens tillstånd när något händer i processen, så ur denna synvinkel är det imperativa paradigmet med rätta det största och mest använda av paradigmen.

## 3 Det funktionella paradigmet

Det funktionella programmeringsparadigmet innebär ett helt annat sätt att se på programmering. Som tidigare nämnts bygger det imperativa paradigmet på Turingmaskinen; det funktionella

[1] Översättning föreslagen av John Hughes från Chalmers tekniska högskola.

paradigmet bygger också på en beräkningsmodell, Lambdakalkylen, utvecklad ungefär samtidigt av matematikern Alonzo Church. De två modellerna är ekvivalenta enligt Church-Turings tes: alla program skrivna i det ena paradigmet har en motsvarighet i det andra som gör samma sak, men det funktionella paradigmet och Lambdakalkylen manar till ett helt annat tänkesätt.

## 3.1 Funktioner

I funktionell programmering är alla program funktioner och alla program består bara av funktioner. Med ”funktion” menas den matematiskt definierade funktionen, och därför tillåter man inte några *sidoeffekter* i funktionell programmering. Alla funktioner hämtar sina invärden och beräknar sina utvärden, och de får inte ändra sina argument eller utföra något annat bestående därutöver. Något som motsvarar procedurer finns inte i funktionell programmering, utan alla funktioner måste ge ett returvärde. Variabler kan existera, men markerar inte tillstånd utan kan bara vara mellanlagring i en uträkning.

Med tidigare vokabulär existerar inga kommandon i funktionell programmering, utan allt är uttryck som beräknas, ger värden åt variabler i andra uttryck, eller blir parametrar i funktionsanrop. Kärnan i funktionell programmering är därför *uträkning*. Alla programsatser är uttryck som kan beräknas och ge ett värde, vilket ytterst gäller programmet självt. Programmet fungerar precis som en funktion i matematisk mening, med definitionsmängd och värdemängd.

## 3.2 Funktioner av högre ordning

Den funktionella programmeringens två viktigaste koncept är *funktioner som första klassens objekt*, (functions as first-class objects), och *slö uträkning* (lazy evaluation). Att funktioner är första klassens objekt innebär att de kan ses som data. I imperativ programmering är det en klar skillnad mellan data och kod, men om funktionerna är första klassens objekt blir den skillnaden mycket mindre. Funktioner kan ges som argument till andra funktioner, precis som vilka värden som helst, och funktioner kan dynamiskt skapa nya funktioner och returnera dem. Funktioner som tar andra funktioner som argument eller returnerar funktioner kallas *funktioner av högre ordning*, (higher-order functions). Detta maskineri ger stora uttrycksmöjligheter som inte finns i det imperativa paradigmet, och det är ett av orsakerna till styrkan i det funktionella. Man kan lättare göra högre abstraktioner, eftersom man kan koda mer generella funktioner. I figur 2 visas ett exempel på detta i Scheme. Det kan tyckas att man kan få samma resultat i exempelvis C genom att använda pekare eller dylikt, och det är sant för enkla exempel på funktioner av högre ordning, men när man ska göra mr avancerade konstruktioner, där fler indata är okända, går det inte längre. I funktionell programmering är det också just själva funktionen i sig man ger som argument och inte någon form av pekare.

```
(define (apply-binary-operator operator arg1 arg2)
    (operator arg1 arg2))

(define (addition num1 num2)
    (apply-binary-operator + num1 num2))

(define (multiplication num1 num2)
    (apply-binary-operator * num1 num2))
```

Figur 2: Funktion av högre ordning

## 3.3 Slö uträkning

*Slö uträkning* är en mekanism för uträkning av argumenten till en funktion. I imperativ programmering är det oftast så, att när en funktion skall anropas och parametrarna inte är *atomära*, utan är sammansatta uttryck, som måste räknas ut, så räknas samtliga parametrar ut och därefter anropas funktionen med dessa värden. Detta kallas *ivrig uträkning*, (eager evaluation), och tillämpas även i vissa halvt funktionella språk som Scheme. "Slö uträkning" innebär istället, att bara så mycket av argumenten som behövs räknas ut, och de räknas ut först när funktionen behöver dem. Om ett argument inte används i funktionen, räknas det inte ut, och en del tid sparas. Intressantare blir det också om man tillåter att delvis icke uträknade uttryck kan ingå i större sammansatta uttryck. Man kan då ha exempelvis oändliga listor som argument till funktioner, eftersom bara så mycket som funktionerna behöver kommer att räknas ut av dem, och funktionen dessutom kan returnera den delvis icke uträknade listan.

Bruket av slö uträkning är en av orsakerna till att man i funktionell programmering inte tillåter sidoeffekter hos funktionerna. Att ha sidoeffekter tillsammans med slö uträkning leder lätt till kaos, vilket beror på att man inte kan förutsäga när saker och ting räknas ut, och att man därför inte vet när sidoeffekterna kommer, eller exakt i vilken ordning. Den totala kontrollen finns inte, vilket gör att slö uträkning inte passar in i imperativ programmering.

## 3.4 För- och nackdelar

Det funktionella programmeringparadigmets viktigaste fördel är att det uppmanar till god programmeringsteknik. Paradigmet gör det möjligt att bryta ner programmen i mindre funktioner, och det är t.o.m. så att det blir krångligt att skriva lite större funktioner, varför man tenderar att verkligen göra små funktioner och modularisera mera, vilket naturligtvis är en god sak. Hög abstraktionsnivå anses vanligtvis också vara en fördel, och funktionella språk har en förhållandevis hög abstraktionsnivå, vilket ger en möjlighet att mer inrikta sig på själva problemet istället för på diverse kringarbete.

Den högre abstraktionsnivån gör också att program skrivna i funktionella språk är kortare än motsvarande i t.ex. imperativa språk. Det går ofta fortare att skriva ett visst program i ett funktionellt språk, och det är lättare att förstå, eftersom programkoden uttrycker precis det som ska hända på ett kort och koncist sätt, utan en massa annan kod inbakad som styr programflödet, vilket ofta finns i ett mindre uttrycksfullt imperativt språk. I figur 3 visas ett program i Scheme som beräknar längden av en lista. Programmet är en eller två rader långt, beroende på hur man räknar, och är nästan en avskrift av den rekursiva definitionen på längden av en lista. Motsvarande program i t.ex. C skulle bli längre och svårare att tyda.

```
(define (list-length lista)
  (if (null? lista) 0 (1+ (list-length (cdr lista)))))
```

Figur 3: Kort funktionellt program

Nackdelarna med funktionell programmering är först och främst att exekveringen inte går så fort. Hög abstraktionsnivå är ett tveeggat vapen, och även de andra finesserna, som slö uträkning, gör det svårt att få programmen att gå riktigt fort, jämfört med "handkodade" imperativa program. En annan nackdel är att inmatning och utmatning, antingen det är till eller från en användare eller till någon fil eller annan enhet, är svår att få in i det funktionella paradigmet på ett snyggt sätt. Det är svårt att koda dessa funktioner utan sidoeffekter. I vissa funktionella språk har man löst det genom att denna del av språket tillåts ha icke-funktionella inslag men ett strikt funktionellt gränssnitt mot resten av språket. I andra språk, bl.a. Haskell, har man använt s.k. monader, som vi inte går närmare in på, för att kunna hantera in- och utmatning på ett rent funktionellt sätt.

Men inget av dessa sätt är helt utan brister, och funktionella språk har ofta brister i just in- och utmatningsdelen.

# 4 Jämförelse

Vi har nu beskrivit imperativ och funktionell programmering. I det följande avsnittet skall vi göra en jämförelse mellan dem. Sådana har redan förekommit här och var i texten men vi skall utvidga resonemangen här.

## 4.1 Variabler

Först och främst ligger det en skillnad i hur variablerna används. I imperativ programmering markerar variablerna tillstånd. Därför måste variablerna gå att ändra, vilket är karakteristiskt för det imperativa paradigmet. Det är egentligen detta, och inte själva existensen av variabler, som är det viktiga. I funktionell programmering existerar också variabler, men liksom i matematiken kan deras värden inte ändras sedan de fått ett värde. Detta är en följd av det inte finns några tilldelningssatser i funktionell programmering, till skillnad från imperativ programmering där just tilldelningen är den mest fundamentala instruktionen. Variablerna markerar inte heller något egentligt tillstånd som i imperativ programmering. Globala variabler används inte, utan alla variabler är lokalt knutna till någon funktion. Detta är förstås en följd av att alla program *är* funktioner och inget mer. Någon plats för variabler utanför funktionerna finns inte.

## 4.2 Deklarativ kontra imperativ programmering

En annan mer fundamental skillnad mellan paradigmen är att funktionell programmering är mer eller mindre deklarativ medan det imperativa paradigmet förstås är imperativt. I imperativa språk beskriver man *hur* något ska göras, medan man i deklarativa språk beskriver *vad* som ska göras, och sen låter man i princip datorn sköta resten. Det säger sig självt att deklarativa språk har en högre abstraktionsnivå jämfört med imperativa språk, där man ger rätt detaljerade beskrivningar av hur data skall flyttas omkring i minnet, via tilldelningar och iterationer, medan det i funktionella språk är annorlunda. Där ger man definitioner av funktioner, men det finns inte någon detaljerad beskrivning av hur det hela skall utföras. Det är denna jämförelse som visar den generella skillnaden i abstraktionsnivå mellan paradigmen.

När vi beskrev imperativ programmering talade vi om flöden, uttryck och kommandon. Imperativa program är som ett flöde av kommandon som beskriver vad som händer. Funktionella program istället funktioner som kan räknas ut. Kommandon existerar inte. Denna jämförelse visar den kanske viktigaste skillnaden mellan paradigmen, eftersom den inbegriper två grundbegrepp i de båda paradigmen.

## 4.3 Begreppsmässig genomskinlighet

*Begreppsmässig genomskinlighet* definieras som att man kan byta ut alla variabler mot sina värden i programkoden, och programmet skall ändå ge samma resultat. Variablerna är alltså på det sättet ”genomskinliga” i avseende på vad de refererar till. Denna egenskap saknas hos imperativa språk och halvt funktionella språk, men finns i äkta funktionella språk. Att detta kan vara fallet hänger samman med att funktionerna inte kan ha några sidoeffekter. De beräknar bara sitt värde, och kan beräkna det när som helst, så i vilken ordning funktionerna körs i ett program spelar ingen roll, det ger alltid samma resultat. Detta innebär att man även kan byta variablerna mot sina värden och få samma resultat, och alltså har vi begreppsmässig genomskinlighet. Detta har stor betydelse, vilket vi nu skall se.

Eftersom funktionell program är begreppsmässigt genomskinliga, och ordningen därmed är godtycklig, kan man inte göra iterationer, vilka i imperativ programmering är otroligt vanliga, ty dessa kräver att kodraderna har en definierad inbördes ordning. Istället är rekursion mycket vanlig och viktig, vilket förstås inte är någon nackdel, eftersom rekursion är ett så viktigt begrepp i datalogin och så många företeelser kan beskrivas rekursivt. Snarare är det en fördel, eftersom det kan vara lättare att bevisa att funktionen är korrekt. Det gör också att rekursivt definierade datatyper, som listor och träd, ofta finns inbyggda i de funktionella språken, vilket sällan är fallet i imperativa språk. En annan anledning till detta är att funktionella språk tvingas ha automatisk minneshantering, eftersom man annars måste ha tillgängliga funktioner som bl.a. avallokerar minne, och dessa är mycket svåra att koda utan sidoeffekter. Det är alltså inte möjligt att på ett enkelt sätt hantera dynamiska datatyper om de inte finns inbyggda i språket, eftersom man då skulle behöva hantera någon form av pekare. I imperativa språk har man inte det problemet, för där sköter programmeraren just minneshanteringen själv, vilket ofta är bra i imperativ programmering, eftersom man får mer kontroll över vad som händer och kan arbeta maskinnära. Sidoeffekter är då inte heller något hinder, eftersom de ju är tillåtna i imperativ programmering. Denna jämförelse visar på de praktiska skillnader som finns mellan imperativ och funktionell programmering. I funktionella språk finns ofta många avancerade datatyper inbyggda, och man använder ofta rekursion, medan imperativa språk sällan har mer än de rent atomära datatyperna och har iteration som den viktigaste kontrollstrukturen.

## 4.4 Modularisering

Den jämförelse som brukar lyftas fram, när man vill visa hur bra det funktionella paradigmet är, rör modularisering. Man kan göra en analogi med strukturell programmering jämfört med s.k. spaghettiprogrammering. Strukturell programmering innehåller bl.a. inga ”goto”-satser, och man skulle kunna hävda att man bara tagit bort några nyttiga verktyg från spaghettiprogrammeringen. Dock är det snarare så att strukturell programmering ger bättre möjligheter till modularisering av programmen, och detta är av godo, eftersom det ger mindre fel. En liknande jämförelse kan göras mellan imperativ och funktionell programmering. Den senare ger möjlighet, och tvingar en nästan till, att modularisera programmet, göra funktionerna korta och dela upp dem i mindre och mer generella funktioner.

De två koncept, som gör funktionell programmering särskilt lämpad att bygga upp större funktioner från mindre och mer generella sådana, är just de tidigare beskrivna *funktioner av högre ordning* och *slö uträkning*. Dessa markerar också en stor skillnad mellan paradigmen. Funktioner av högre ordning gör att skillnade mellan data och kod delvis suddas ut, medan det i imperativ programmering alltid funnits en klart definierad skillnad mellan data och kod, vilket förstås beror på att imperativ programmering ligger närmare maskinkoden. Det man vinner på att inte skilja mellan data och kod är att man får helt nya uttrycksmöjligheter och lättare kan koda mer avancerade konstruktioner.

Slö uträkning är också något man har svårt att tänka sig i det imperativa paradigmet, för det är av en så mycket högre abstraktionsnivå än vad som annars finns i detta paradigm. Styrkan i slö uträkning är att man kan modularisera *kontroll* från *uträkning*. Man kan ha en funktion som räknar ut alla värden och lägger in dem i en lista. Sedan kan man ha en annan funktion som gör själva kontrollen och tar en uträkningsfunktion som argument. När kontrollfunktionen behöver data aktiveras uträkningsfunktionen, eftersom den är ett argument. Det innebär att den körs, men bara så mycket som behövs, d.v.s det är fråga om slö uträkning. Sen kan man lätt byta ut kontroll- och/eller uträkningsfunktionen och få ett annat resultat. Det är som sagt modularisering som talar för det funktionella paradigmet.

Nu ska vi då till slut beskriva två tillämpningar, den ena där imperativ programmering passar bäst och den andra där funktionell programmering passar bäst, och vi ska motivera varför i varje särskilt fall.

## 4.5 Tillämpning 1: Actionspel

Den första tillämpningen är ett actionspel, ett s.k. *first-person-3D-shoot'em-up*, av samma typ som de välkända Wolfenstein 3D, Doom, Quake m.m. Spelet skall vara utrustat med det senaste i 3D-grafikväg och måste vara snabbt. Jag tror att en utförligare beskrivning är onödig eftersom konceptet är så välkänt.

I denna tillämpning är det imperativa paradigmet ett naturligt val. Detta p.g.a. kravet på låg abstraktionsnivå som är en direkt följd av den snabbhet som behövs. Spelet kommer ligga på gränsen till vad de senaste persondatorerna klarar av, vilket alla spel i samma genre gör. Man måste därför pressa ut all prestanda som är möjlig, och allt som gör att det går långsamt måste slopas. Det funktionella paradigmet är långsamt, och i ett så tidskritiskt program, som vi nu har att göra med, hjälper inte några funktioner av högre ordning, om det går långsammare än vad som överhuvudtaget är möjligt med den hårdvara man har. En låg abstraktionsnivå är nödvändig för att det skall gå tillräckligt snabbt, och alltså måste man välja det imperativa paradigmet. Inte sällan behöver man t.o.m. använda någon form av assemblerspråk.

Man kanske tycker att det vore möjligt att använda det imperativa paradigmet enbart till de tidskritiska delarna av programmet, som grafikrutinerna o.d., men funktionella språk har i allmänhet ett dåligt gränsnitt mot andra språk, så det skulle inte heller fungera.

Det är dock inte bara snabbheten som talar för det imperativa paradigmet. Själva koden för ett dylikt spel behöver egentligen inte vara så tekniskt avancerad, vad gäller avancerade datatyper och andra konstruktioner. Istället är det häftig grafik och dylikt som är det viktigaste. Det funktionella paradigmets avancerade konstruktioner kommer då inte till så stor nytta, utan gör som sagt det hela långsammare istället. Dessutom är ett dylikt actionspel lämpligt att göra i en objektorienterad form, och objektorientering ingår i det imperativa paradigmet, åtminstone till en del, eftersom man bl.a. har variabler som markerar tillstånd hos objekt. Att det imperativa paradigmet på ett naturligt sätt simulerar objekt med sina variabler återkommer alltså nu. Det funktionella paradigmet kan också simulera tillstånd, men inte på ett lika naturligt sätt.

## 4.6 Tillämpning 2: Luffarschack

Den andra tillämpningen jag skall beskriva är också ett spel, nämligen luffarschack. Det intressanta här är att man skall spela mot datorn, och därför behövs AI-rutiner. Funktionell programmering visar sig vara mycket lämplig för AI-rutiner, p.g.a. slö uträkning och funktioner av högre ordning. Det vi ska beskriva är egentligen inte specifikt för just luffarschack, utan är rätt generella AI-rutiner för spel, men jag har valt luffarschack som specifik tillämpning.

Datorn måste kunna räkna ut hur bra dess ställning i spelet är, och för detta ändamål finns en algoritm som kallas Alfa-Beta. Den fungerar som följer.

Man bygger upp ett träd med alla möjliga spelomgångar, där barnen till varje nod är de möjliga positionerna efter det drag i spelet som representeras av bågen mellan noden och barnet. Detta träd kan vara oändligt, vilket det kan vara i luffarschack, och funktionen som bygger trädet behöver inte ha ett egentligt slut, vilket vi skall se inte är något problem. Vi kan dock inte räkna oss igenom ett oändligt träd när vi ska bestämma positionens värde, utan vi ska bara räkna några drag framåt i spelet, alltså en del av trädet. Det behövs även ett "godhetstal" i varje position så att man har något att jämföra. Godhetstalet är en gissning hur bra positionen är. Vi behöver alltså en funktion som ur det stora trädet tar ut ett subträd som innehåller ett antal drag framåt från den aktuella positionen, sen behöver vi en funktion som beräknar godhetstalet för varje nod, och sen kan vi konstruera vår funktion som räknar ut hur bra nuvarande position är. Detta görs genom att vi antar att alla spelare hela tiden gör det bästa möjliga draget, och för att räkna ut hur bra nuvarande position är, d.v.s. dess värde, tar vi det maximala värdet från positionens barn i trädet och fortsätter rekursivt. Basfallet är när vi kommer till löven i vårt subträd. Där låter vi värdet vara detsamma som godhetstalet.

Funktionell programmering tillåter oss att göra ovanstående på ett snyggt och modulariserat sätt, precis som det är beskrivet. Vi kan ha en funktion för varje länk i kedjan, och sen bara låta dem kalla på varandra. Den översta funktionen, som beräknar positionens värde, anropar funktionen som tar maxvärdet av barnens positioner rekursivt, som i sin tur anropar funktionen som sätter godhetstal på alla noder i subträdet, som i sin tur anropar funktionen som tar ut subträdet ur hela trädet, som i sin tur anropar funktionen som gör själva trädet.

Vi kan nu utnyttja den funktionella programmeringens egenheter. Funktionen, som beräknar godhetstal, kan t.ex. ta en annan funktion som argument, som beräknar godhetstal till en specifik nod, och så har vi utnyttjat en funktion av högre ordning för att modularisera. Dessutom måste funktionen, som tar ut subträdet, använda slö uträkning, eftersom funktionen, som tar fram hela trädet, kan hålla på i oändlig tid, och eftersom alla funktioner är direkt länkade till varandra blir hela kedjan beroende av slö uträkning för att det skall fungera. Vill man inte använda slö uträkning måste man göra allt som en enda funktion, som direkt tar ut bara en del av trädet och arbetar på den, vilket blir mycket svårare och mycket mindre modulariserat.

I denna tillämpning är alltså funktionell programmering att föredra, eftersom slö uträkning och funktioner av högre ordning visar sig mycket användbara. Utan dessa skulle det bli mycket svårare och större risk för fel. Genom att tillåta funktioner av högre ordning kan vi t.ex. byta ut funktionen som beräknar godhetstal till varje nod och därmed kan vi göra denna situation mycket generell och inte specifik för just luffarschack. Det är helt klart att imperativ programmering skulle göra det hela mycket svårare. Vi hade inte kunnat dela upp allt i olika funktioner, utan vi skulle behöva ha allt i en enda stor funktion, vilket skulle ge större risk för fel, och det hade blivit svårare att överblicka och underhålla. Om vi behövde ändra någonstans, kanske för att generalisera eller använda i någon annan tillämpning, hade det varit svårare, eftersom en liten förändring kanske påverkat beteendet så att det inte fungerade någon annanstans.

# 5 Slutsatser

Vi har nu beskrivit två programmeringsparadigm och jämfört dem. Dessutom har vi beskrivit två tillämpningar där respektive paradigm var bättre än det andra. Av det kan man dra slutsatsen att det inte generellt går att säga att det ena paradigmet är bättre än det andra. Båda paradigmen visade sig ha för- och nackdelar, så man får för varje tillämpning bestämma vad som är viktigt i just den tillämpningen, och med det som utgångspunkt bestämma vilket paradigm som lämpar sig bäst. Är hög abstraktionsnivå och stora uttrycksmöjligheter ett krav, väljs med fördel funktionell programmering. Är istället extrem snabbhet och maskinnärhet ett krav, är imperativ programmering det bästa valet. Det kan naturligtvis också hända att inget av de två beskrivna paradigmen passar bra, och då får man välja något annat, som logikprogrammering eller evolutionär programmering.

## Referenser

[Ben-Ari] Ben-Ari, M. 1996. *Understanding Programming Languages.* John Wiley & Sons Ltd.

[Hendersson] Hendersson, P. 1980. *Functional Programming: Application and Implementation.* Prentice-Hall International, Inc., London.

[Hughes] Hughes, J. 1989. Why functional programming matters. *The Computer Journal, Vol. 32, No. 2.*
Internet URL (1999): <http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html>.

[Watt] Watt, D. A. 1990. *Programming Language Concepts and Paradigms.* Prentice Hall International (UK) Ltd.