Scala som nybörjarspråk i undervisningen

Andreas Lindroos

Kandidatuppsats inom datavetenskap
Handledare: Annamari Soini

Institutionen för informationsteknologi
Åbo Akademi

Åbo, april 2012

Referat

Medan efterfrågan på programmerare ökar världen över, så har högskolor och universitet mera press på sig att snabbare utbilda sina studerande och få dem ut på arbetsmarknaden. Ett problem för många institutioner är att välja vilket programmeringsspråk som ger bästa möjliga starten på studierna. Det finns inga standarder på vilka krav som skall ställas på ett introduktionsspråk och många utbildare och forskare har olika åsikter om saken. Man kan inte heller bevisa vilka krav som är bäst, eftersom alla studerande är unika och tänker olika. Detta försvårar valet av introduktionsspråk för institutionerna ytterligare.

Detta arbete sammanfattar de krav som ställs på introduktionsspråk av många utbildare och forskare inom området, med fokus på IT-studerande och ur den pedagogiska synvinkeln. Arbetet sammanfattar även programmeringsspråket Scalas egenskaper, samt behandlar huruvida språket motsvarar de krav som ställs på introduktionsspråk, genom analys av egenskaper och kodexempel.

Innehållsförteckning

1. Introduktion

Detta arbete är uppdelat i tre delar. Andra kapitlet sammanfattar de krav som ställs på programmeringsspråk i olika publikationer. Kapitlet beskriver de krav som många utbildare och forskare verkar vara överens om. Tredje kapitlet ger en introduktion till programmeringsspråket Scala, samt behandlar dess egenskaper. Fjärde kapitlet visar, med exempelprogram, huruvida Scala uppfyller de krav som ställdes i andra kapitlet.

Läsaren förväntas ha kunskaper inom programmering och grundkunskaper inom programmeringsparadigm.

2. Nybörjarspråkets egenskaper

Orsaken till att man studerar olika introduktionsspråk och deras egenskaper är att man vill effektivera inlärningen av programmeringsspråken och underlätta inlärningen av begreppen inom programmering. Första delen av detta kapitel är en snabb överblick över fördelar och nackdelar med olika programmeringsparadigm. Resten av kapitlet handlar mer ingående om olika krav på nybörjarspråk.

2.1 Programmeringsparadigm för nybörjare inom programmering

Det finns många programmeringsparadigm ute på marknaden och de har alla sina fördelar och nackdelar. Därför är det svårt att veta vilka paradigm som bäst passar för de studerande, samt ger en bra grund inom programmeringen. I detta kapitel tar vi reda på vilket programmeringsparadigm som idag kan rekommenderas som introduktion för nybörjare inom programmering.

Många utbildare anser att huvudämnesstuderande inom datavetenskap som huvudämne bör lära sig åtminstone två programmeringsparadigm under sina studier (Hadjerrouit, 1998). De två mest använda programmeringsspråktyperna på marknaden idag är de objektorienterade språken och de procedurala språken (se Tabell 1) och därför väljer många institutioner att lära ut dessa paradigm i första hand.

Kategori	Popularitet, februari 2011	Delta, februari 2010
Objektorienterade språk	57,7 %	+ 4,2 %
Procedurala språk	37,0 %	– 5,0 %
Funktionella språk	4,1 %	+ 1.0 %
Språk inom logikprogrammering	1,2 %	– 0.2 %

Tabell 1: Statistik på användning av paradigm (TIOBE Software BV, 2011)

Varför det objektorienterade programmeringsparadigmet är så populärt bland institutionerna, beror delvis på populariteten av de objektorienterade språken på marknaden (de Raadt, o.a., 2002). Detta leder till att första årets studerande som har tidigare erfarenheter inom andra paradigm måste utföra ett paradigmbyte för att lära sig begreppen bakom objektorienterad programmering (Tošić, o.a., 2008). Problemet med paradigmbyte för dessa nya studerande är att de bygger vidare på sin kunskap på basen av tidigare erfarenheter (Hadjerrouit, 1998). Det är svårt att lära ut de nya abstrakta och svåra begreppen i det objektorienterade paradigmet, om de studerande klänger sig till de tidigare kunskaperna inom t.ex. imperativ programmering.

Bytet till det objektorienterade paradigmet blir speciellt svårt för de studerande som lärt sig t.ex. det procedurala paradigmet från tidigare eller för de som är självlärda (Hadjerrouit, 1998) (Kölling, 1999). Detta beror på att de studerande som tidigare lärt sig den procedurala paradigmen har gått över till att använda metoder som procedurer istället för att använda objektorienterade koncept som polymorfism och arv (Hadjerrouit, 1998). Det alltså svårt att byta ut gamla vanor och man måste få studerande att förstå att kursuppgifter inte nödvändigtvis är klara, även fast de kompilerar och t.o.m. ger korrekt utskrift.

Svårigheterna med paradigmbytet har lett till att många väljer att lära ut det objektorienterade paradigmen först (de Raadt, o.a., 2002), men det finns även problem med detta tillvägagångssätt. Syntaxen och semantiken inom objektorientering blir ibland för abstrakt för nya studerande och detta leder bl.a. till svårigheter att förstå vad som händer i exempelprogram (Hadjerrouit, 1998). Genom att lära ut objektorienterad programmering först går det även mer tid åt att förklara objekt och dess innebörd, vilket betyder att för en del kan detta bli frustrerande då de inte förstår grundkoncepten i kursen.

De institutioner som har tacklat paradigmbytet med att först lära ut det objektorienterade grundkonceptet, har oftast börjat med att lära ut grundbegrepp som förekommer i alla paradigm, såsom utskrift, inmatning och kontrollstrukturer. Sedan har man ganska tidigt i kursen introducerat grundbegreppen inom objektorientering och gradvis introducerat mer avancerade begrepp (Hadjerrouit, 1998). Därför är det viktigt att välja ett språk som erbjuder möjligheten att gå igenom alla objektorienterade begrepp, men även erbjuder en lätt grund för inlärningen av grundfunktioner. Även om man har gått igenom begreppen gradvis har man ändå stött på problem med att de studerande inte hänger med i kursen. Det har alltså visat sig att studerande ändå har svårt att förstå begreppen inom objektorienterad programmering, men det är ändå lättare att lära ut begreppen åt studerande som inte har tidigare bakgrund inom andra paradigm. (Hadjerrouit, 1998)

Det finns inga direkta bevis på att objektorienterad programmering passar som första paradigm för alla studerande. Dock kan man, på basen av den popularitet på marknaden (TIOBE Software BV, 2011), bland universitet (de Raadt, o.a., 2002) och publikationer (Parker, o.a., 2006) (Kölling, 1999), dra den slutsatsen att undervisningen av ett objektorienterat språk som första språk till de studerande har stora fördelar, om man antar att de förr eller senare måste lära sig det objektorienterade paradigmet.

Då man lär ut objektorienterade programmeringsspråk bör man ta upp grundbegreppen inom objektorienterad programmering såsom arv, inkapsling, abstraktion och polymorfism (Kölling, o.a., 1995) (Parker, o.a., 2006). Det är viktigt att se till att det objektorienterade språk som väljs till introduktionskursen stöder liknande koncept, eftersom det finns hybridspråk ute på marknaden som påstås stöda objektorienterad programmering, men i verkligheten inte stöder alla grundkoncept (Parker, o.a., 2006).

I denna uppsats utgår jag ifrån att det objektorienterade paradigmet är det bästa alternativet som första paradigm för studerande och kommer därför att jämföra huruvida Scala passar in i dess begrepp. Denna slutsats dras på basen av de källor som nämnts tidigare i kapitlet och på grund av att majoriteten av forskarna verkar vara överens om att det objektorienterade paradigmet är det bästa alternativet. Eftersom Scala är ett språk som erbjuder egenskaper från både de objektorienterade och de funktionella språken, så kommer jag även kort att behandla den funktionella sidan av Scala.

2.2 Krav på språk för nybörjare

Det finns inga officiella standarder på krav på nybörjarspråk för studerande, eftersom alla krav har sina fördelar och nackdelar. Det finns dock publikationer som handlar om ämnet och tar upp de krav som en del utbildare och institutioner anser att man skall ställa på introduktionsspråket.

Enligt McIver och Conway (McIver, o.a., 1996) är följande egenskaper viktiga och bör analyseras då man väljer språk för nybörjare:

- Syntaxen: Är syntaxen för komplex eller för flummig?
- Kontrollstrukturerna, operatorerna och de inbyggda funktionernas mnemotekniska egenskaper: Är de lätta att komma ihåg; påminner de om tidigare strukturer inom t.ex. matematiken?

- Semantiken: Är språkets semantik onödigt komplicerad? Kommer språkets uppbyggnad att förvirra nybörjare?
- Felmeddelanden: Hurdan feedback ges av språket? Kommer nybörjaren att få tillräckligt klar feedback för att hitta fel och korrigera dem?
- Onödigt hårdvarubundet eller andra implementationsrestriktioner: Kommer språket att erbjuda de egenskaper som man vill undervisa och hur svårt är det att förklara restriktioner åt nybörjare?
- Språkets abstraktion: Har språket för många eller för få funktioner? Kommer språkets abstraktion att förvirra studeranden?
- Språkets fördelar: Är språkets egenskaper lika klara för nybörjare som de är för utbildaren? Kommer studerande att ha svårt att begripa egenskaper som är självklara för utbildaren?

Denna lista ger en bra bild av egenskaper som kan krävas av språk och därför används den som grund i detta kapitel. Vi kommer även att ta upp några andra krav som kan anses vara viktiga.

2.2.1 Utvecklingsmiljö för språk

Medan kurser inom programmering utvecklas och blir mer avancerade, är det viktigt att erbjuda nybörjare en inlärningsmiljö där de inte behöver spendera extra tid på att hitta syntaxfel eller motsvarande småfel. Tidigare användes enkla utvecklingsmiljöer som bestod av en texteditor och en kompilator för att lära ut programmering (Kölling, 1999). En bra utvecklingsmiljö tillåter de studerande att fokusera på sin inlärning av ämnet och gömma detaljer såsom operativsystemets egenskaper i bakgrunden (Kölling, 1999).

Kraven på en bra utvecklingsmiljö är att den skall vara lätt att använda, stöda objektorienterad programmering och kunna erbjuda stöd i undervisningen.

Stödet kan bestå av t.ex. visuella diagram, stöda återanvändning av kod, erbjuda stöd för grupparbeten och vara tillgängligt för de studerande (Kölling, 1999).

Idag har integrerade utvecklingsmiljöer blivit allt mer populära. Eftersom kurser lägger mer vikt på att lära ut testning av program (debugging), återanvändning av kod och avlusning av program, är en bra utvecklingsmiljö ett allt viktigare verktyg i introduktionskurserna (Kölling, 1999). Det finns många integrerade utvecklingsmiljöer ute på marknaden som erbjuder funktioner som motsvarar de tidigare nämnda kraven i bättre eller sämre utsträckning, beroende på språket i fråga. Detta tillåter institutionerna att själv välja en miljö som passar dem bäst.

2.2.2 Böcker och material

För att underlätta undervisningen är även dokumentationen och tillgängliga läroböcker av språket en faktor (Parker, o.a., 2006). Genom att man har tillgång till mer material, är det lättare för studerande och utbildare att söka sig till andra källor som ger flera perspektiv på problem och lösningar.

Det finns många faktorer kopplade till språkets dokumentation och antalet textböcker som finns tillgängliga. Språkets livscykel är en faktor, eftersom nya språk oftast inte har lika omfattande textböcker som de äldre språken. Det kan vara svårt att hitta bra material för nya språk, men vartefter språket utvecklas ökar även textböckernas antal kring språket. (Parker, o.a., 2006) Det finns dock praktiska exempel som visar att detta inte alltid stämmer, t.ex. då man jämför materialet kring språken Java och C; Java har en bättre dokumentering av språket, även fast det är betydligt nyare än C.

Det akademiska godkännandet av språket är även en stor faktor kring tillgängligheten av textböcker (Parker, o.a., 2006), eftersom godkännandet gör att språket blir mer allmänt accepterat och därmed leder till en större marknad för textböckerna kring språket, vilket i sin tur leder till fler böcker.

2.2.3 Tillgänglighet för studerande

Då man analyserar programmeringsspråkets lämplighet för användning i en utbildning, bör man även beakta huruvida eventuella kostnader för licenser och dylikt dyker upp (Parker, o.a., 2006). Flera utvecklare erbjuder gratis versioner av kompilatorer och även utvecklingsmiljöer, medan vissa kan kräva licenser som kostar. Kostnaden för dessa licenser kan variera på basen av vem som skall använda mjukvaran. Ibland kan även licenskostnader vara förmånligare för utbildningsinstitutioner än för företag.

Licenser och dylikt skall tas i beaktande eftersom man vill att de studerande skall ha möjlighet att på egen hand prova på språket och lösa kursuppgifter. Det är viktigt att erbjuda de studerande en miljö där de kan arbeta på sin egen tid och när de själva känner för det. (Parker, o.a., 2006). Genom att tillåta de studerande att installera de verktyg som krävs på en hemdator, tillåter man dem att lösa hemuppgifter oberoende av laboratorietider. Institutioner kan ofta lösa detta med att erbjuda verktygen över en nätförbindelse eller genom att köpa in en licens som även täcker de studerande. Vissa innehavare av rättigheterna till verktygen kan även erbjuda gratisversioner för studerande eller versioner som endast innehåller de funktioner som innehavarna själva anser att de studerande behöver i studierna.

2.2.4 Syntaxen

Språkets syntaktiska läsbarhet är något som bör tas i beaktande då man analyserar lämpligheten för nya studeranden. Syntaxen i ett språk är det första som en studerande kommer i kontakt med i användningen av språket. Om syntaxen är för svår att förstå eller om den inte är tillräckligt intuitiv så blir inlärningen svårare. Detta påverkar ofta studerandes motivation att lära sig programmering på ett negativt sätt.

Då språk använder sig av hela ord istället för symboliska tecken i syntaxen hjälper det nya studerande att enklare förstå vad en programkod innebär och vad den gör (Kölling, 1999). Programkoden blir mer förståelig då man kan läsa av den direkt, istället för att fundera på vad olika symboler har för betydelse. Ett exempel på detta kan vara skillnaden mellan Javas och Pythons if-satser (se Tabell 2). Även om en erfaren Javaprogrammerare läser satsens argument utan problem, så är det mer naturligt och flytande för en nybörjare att läsa Pythons argument. Orden ”and” och ”or” är lättare att förstå än symbolerna ”&&” och ”||”, vilket innebär att någon som har läst grunderna i det engelska språket nästan kan gissa sig till vad satsen betyder.

Java	Python
`if (arg0 == true && (i == 1 || i == 0))`	`if arg0 == True and (i == 1 or i==0):`

Tabell 2: If-sats i Java och Python

För nya studerande kan det vara svårt att utesluta vad som är det viktiga då de läser programkod och vad som är förväntat att de skall få ut av koden. Därför vill man helst använda ett språk som erbjuder en enkel och ren syntax. Möjligheten att exekvera korta exempelprogram med så lite extra kod som möjligt är även ett plus, eftersom detta tar bort onödig extra syntax som man kanske inte ännu vill att studerande skall fundera på. Exempel på denna syntaxskillnad i språk kan man se då man jämför Pythons och Javas HelloWorld-program (se Tabell 3). HelloWorld är oftast det första program man skriver i ett nytt språk och i detta skede av vill man att studerande skall få en enkel utskrift och inte behöva fundera på vad en publik klass eller en statisk metod innebär.

Java	Python

`public class HelloWorld{`
`  public static void main(String args[]){`
`  System.out.println(“Hello World!”);`
`  }`
`}`
	`print ”Hello World!”`

Tabell 3: HelloWorld i Java och Python

Syntaktiska synonymer kan även visa sig vara ett problem för nybörjare. Syntaktiska synonymer innebär att ett språk tillåter användaren att göra samma sak med olika syntaktiska strukturer. Ett exempel på ett syntaktiskt synonym är sättet att läsa räckor i C. I C kan användaren läsa räckors element både med kodraden ”racka[2]” och ”2[racka]”. (McIver, o.a., 1996) Detta kan misstolkas som en bra sak eftersom man tillåter användaren att göra saker på det sätt den vill. Detta kan dock leda till mer förvirring för nybörjaren då exempelprogram från böcker eller internet kan använda dessa syntaktiska synonymer, medan utbildaren inte har nämnt dem i kursen eller studerande inte har förstått dem.

2.2.6 Kurskrav bland utbildningar i Sverige och Finland

Då man ser på valet av introduktionsspråk bland högskolor och universitet från Sverige och Finland har de olika lärandemål för sina kurser, men i grund och botten liknande grundkrav som de utgår från. Detta beror på att utbildningarna inom de olika skolorna och universiteten fungerar på olika sätt. Vissa har valt att integrera fler krav i en kurs, medan andra utbildningar har valt att splittra kurser i fler mindre kurser.

Introduktionen till programmering för datavetenskapsstuderande i Uppsala Universitet sker i kursen ”Programkonstruktion och datastrukturer” (Uppsala Universitet, 2010). Kursen är väldigt omfattande och avklarad kurs ger studerande 20 högskolepoäng. Lärandemålen på denna kurs inkluderar att studerande har grundläggande kunskaper inom testning av program, avlusning, iteration, användning av datastrukturer, samt grundläggande kunskaper i syntax och semantik inom ett funktionellt språk. Eftersom kursen är så omfattande tar de även upp mer om algoritmer för sökningar, matematik inom algoritmanalys etc., vilket inte detta arbete kommer att behandla utförligare.

Åbo Akademi har valt att splittra upp kurserna inom ämnet datavetenskap så att man har en specifik kurs som heter ”Programmering grundkurs” och ger studeranden 5 studiepoäng (Åbo Akademi, 2012). Denna kurs behandlar

grunderna inom programmering och har som lärandemål att studerande skall ha grundläggande kunskaper inom bl.a. analys och förklaring av enkla program, modifikation, utökning och implementation av kontrollstrukturer och funktioner, testning av program, avlusning, samt tillämpning av strukturella och modulära tekniker för att dela upp program i mindre delar.

Åbo Akademi har även en kurs för ämnet Kemiteknik som introducerar C för nya de studerande inom ämnet. Kursens innehåll motsvarar 5 studiepoäng och har som lärandemål att studerande skall kunna förklara hur ett program fungerar, enkla styrstrukturers effekt på data som behandlas och kunna algoritmiskt lösa enkla problem i språket C.

I Kungliga Tekniska högskolan har man en introduktionskurs till programmering som heter ”Grundläggande programmering och datalogi” (Kungliga Tekniska högskolan, 2012). Efter avklarade prestationer får studerande 7,5 högskolepoäng. I kursen används i huvudsak programmeringsspråket Python, men man kräver även att de studerande skall kunna använda språket Matlab för visualisering och lösning av matematiska problem. Kursens innehåll består av bl.a. objekthantering, kontrollstrukturer, iterationer, grunder inom datastrukturer och sortering.

Kurskraven ser väldigt liknande ut i grund och botten, men man hanterar programmering i varierande utsträckning. Eftersom man i Uppsala Universtitet har valt att kursen skall vara väldigt omfattande, ger den även fler poäng än t.ex. grundkursen i programmering vid Åbo Akademi. Dock kan man se att alla kurser i stort sett har samma grundkrav på sina studerande som t.ex. iteration, kontrollstrukturer, grundläggande datastrukturer, avlusning mm.

Användning av språk inom universiteten är varierade. Uppsala universitet har valt att fokusera sig mer på funktionella språk och kräver att de studerande skall kunna redogöra för syntax och semantik i ett funktionellt språk som t.ex. Standard ML (Uppsala Universitet, 2010). Kungliga Tekniska högskolan har däremot valt att införa mindre funktionella språk, såsom Python och Matlab till introduktionskurserna och betonar även kravet på grundkunskaper inom Matlab

bland lärandemålen för kursen (Kungliga Tekniska högskolan, 2012). Åbo Akademi har valt en mer språkneutral linje för nya IT-studerande och skriver inte uttryckligen något specifikt språk eller paradigm i det officiella lärandemålet. I praktiken används dock antigen C eller Python som introduktionsspråk för studerade inom Åbo Akademi. Vid det Öppna Universitet inom Åbo Akademi erbjuds även Java som introduktionsspråk. Denna kurs motsvarar den kurs som erbjuds åt studerande inom datavetenskap, men har lite annorlunda innehåll.

För Åbo Akademis kemitekniska linje har man valt att införa C som startspråk, mest på grund av den starka standarden av C inom kemiindustrin. Det är inte heller förväntat att en kemiteknisk studerande skall kunna designa några större applikationer, utan bara kunna lösa enkla problem på egen hand. Detta betyder alltså att problemet med paradigmbyte och motsvarande inte är lika aktuellt inom ämnet, som det är inom t.ex. datavetenskap eller datateknik.

Man kan alltså konstatera att även om de grundläggande kurskraven för introduktionskurser i programmering verkar vara liknande, verkar språken variera mellan utbildningarna. Detta är ännu ett exempel på hur man inte kan bevisa vilket eller vilka språk som fungerar bäst som introduktionsspråk inom alla utbildningar, utan varje utbildning måste göra ett enskilt beslut om vilka saker de vill behandla i vilka kurser och i vilken ordning dessa skall tas upp.

3. Programmeringsspråket Scala

Detta kapitel kommer att fungera som introduktion till språket Scala och ge en grund i vad Scala har att erbjuda. Det är orimligt att gå igenom alla Scalas egenskaper i detalj. Så i detta kapitel fokuserar vi oss endast på de viktigare egenskaperna som Scala erbjuder och ur dessa plockar vi fram de egenskaper som kan anses vara nödvändiga för att kunna behandla huruvida Scala passar som nybörjarspråk för nya studerande.

3.1 En kort introduktion till Scala

Idén till Scala kom från programmeringsspråket Pizza, som kombinerade det objektorienterade och det funktionella paradigmet i en omgivning som även kan samarbeta med Java. Språket Pizza fick aldrig någon stor användarkrets, men visade att man kunde kombinera dessa två paradigm och resultatet av kombinationen var ett kraftigt hybridspråk. Bland utvecklarna av Pizza fanns även Scalas designer Martin Odersky.

Utvecklingen av Programmeringsspråket Scala fick sin början under åren 2001-2004 i ett laboratorium i EPFL (École Polytechnique Fédérale de Lausanne) i Schweiz. Orsaken till att man började utveckla språket var att man ville uppnå ett bättre stöd för utveckling av program i komponenter (Component-based software engineering). Man skulle uppnå detta genom att erbjuda programmeraren en skalbar miljö, samt en kombination av det funktionella och det objektorienterade paradigmet. (Odersky, o.a., 2004) Namnet Scala härstammar från dess skalbarhet; språket har en förmåga att växa på basen av användarens behov och passar för både mindre skript och större system (Odersky, o.a., 2008).

För att kunna vara säker på att utvecklingen gick åt rätt håll behövde man kommentarer från användarna av språket. Därför var det viktigt att göra det lätt för användarna av andra språk att pröva på Scala, utan att behöva lära sig en överväldigande mängd nya syntaktiska strukturer och tankesätt. På grund av detta utvecklade man språket så att det syntaktiskt liknar språk som redan används på marknaden, såsom C# och Java, och fungerar väl i samband med dessa. Scala lånar även implementationsidéer från språk som Haskell, SML och Smalltalk. (Odersky, o.a., 2004)

3.2 Utvecklingsmiljöer & licenser

Detta kapitel är uppdelat i två delar. I första delen behandlas de tre olika sätt som man kan exekvera kod i Scala. I andra delen diskuteras kort några av utvecklingsverktyg som finns på marknaden och huruvida hot mot Scalas öppna källkod existerar.

3.2.1 Exekvering av kod

Scala-kod kan exekveras på tre olika sätt: användning av en kommandotolk, körning av Scala skript eller kompilering och exekvering. I denna del går vi igenom dessa sätt att exekvera kod, samt deras fördelar och nackdelar.

Scalas kommandotolk

Det första sättet är att använda Scalas egen tolk som kallas REPL (Read Evaluate Print Loop). Tolken kommer med då man installerar Scala och startas enkelt genom att endast exekvera kommandot ”scala” i operativsystemet. Tolken fungerar på samma sätt som de flesta vanliga kommandotolkar i andra språk (t.ex. Python) och den klarar av alla de vanligaste operationerna som t.ex. att spara variabler, definiera funktioner och modifiera objekt.

Största fördelen med en kommandotolk är att man snabbt kan testa enkla algoritmer och göra enkla beräkningar eftersom den inte kräver att man definierar hela klasser, utan den tillåter att man exekverar t.ex. endast en loop som innehåller en utskrift.

Eftersom kommandotolken exekverar koden vartefter man matar in den, klarar tolken av att genast hitta syntaxfel. Detta erbjuder nybörjaren en enkel och snabb miljö för att prova på kodsnuttar, eftersom resultatet av koden kan ses vartefter den matas in.

Tolken har även en inbyggd ”auto completion”-funktion som ger användaren en lista på metoder tillhörande en klass eller hjälper till att hitta tidigare definierade variabelnamn då man trycker på ”tab”-knappen. Detta innebär att nybörjare lätt kan se vilka metoder som är tillgängliga och kan slutföra halvskrivna kommandon om han/hon inte kommer ihåg den exakta syntaxen.

Scala-skript

Det andra sättet att exekvera Scala-kod är att skriva koden i en fil och exekvera programmet som ett skript. På detta sätt kan man lättare skriva hela program eller kodsnuttar i en editor och sedan exekvera hela koden på en gång. Fördelen med att exekvera koden på detta sätt jämfört med kommandotolken är givetvis att det är lättare att modifiera större program och man är inte heller bunden till REPL-tolkens gränssnitt. Skriptet sparas i en fil med filändelsen ”.scala” och exekveringen av skriptet görs med kommandot ”scala <filnamn>.scala”.

Scalas kompilator

Det tredje sättet att exekvera program i Scala är att använda sig av Scalas kompilator och sedan exekvera den kompilerade filen. Kompileringen görs med kommandot ”scalac <filnamn>.scala”. Exekveringen görs med kommandot ”scala <filnamn>”. Nackdelen med att använda en kompilator är att man måste definiera en hel klass för att den skall kunna kompileras, vilket tar bort möjligheten att exekvera kodsnuttar. En annan nackdel är att kompileringstiden i

Scala är lång jämfört med till exempel Java. Då man jämför kompileringstiden för ett helloworld-program i Scala (se Tabell 4) och Java (se Tabell 3), så kan man se att Java-kompilatorn är nästan tio gånger snabbare.

```
object hellocomp {
           def main (args: Array[String]){
                      println("HELLO WORLD")
           }
}
```

Tabell 4: Helloworld i Scala

Fördelen med att använda Scalas kompilator istället för den skriptbaserade varianten är att kompilatorn skapar s.k. JVM (Java Virtual Machine) bytekod. Bytekoden som skapas av Scalas kod är nästan omöjlig att åtskilja från motsvarande kompilerad Javakod. Man kan till och med avkompilera den kompilerade Scala-koden och få ut motsvarade kommandon som Javakod. (Kullberg, o.a., 2011) Detta betyder den kompilerade bytekod som fås från Scalas kompilator kan köras utan att ha Scala installerat och exekveringen kräver endast att en JVM-miljö är installerad. En annan fördel med att använda kompilatorn istället för att skriva skript är att själva exekveringen av programmet är snabbare, vilket spelar större roll då man skriver större applikationer eller program som skall exekveras fler gånger.

3.2.2 Verktyg och licenser

Utöver Scalas egen tolk finns det även en mängd andra utvecklingsmiljöer att skriva Scala program i. Eclipse[1] är en av de mer populära utvecklingsmiljöerna på marknaden idag och som plugin till Eclipse hittar man även Scala IDE[2]. Scala IDE erbjuder de vanligaste egenskaperna som man kan förvänta sig av en utvecklingsmiljö, som t.ex. hyperlänkar, markering av syntaxfel, avlusningsverktyg och automatisk indentering (Scala IDE, 2012). Scala IDE är även

[1] http://www.eclipse.org/
[2] http://scala-ide.org/

sponsorerat av det internationella företaget Typesafe[3], som Martin Odersky både leder och varit med och grundat. Typesafe erbjuder bl.a. stöd och utbildning i Scala för företag runtom i världen. Utöver Scala IDE finns det plugins till många andra utvecklingsmiljöer som t.ex. IntelliJ och Netbeans.

Verktygen som erbjuds idag utvecklas konstant och man kan inte förvänta sig att utvecklingsmiljöerna idag stöder Scala lika bra som de t.ex. stöder Java. Detta beror på att verktygen oftast utvecklas inom intressegrupper där medlemmarna arbetar gratis och på sin egen tid; eftersom Scala inte är lika populärt som Java finns det inte lika många frivilliga att utveckla verktygen.

Scala har öppen källkod och är tillgängligt för alla. Språket upprätthålls av en grupp som heter LAMP[4] på EPFL. Detta innebär att det inte finns något stort företag bakom språket och pengarna bakom underhållet fås från skattemedel. Detta tryggar även språkets framtid, eftersom de stora företagen inte kan köpa upp rättigheterna eller på något annat sätt påverka Scalas licens. (Kullberg, o.a., 2011)

3.3 Egenskaper

I denna del går vi igenom en del av de egenskaper som Scala hämtar med sig. Som grund används egenskaper ur böckerna ”Programming in Scala” (Odersky, o.a., 2008) och ”Ett första steg i Scala” (Kullberg, o.a., 2011).

Scala är kompatibelt med Java. I Scala kan man bl.a. kalla på Javametoder, utvidga (extend) Javaklasser och implementera Javadefinierade gränssnitt (interface). Detta kan göras utan någon speciell syntax eller omskrivna gränssnitt, och Scala använder även internt Javabibliotek i en stor utsträckning. Även variablerna och räckorna i Scala använder Javas motsvarighet för att allokera data. Utöver att endast utnyttja Javas variabler utökas även variablerna i Scala

[3] http://www.typesafe.com/
[4] http://lamp.epfl.ch/home

med fler funktioner (Odersky, o.a., 2008), vilket innebär att man kan lättare göra typomvandlingar och dylikt. Ett exempel på detta är att man lätt kan typomvandla strängar till heltal genom att använda ””123”.toInt”. I Java måste man hänvisa till heltalsklassen med kommandot ”Integer.parseInt(”123”);”, vilket i sin tur returnerar heltalet 123.

Syntaxen i Scala är designad så att den skall minska på antalet kodrader. Detta leder till att flerradiga lösningar i andra språk ibland kan lösas med ett anrop i Scala. Vilket i sin tur leder till att programmen blir mer lättlästa för det tränade ögat och minskar även på antalet fel i koden (Odersky, o.a., 2008). Ett exempel på hur man minskar på onödig kod i Scala är att man har gjort användningen av semikolon (som signalerar kodradens slut i bland annat Java och C) till valfritt. Ett lite mer betydande exempel är att en klass med konstruktor lätt kan skapas med ”class minKlass(var variabelA: Int, var variabelB: Int)”. Denna kodrad skapar alltså en konstruktor för klassen ”minKlass” som tar in två heltal och sparar dem som privata variabler i objektet, vilket skulle kräva fler kodrader i språk som Java och Python. Med hjälp av bl.a. dessa syntaktiska egenskaper så kan man säga att antalet kodrader i ett Scala program brukar vara 25-50 % mindre än motsvarigheten i Java (Kullberg, o.a., 2011).

Scala använder sig av statisk typning (Odersky, o.a., 2008). Detta innebär bland annat att Scala sparar sina variabler med en viss typ och det uppstår fel om man senare försöker tilldela någon annan typ av värde till den. Ett exempel på detta är då man deklarerar en variabel med värdet ”hej” (sträng) och senare försöker tilldela den värdet 123 (heltal). I ett dynamiskt språk skulle tilldelningen tillåtas och varken datorn eller programmeraren skulle märka någon skillnad, ända tills man använder variabeln och den innehåller något helt annat än man förväntar sig. I ett statiskt språk skulle detta resultera i ett kompileringsfel och därmed märker man felet innan man exekverar programmet. På detta sätt får man automatiskt bort liknande triviala fel, innan man lanserar programmet eller börjar använda det.

I Scala används ett avancerat system för att hålla reda på vilken typ av variabel man deklarerar. Deklarationen påbörjas med antigen ”val” eller ”var”. Användningen av ”val” anger att värdet som man ger till den är konstant och kan inte utbytas mot något annat värde. ”var” deklarerar en variabel som man kan ändra på och omdefiniera vid behov (Odersky, o.a., 2008). Eftersom Scala själv håller reda på variabeltypen behöver man som programmerare inte skriva ut vilken typ av variabel man vill deklarera. Man behöver alltså endast skriva ”var nummer = 123” för att deklarera en variabel vid namnet ”nummer” och Scala märker själv vilken sorts typ variabeln motsvarar och deklarerar det till ett heltal. Scala tillåter även att man deklarerar sina variabler till specifika typer, vilket innebär att man kan spara ett heltal i ett långt heltal (Long) med kodraden ”var heltal:Long = 12345”. Man bör dock notera att Scala inte tillåter man senare försöker tilldela någon annan typs värde till en tidigare definierad variabel. Alltså att man först deklarerar en heltalsvariabel och senare försöker tilldela samma variabel en sträng med kodraden ”heltal = ”hej””. Som parametrar till en funktion krävs dock att man definierar typen som kommer in till funktionen på liknande sätt som man även måste definiera typen för parametrar i Java (se Tabell 5). Man måste även definiera typen av returvärdet, om ett sådant skulle vara aktuellt. Scalas avancerade typsystem har förenklat programmeringen till en del, men för att uppnå en bra struktur på program så har man krävt att programmeraren använder typdeklarationer på vissa ställen.

Java	Scala
`void exekvera (String x, int y) {...}`	`def exekvera(x: String, y:Int) {...}`

Tabell 5: Jämförelse mellan Java och Scala

Refaktoreringen (refactoring) av ett större program är lättare att göra i ett statiskt språk som Scala. Eftersom variabler har en viss klass associerad med dem, så kan utvecklingsverktyg som Eclipse klara av att byta ut klasser och omdefinierade dem automatiskt (Kullberg, o.a., 2011). I dynamiska språk måste refaktoreringen oftast göras manuellt, vilket ofta kan leda till att

programmeraren glömmer att byta på alla platser och felet ligger i koden ända tills man exekverar den bit av programmet som kan resultera i att programmet kraschar.

3.4 Scala är fullt objektorienterat

Scalas designer klassar Scala till ett fullt objektorienterat språk, på samma sätt som språket Smalltalk klassades till ett fullständigt objektorienterat språk; allt är objekt (Odersky, o.a., 2004). Detta innebär att förutom att det uppfyller grundprinciperna i ett objektorienterat språk (arv, inkapsling mm.) så är även varje operation i språket ett metodanrop och varje variabel är ett objekt och tillhör en klass (Odersky, o.a., 2008).

Det finns inga operatorer i Scala, utan endast metoder (Kullberg, o.a., 2011). För att demonstrera detta så kan man se på exempelkoden i Tabell 6. Den högra sidan är ett exempel på hur man kunde tänka sig att en programmerare skulle skriva det och på vänstra sidan av tabellen är hur Scala själv tolkar högra sidan. Dock är båda sidorna lika korrekt att skriva i Scala och båda exemplen returnerar 10. Eftersom operatorer är metoder och det ända som skiljer dem åt är hur man skriver syntaxen, så kan man även använda metoder som operatorer. Detta kan till exempel göras om man har en variabel ”text” som innehåller strängen ”hello world” och sedan exekverar ”text indexOf (’w’)”. Man kan nu säga att indexOf-metoden har använts som en operator mellan variabeln text och tecknet w. Eftersom Scala tillåter att man överbelastar (overloading) metoder så kan man även laga egna metoder för t.ex. + och – som hanterar de värden man behöver hantera i till exempel ett bibliotek (Odersky, o.a., 2008). På detta sätt kan man laga väldigt lättanvända bibliotek, men detta kan även leda till svårigheter då andra medarbetare skall försöka tolka ditt program och man använder vanliga operator-tecken mellan blandade variabler.

`val heltal = 5` `heltal.+(5)`	`val heltal = 5` `heltal + 5`

Tabell 6: Båda exemplen slutar med 10 som resultat.

Alla värden man anger i Scala är objekt och hör till en klass (Odersky, o.a., 2008). Alltså tolkas även primitiva värden som ”123” till objekt och de innehåller metoder för till exempel plus, minus och division. Man kan alltså kalla på dessa metoder genom att skriva ”(5).+(5)” för att få resultatet 10 eller ”5.toString” för att få en tolkad sträng på värdet 5. Konceptet med att tolka varje variabel som ett objekt är något som inte många språk har implementerat och det är en av orsakerna till att man inte kan säga att Java är fullt objektorienterat.

3.5 Scala är funktionellt

Scala är inte bara objektorienterat, det är även ett funktionellt språk. Det funktionella paradigmet kommer ursprungligen från matematikens lambda beräkningar, vilket utvecklades av Alonzo Church redan på 1930-talet (Hudak, 1986). Idén bakom funktionell programmering gick sedan vidare till riktiga språk som Lisp, ML och Haskell. Odersky (Odersky, o.a., 2008) nämner att funktionella språk har två huvudidéer bakom sig. Första idén är att funktioner skall kunna tolkas som vanliga variabler och andra idén är att variabler endast skall tillåta oföränderliga tillstånd (immutable state).

Scalas funktioner skall kunna tolkas som vanliga variabler för att språket skall kunna klassas som funktionellt. Funktioner skall alltså kunna sparas i vanliga variabler, men det skall även gå att skicka dem som parametrar till funktioner och returnera dem. Detta innebär även att man måste kunna definiera en funktion i en funktion (Odersky, o.a., 2008). Ett exempel på detta kan ses nedan.

```
1  object funcDemo {
2    def main (args: Array[String]){
3
4          lazy val returnedfunc = get(stringToBool)
5          println(returnedfunc(5))
6
7          def stringToBool(x:String):Boolean = {
8              x == "test"
9          }
10   }
11
12   def get(y:String => Boolean): Int => String = {
13
14         def intToString (x:Int):String = {
15              val y = x + 5
16              y.toString
17         }
18
19         println(y("test"))
20         intToString
21   }
22 }
```

Tabell 7: Exempel på hur funktioner kan användas i Scala

Programmet i Tabell 7 börjar körningen i main-funktionen med att definiera ett lazy-värde[5] ”returnedfunc” och tilldelar det värdet som returneras från get() funktionen. I detta fall returnerar ”get” en funktion som tar in ett heltal och returnerar en sträng (rad 12: ”Int => String”). Funktionen ”get” tar även in en parameter y. Parametern y skall innehålla en funktion som tar in en sträng och ger ut ett booleskt värde (rad 12: ”y:String => Boolean”). I vårt fall har vi skickat stringToBool, som är definierad inuti main-funktionen (rad 7) som parameter. Funktionen ”stringToBool” tar in en sträng och kollar om den motsvarar ”test” och returnerar svaret. Körningen kommer att fortsätta i get-funktionen och exekverar till näst println-satsen (rad 19). I println-satsen har vi y som parameter, alltså kommer nu en kopia av stringToBool att exekveras med parameter ”test” och returnera svaret ”True”, vilket skrivs ut på skärmen. Det sista vi gör i get-funktionen är att vi returnerar intToString funktionen (rad 20) till main. Notera även här att funktionen ”intToString” är definierad inuti get-funktionen. I main

[5] Variabeln måste bli klassad som ett lazy-värde, eftersom kompilatorn annars kommer att försöka initialisera värdet då funktionen ”get” returnerar något, vilket i sin tur kommer att ge felmeddelande då kompilatorn försöker initialisera funktionen som kommer tillbaka. Ett lazy-värde initialiseras först då det refereras för första gången.

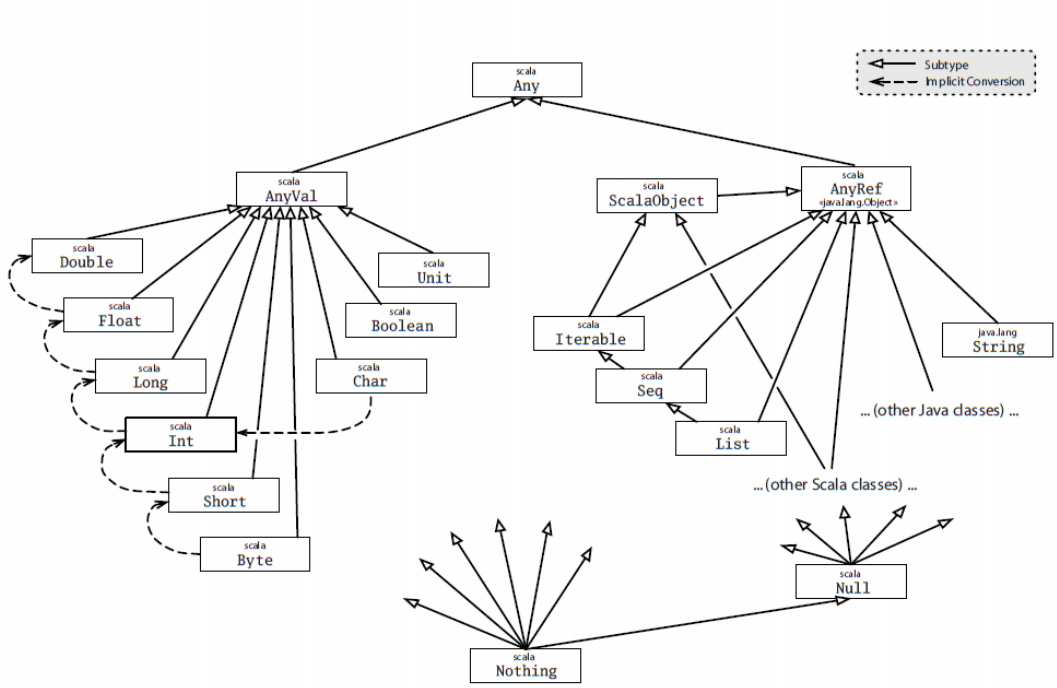
kan vi nu alltså använda värdet ”returnedfunc” som en funktion (precis som vi använde y tidigare) och den kommer att exekvera en kopia på koden som fanns i intToString. Resultatet av exekveringen av println-satsen (rad 5) kommer att lägga till ”10” på skärmen. Med detta exempel har vi nu visat att Scala kan tolka funktioner som vanliga variabler och även klarar av att skicka funktioner som parametrar och returvärden. Vi har även visat att Scala klarar av att ha funktioner i funktioner.

Ett funktionellt språks variabler skall stöda oföränderliga tillstånd. Med detta menar man att en variabel endast kan ha ett värde genom hela programmets gång. Ett exempel är att man i matematiken anser ”x = x + 1” att vara en matematisk omöjlighet och detta anses även vara en omöjlighet i funktionella språk. Medan man i språk som t.ex. Java tillåter ”x = x + 1” och resultatet blir att ett adderas till variabeln x. (Kullberg, o.a., 2011) I Scala har man dock ansett att detta krav är lite för extremt begränsa sig till, men man har istället gjort det lättare för användaren att deklarera oföränderliga variabler med hjälp av ”val” (diskuterades i kapitel 3.3 Egenskaper).

Odersky (Odersky, o.a., 2008) säger även att funktioner inte skall ha några sidoeffekter. Detta innebär att funktioner endast skall ta in parametrar och returnera värden. Funktioner skall alltså inte ha sidoeffekter som t.ex. att ändra på värden utanför sin räckvidd (scope). Ett exempel på detta är att då man använder metoder på strängar i t.ex. Java. Där returneras endast ett värde från metoderna och strängen som man använder metoden med förblir oförändrad.

3.6 Scalas Klasshierarki

Scalas klasshierarki ger en bra översyn över språket och hur det är uppbyggt. I denna del tar vi en snabb titt på hur Scalas klasshierarki ser ut och tillika får vi en syn på hur Scala är sammankopplat med Java.



Figur 1: Scalas klasshierarki (Källa: (Odersky, o.a., 2008))

I Scala ärver varje objekt i språket från klassen Any (se Figur 1). Detta innebär att varje objekt i språket kan använda sig av vissa grundmetoder från Any-klassen som t.ex. jämföra sig med andra objekt (== och equals) eller skriva ut objekt till en motsvarande sträng (toString). Då man går neråt i hierarkin så delar man på variabeltyper och objekten till AnyVal och AnyRef. AnyRef är ett alias för java.lang.Object från Java och ger en grund till varje referensklass i Scala. (Odersky, o.a., 2008) I Figur 1 kan man även se att Scala direkt använder sig av Javas java.lang.String klass, vilket igen visar hur sammankopplat Scala är med Java.

AnyVal är en klass som ärvs av Scalas nio inbyggda värdeklasser: Double, Float, Long, Int, Short, Byte, Char, Boolean och Unit. De första åtta värdeklasserna representeras under körning på liknande sätt som värdeklasserna i Java; t.ex. då man skriver ”123” så tolkas det till en Scala Int. Unit är konstant definierat till ”()” och behövs i Scala eftersom funktionerna i Scala alltid returnerar något och om inget annat är definierat av användaren så returneras Unit (Odersky, o.a., 2004). Man kan alltså i stort sett säga att Unit motsvarar Javas void. Istället för att

värdeklasserna ärver av varandra så har man valt att vid behov konvertera värdena mellan dem och representera dem i skilda klasser. Detta görs bl.a. för att vara säker på att värden skall behålla sin representation även om man typomvandlar dem (Odersky, o.a., 2004).

Null-klassen och Nothing-klassen finns på längst ner på Scalas hierarki. Null representerar en null-referens och är en underklass till alla referensklasser i Scala. Eftersom den endast är en underklass till referensklasserna, så kan man inte tilldela null till till exempel ett Int-objekt. Nothing är kopplat till alla klasser i Scala och används endast för att meddela användaren om ett fel har uppstått. Om en funktion returnerar ett Nothing-värde, så vet användaren att något fel har uppstått under exekveringen av funktionen. (Odersky, o.a., 2008)

4. Scala som nybörjarspråk

Detta kapitel kommer att behandla huruvida Scala passar som nybörjarspråk. Som grundkrav på språket används kraven som är fastställda i kapitel 2. För att kunna bättre motivera hur bra kraven uppnås eller inte uppnås, så jämförs Java med Scala. Kapitlet kommer att innehålla kodexempel i både Java och Scala.

5. Sammanfattning

I detta kapitel sammanfattar jag mina egna tankar bakom arbetet, vad jag kom fram till, mina egna åsikter om resultatet, finns det frågor i arbetet som blev för snabbt behandlade eller kunde diskuteras ytterligare osv.

Litteraturförteckning

de Raadt Micheal, Watson Richard och Toleman Mark Language Trends in Introductory Programming Courses [Konferens] // In The Proceedings of Informing Science and IT Education Conference. - Cork, Ireland : InformingScience.org, 2002. - ss. 329-337.

Hadjerrouit Said Java as first programming language: a critical evaluation [Tidskrift] // SIGCSE Bulletin. - June 1998. - 2 : Vol. 30. - ss. 43-47.

Hudak Paul Conception, Evolution, and Application of Functional Programming [Tidskrift]. - New Haven, Connecticut : ACM Computing Surveys, 1986. - 3 : Vol. 21.

Kullberg Olle, Klang Viktor och Lundberg Örjan Ett första steg i Scala [Bok]. - Malmö, Sverige : Studentlitteratur, 2011. - Vol. 1 : ss. 8-11, 14-15, 29, 32, 40, 159.

Kungliga Tekniska högskolan KTH | DD1345 Grundläggande programmering och datalogi [Online] // KTH | Kungliga Tekniska högskolan. - den 20 Februari 2012. - den 5 Mars 2012. - http://www.kth.se/student/kurser/kurs/DD1345.

Kölling Michael The Problem of Teaching Object-Oriented Programming, Part 1: Languages [Tidskrift] // Journal of Object-Oriented Programming. - 1999. - 8 : Vol. 11. - ss. 8-15.

Kölling Michael The Problem of Teaching Object-Oriented Programming, Part 2: Environments [Tidskrift] // Journal of Object-Oriented Programming. - 1999. - 9 : Vol. 11. - ss. 6-12.

Kölling Michael, Koch Bett och Rosenberg John Requirements for a First Year Object-Oriented Teaching Language [Tidskrift] // SIGCSE Bulletin. - 1995. - 1 : Vol. 27. - ss. 173-177.

McIver Linda och Damian Conway Seven Deadly Sins of Introductory Programming Language Design [Konferens] // Software Engineering: Education and Practice, 1996. Proceedings. International Conference. - Dunedin : [u.n.], 1996.

Odersky Martin [o.a.] An Overview of the Scala Programming Language [Rapport]. - Lausanne, Switzerland : École Polytechnique Fédérale de Lausanne, 2004.

Odersky Martin, Spoon Lex och Bill Venners Programming in Scala [Bok]. - Mountain View : Artima, Inc., 2008. - ss. 3-5, 9-12, 12-19, 26, 112, 206-212.

Parker Kevin R. [o.a.] A Formal Language Selection Process for Introductory Programming Courses [Tidskrift] // Journal of Information Technology Education. - 2006. - Vol. 5. - ss. 134-151.

Scala IDE [Online] // Scala IDE for Eclipse. - den 11 Mars 2012. - den 18 Mars 2012. - http://scala-ide.org.

TIOBE Software BV TIOBE Programming Community Index for February 2011 [Online] // TIOBE Software: The Coding Standards Company. - Februari 2011. - den 25 Februari 2011. - http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Tošić Dušan och Vujošević-Janičić Milena The role of programming paradigms in the first programming courses [Tidskrift] // THE TEACHING OF MATHEMATICS. - 2008. - 2 : Vol. 11. - ss. 63–83.

Uppsala Universitet Programkonstruktion och datastrukturer - Uppsala universitet [Online] // Uppsala Universitet. - den 18 Mars 2010. - den 5 Mars 2012. - http://www.uu.se/utbildning/utbildningar/selma/kursplan/?kKod=1DL201.

Åbo Akademi Åbo Akademi - Datavetenskap [Online] // Åbo Akademi. - den 17 Januari 2012. - den 5 Mars 2012. - http://www.abo.fi/public/datavet.