Avdelningen för datavetenkap

Patrik Isaksson och Mikael Lindmark

En analys av fem skriptspråk - Egenskaper och utveckling

An analysis of five scripting languages - Characteristics and development

Datavetenskap
D-uppsats (10p)

Datum/Termin:	06-03-31
Handledare:	Thijs Holleboom
Examinator:	Donald Ross
Löpnummer:	D2006:04

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

En analys av fem skriptspråk - Egenskaper och utveckling

Patrik Isaksson och Mikael Lindmark

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en magisterexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Patrik Isaksson och Mikael Lindmark

Godkänd, 31 mars 2006

Handledare: Thijs Holleboom

Examinator: Donald F. Ross

Opponent: Katarina Asplund

Sammanfattning

Skriptspråk har under det senaste decenniet fått en ökad spridning, både gällande användare och gällande användningsområden. Från att huvudsakligen ha använts till enklare dagliga administratörsuppgifter används idag skriptspråken inom många områden där tidigare enbart systemspråk var ett alternativ. Denna uppsats undersöker och granskar fem skriptspråk: PHP, Perl, Ruby, Tcl och PostScript. Målet är att undersöka språkens egenskaper och se på skillnader språken emellan. Vi jämför också skriptspråken mot systemspråket C, som dock inte granskas för sig. Vi utför ett test där quicksort-algoritmen används för att sortera ett antal element som läses in från fil. Ett testskript skrivs i varje språk, och dess effektivitet och expressivitet jämförs.

Vi kommer i denna uppsats fram till att definitionen för skriptspråk, och det som skiljer skriptspråk från systemspråk, är skriptspråkens avsaknad av ett separat kompileringssteg. En annan viktig aspekt är skriptspråkens användning av dynamisk typbindning för variabler. De tester vi gjort visar att inget av skriptspråken kan mäta sig med systemspråket C vad gäller exekveringstid, däremot är de bättre gällande expressivitet. Av skriptspråken är Perl det språk som är snabbast och PostScript det språk som är långsammast.

An analysis of five scripting languages - Characteristics and development

Abstract

During the last decade, scripting languages have seen an increase in both number of users and areas of development. Earlier, scripting languages were mainly used for everyday system administration tasks. Nowadays, scripting languages are used in areas where previously only system languages were an alternative. In this dissertation, five scripting languages are compared and reviewed. The languages are: PHP, Perl, Ruby, Tcl, and PostScript. The goal is to examine characteristics of the languages and illustrate the differences between them. We also compare the scripting languages against the system language C, which is not examined in this dissertation. We perform a test where the quicksort algorithm is used to sort a number of elements which are read from file. A test script is written in every language, and their efficiency and expressiveness are compared.

In this dissertation we come to the conclusion that the definition of scripting languages, and the main difference between scripting languages and system languages, is the absence of a separate compiling stage for scripting languages. We also consider the use of dynamic typing regarding variables to be an important aspect of scripting languages. The tests have shown that scripting languages can not compete with the system language C regarding efficiency of execution, but the scripting languages have a higher level of expressiveness. Perl is fastest among the scripting languages and PostScript is the slowest.

Tack till

Tack Thijs Holleboom för all den tid du lagt ner och för den hjälp du gett oss under vårt skrivande av denna uppsats.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

En dators instruktioner kallas maskinkod, och består av ettor och nollor som kan bearbetas direkt av CPU:n. Dessa ettor och nollor är väldigt svåra för en människa att tolka och skriva. För att underlätta och effektivisera skrivandet av datorns instruktioner skapades assemblerspråk, där korta instruktioner assemblerkod motsvarar ett större stycke maskinkod. En likadan utveckling ledde till att de första systemprogramspråken skapades. Ett stycke systemprogramkod motsvarades alltså av ett ännu längre stycke maskinkod än assembler.

För att underlätta återkommande enklare uppgifter skapades de första skriptspråken. Skriptspråken band till en början ihop program genom att ta ett programs utdata och skicka in det som indata till ett annat program. Skriptspråken utvecklades och kom att bli mer avancerade. I takt med denna utveckling utökades skriptspråkens användningsområden.

Idag används skriptspråk inom många av de områden där tidigare enbart systemprogramspråk användes. Undantagen är de fall då kraven på snabb exekvering och låg minnesanvändning är väldigt höga.

I denna uppsats beskrivs och jämförs fem olika skriptspråk. Deras egenskaper, utveckling, syntax, syfte och olika konstruktioner gås igenom. För att jämföra de olika språken utförs ett test i varje språk där quicksort-algoritmen används för att sortera ett antal element. En analys görs också av språkens egenskaper och testresultat för att försöka kategorisera skriptspråken.

1.1 Bakgrund

Vi har valt ut fem språk att undersöka och jämföra. Dessa språk är PHP, Perl, Tcl, Ruby och PostScript. Språken har vi valt efter olika kriterier. Vi bestämde oss för att välja språk som utmärker sig inom något eller några av nedanstående kategorier:

- Popularitet
- Ålder
- Språkets syfte
- Programmeringsparadigm

PHP är det populäraste skriptspråket idag [1]. Språket är även relativt ungt då dess första version släpptes 1995. PHP främsta användningsområde är som tilläggsmodul till webbservrar.

Perl är det näst populäraste skriptspråket för tillfället men är i motsats till PHP betydligt äldre då dess första version kom redan 1986.

Tcl är likt Perl relativt gammalt och började utvecklas 1988 med ett specifikt syfte att vara ett inbäddningsbart kommandospråk.

Ruby valdes för att det är ett helt objektorienterat skriptspråk. Dess snabba utbredning, främst i Asien, gör det också intressant.

PostScript valdes på grund av dess annorlunda användningsområde; att fungera som ett dokumentbeskrivningsspråk. Syntaxen skiljer sig också från många språk då PostScript använder sig av postfixnotation.

Vi hade gärna haft med fler språk i uppsatsen, men för att kunna hålla oss inom de ramar en tiopoängsuppsats begränsar oss till, valde vi att endast behandla dessa fem språk. Gällande det objektorienterade paradigmet var Python ett alternativ, men vi valde istället Ruby, just på grund av att det är helt objektorienterat. Andra språk valdes bort av olika skäl, som t.ex. svårigheter med att hitta tillräckligt med information.

1.2 Syfte

Skriptspråk har under det senaste decenniet växt fram som ett alternativ till de klassiska programspråken inom många områden. I denna uppsats granskar vi fem skriptspråk närmare för att ge en bild av hur dessa skriptspråks egenskaper, syften och utveckling ser ut. Vi vill också se hur väl skriptspråken står sig mot ett systemprogramspråk vad gäller utvecklingstid, expressivitet och effektivitet gällande exekvering.

1.3 Forskningsfrågor

- Vad definierar ett skriptspråk och vad särskiljer ett skriptspråk från de klassiska systemspråken?
- Hur ser de undersökta skriptspråkens utveckling ut?
- Går det klassificera de undersökta skriptspråken i andra grupper än de klassiska paradigmen, och i sådana fall, hur ser en sådan klassificering ut?
- Hur står sig de undersökta skriptspråken effektivitetsmässigt mot klassiska systemspråk?

1.4 Metodologi

För att uppfylla syftet med uppsatsen och försöka besvara forskningsfrågorna har vi valt att göra en litteraturstudie för varje skriptspråk där olika områden som t.ex. utveckling, flödeskontroll och speciella egenskaper beskrivs. Denna litteraturstudie används sedan för en analys av likheter och olikheter mellan skriptspråken. För att jämföra skriptspråkens effektivitet sinsemellan och mot systemspråk utförs ett test av exekveringstid för de fem skriptspråken samt systemspråket C. Testet går ut på att läsa in ett antal element från fil samt sortera dessa element med quicksort-algoritmen. För varje språk sker tidmätning av inläsning, sortering samt den totala exekveringen.

2 Allmänt om skriptspråk

Här ges en introduktion till skriptspråk. Kapitlet behandlar bakgrund, vanliga användningsområden och egenskaper för skriptspråk. Systemprogramspråk tas även upp kortfattat.

2.1 Bakgrund

Ordet skript tros från början komma från filmvärldens script, eller manuskript som vi säger på svenska, som är instruktioner hur skådespelare och kameramän ska uppföra sig vid inspelningen av en film [2] (s.3). Programmering kan sägas vara att skriva program med hjälp av ett programspråk. Skriptning är då när ett skript skrivs med hjälp av ett skriptspråk.

Ett av de största användningsområdena för skriptspråk är inom World Wide Web, som t.ex. ett CGI-skript som jobbar mot en databas. Skriptspråk är dock mycket mer än så och ligger bl.a. som grund för Visual Basic och en del viktiga komponenter i Microsoft Office [2].

Skriptspråk kan ses på som ett lim som limmar ihop olika applikationer och dess funktionalitet. Skriptspråk har designats för andra uppgifter än systemprogramspråk vilket har lett till skillnader i språken. Traditionella programspråk har skapats för att bygga upp datastrukturer från grunden, medan skriptspråk då fungerar som ett lim som använder sig av redan befintliga komponenter. Skriptspråk har oftast dynamisk typbindning för att underlätta kopplingarna mellan komponenter och snabb utveckling. Traditionella programspråk däremot använder statisk typbindning för att minska komplexitet och för att underlätta för kompilatorn att bättre felsöka koden [3].

Skriptspråken har sina grunder i Unix-världen och inom IBM [2]. I början av 80-talet skapades skript för att automatisera kontroll av applikationer som egentligen var avsedda att styras med hjälp av tangentbordsinmatning. Liknande tekniker användes inom Unix-världen för att utveckla verktyg för systemadministration [2].

2.2 Egenskaper för skriptspråk

Dagens skriptspråk har många gemensamma egenskaper, som tillsammans definierar konceptet skriptspråk.

Skriptspråk beskrivs ofta som interpreterade språk [2], men så är inte alltid fallet. Det viktiga är att språket uppför sig som om det interpreteras. Vissa skriptspråk är implementerade som strikt interpreterade, och läser då koden rad för rad utan lookahead eller backtracking och utför en operation så fort ett giltigt nyckelord eller konstruktion påträffas. Unix shell och tidigare versioner av Tcl är exempel på sådana språk. De flesta språk tillämpar dock en hybridteknik som kompilerar skriptet till ett språk, på en lägre nivå, som i sin tur interpreteras. Detta nya skript kan vara en representation av ett parseträd vilket gör att källkoden fortfarande finns tillgänglig i form av parseträdet vid tiden för körning och kan därmed användas för informativ diagnostik vid fel i koden. Angående dessa språk pratas det om kompileringstillfället trots att koden endast delvis kompileras [2].

Med låg overhead och resultat som snabbt syns under utvecklingen är skriptspråken väl anpassade till att hantera ad hoc-lösningar där skriptspråk används för att skarva ihop olika programdelar eller för att på kort tid ta fram ett mindre program [4]. Variabeldeklarationer är ofta valfria och variabler initieras till lämplig typ när de först används. Språken har dock fått ett större behov av användarbestämda variabeldeklarationer, detta på grund av språkens utveckling som gjort det möjligt för språken att användas för allt större applikationer. Ett exempel på ett sådant språk är Perl där variabeldeklaration kan påtvingas genom användandet av `strict`. Antalet datatyper i skriptspråk är ofta begränsat och vanligt förekommande är att allting är strängar som automatiskt konverteras till lämplig typ när sammanhanget kräver det. Även antalet datastrukturer är begränsat, ofta bara till arrayer som både är indexerade och associativa. I de fall då allting är strängar har tal generellt sett ingen storleksbegränsning [2].

Skriptspråk har vanligtvis utökad funktionalitet inom vissa områden. T.ex. har flera språk kraftfull stränghantering med hjälp av reguljära uttryck, som t.ex. Perl, medan andra språk tillhandahåller enkel åtkomst till operativsystemsenheter på låg nivå, som t.ex. Tcl [5].

Skriptspråk upplevs ofta ha en större enkelhet än systemspråk. Denna enkelhet uppnås dock till en kostnad av effektivitet. De första datorerna hade mycket begränsad kapacitet gällande

minne och processorhastighet, och det var då viktigt att applikationer var väldigt resurssnåla [6]. För de applikationer som skriptspråk designats för är effektiviteten ofta inte det viktigaste. Många skript, som t.ex. shell-skript skrivna av en administratör används bara en gång. Andra skript används regelbundet, men kräver inte snabb exekvering då det kanske är viktigare med en snabb utvecklingstakt och möjligheterna att enkelt och snabbt kunna modifiera kod för att tillmötesgå nya krav [2]. Vinsten av utvecklingstakten betalas även den av effektiviteten. Dagens allt snabbare datorer bidrar ytterligare till att effektivitetskraven på språken minskar [3].

2.3 Användningsområden

Skriptning föddes inom administrationen av Unix-system där administratörerna använde sig av skript för att automatisera saker som behövde göras varje dag, som att lägga till nya användare till systemet, eller göra backup på filsystemet [2]. För dessa typer av uppgifter var den viktigaste egenskapen att kunna komma åt underliggande systemanrop snarare än att kunna utföra komplicerade numeriska beräkningar. I och med att många skript skrevs för att endast köras en enda gång var det viktigt med låg overhead. AWK är ett bra exempel på ett språk som har utvecklats för administrationssyften. Det har kraftfulla verktyg för manipulering av strängar, men för att fullt ut fylla systemadministratörernas krav, var AWK-skripten tvungna att bäddas in i shellskript, vilket kunde bli mycket rörigt. Detta problem ledde till utvecklingen av Perl som kombinerade det som kunde göras med AWK- och shellskript tillsammans, plus många andra kraftfulla verktyg, som t.ex. sed, i ett enda språk [2].

World Wide Web är ett av de störst växande områdena för skriptspråk. Webbskriptning handlar i huvudsak om tre områden: formulärhantering, visuella effekter och användarinteraktion, och html-generering [2]. Vid formulärhantering skickas användardata till skriptet på serversidan som då utvärderar datan och utför lämplig åtgärd. Det kan t.ex. vara användarnamn och lösenord för inloggning på en webbplats. Vid visuella effekter och användarinteraktion körs det ett skript på klientsidan, i webbläsaren. Ett vanligt exempel på detta är JavaScript som kan användas för att t.ex. byta bilder eller skapa dynamiska menystrukturer [7]. Vid generering av html körs det ett skript på serversidan, som t.ex. hämtar information från en databas eller fil och sedan genererar html för att kunna visa informationen för en användare på ett strukturerat och lättläst sätt [2].

Det är vanligt att skriptspråk används som ett gränssnitt, eller lim, mellan två separata applikationer. Detta innebär att ett skriptspråk fungerar som en länk mellan de olika applikationerna eller modulerna. För att kunna fungera som lim måste språket kunna starta upp andra program, samla in dess utdata och starta ett nytt program, eventuellt med det första programmets utdata som indata [2] (s.10). Skriptspråk används också som lim för att sätta ihop komponenter som kan vara skrivna i olika språk, ofta systemprogramspråk för bättre prestanda, och som i sig är svåra att sätta ihop. Skriptspråken ger, till viss del tack vare att de använder dynamiskt typbindning, stora möjligheter för att koppla samman komponenter [3].

Dagens allt snabbare datorer har gjort det möjligt att utveckla allt mer komplicerade fristående program, skrivna i skriptspråk. I många av skriptspråken är det även enkelt att skapa grafiska användargränssnitt jämfört med klassiska systemspråk, vilket ger programmerarna ytterligare en anledning att utveckla sina applikationer i skriptspråk. Som ett exempel krävs det i Tcl endast en rad för att skapa en knapp med text på som skriver ut ”hello” vid nedtryckning. Detta att jämföra med C++ som kräver c:a 25 rader för samma jobb [3].

2.4 Systemprogramspråk

Systemprogramspråken utvecklades som ett alternativ till assemblerspråken för att komma ifrån assemblerspråkens låga nivå där i stort sett varje sats representerade en maskininstruktion. Det var mycket svårt att underhålla stora program skrivna i assembler, eftersom programmeraren var tvungen att hålla reda på allt från registerallokering till i vilken ordning anrop av procedurer har skett [3].

I slutet av 50-talet kom de första språken där en kompilator översatte varje sats till sekvenser av binära instruktioner. Bland de första språken var Lisp, Algol och Fortran och med tiden har många andra språk utvecklats som bl.a. C, C++, Pascal och Java. Systemprogramspråken har över assemblerspråken fördelarna att program kan utvecklas mycket snabbare och kan underhållas lättare. Eventuella fördelar som assemblerspråken har i snabbhet kan ofta negeras genom kompilatorernas förmåga att gör optimeringar, både på lokal och på global nivå. Detta ledde till att systemprogramspråken i stort sett helt ersatte assemblerspråken för utveckling av applikationer [3].

3 De fem skriptspråken

Här går vi igenom de valda skriptspråken och beskriver deras utveckling och egenskaper. Vad som gås igenom för varje språk skiljer sig något åt beroende på dess egenskaper och vad vi ansett vara det viktigaste. Nedan definieras de huvudrubriker som är gemensamma för alla språk som tas upp.

- ***Syfte och filosofi:*** *Här tittar vi på varför språket skapats. Har språket ett speciellt syfte, eller har användare upptäckt små fördelar med språket som sedan fått mer utrymme?*
- ***Utveckling:*** *Under utveckling beskrivs språkets utveckling och historia. Vissa språk har utvecklats under lång tid och genomgått många förändringar medan andra språk är i stort sett likadana idag som när de först kom.*
- ***Speciella konstruktioner:*** *Om det finns några speciella konstruktioner som är utmärkande för språket tas de upp här. Det kan t.ex. vara dictionary-stacken i PostScript eller virtuella attribut i Ruby.*
- ***Flödeskontroll:*** *Här gås det igenom hur språket hanterar flödeskontroll som repetition, selektion och sekvenser.*
- ***Typer:*** *Vad finns det för typer i språket, och hur fungerar de?*
- ***Procedurer:*** *Här tas det upp hur språket hanterar procedurer. Hur parametrar tas emot och hur procedurerna definieras.*

3.1 PHP

PHP (PHP: Hypertext Preprocessor) är ett skriptspråk med möjligheter för inbäddning som ofta används för att generera dynamiskt innehåll på webbsidor [8]. PHP kan från och med version 5 kategoriseras att tillhöra både det objektorienterade paradigmet och det imperativa paradigmet, även kallat det procedurella paradigmet som vi i fortsättningen använder i uppsatsen.

3.1.1 Syfte och filosofi

PHP har från början haft som syfte att användas med World Wide Web, men kan även användas för separata applikationer. Det finns i huvudsak tre områden där PHP används:

skriptning på serversidan, kommandoradsskriptning och skripting på klientsidan med grafiskt användargränssnitt.

- Skriptning på serversidan är det mest vanliga och traditionella användningsområdet. En PHP-motor kopplas till en webbserver som sedan genererar HTML som en användare kan se via en vanlig webbläsare. PHP kan integreras med webbservern som en modul, eller köras som en separat cgi-process. Vid integrering med webbservern skapas det inte nya processer vid användning av PHP-motorn [9].
- Vid kommandoradsskriptning behövs varken webbserver eller webbläsare, utan endast en PHP-motor. Detta användande är vanligt för schemalagda skript som ska exekveras regelbundet för t.ex. underhåll på en dator. Kommandoradsskriptning kan även användas för manipulering av text.
- PHP-GTK är ett tillägg till PHP för att skapa grafiska användargränssnitt. PHP är inte optimalt att skriva fönsterapplikationer med, men kan vara användbart om specifika PHP-funktioner vill användas. Här finns det möjligheter att skriva plattformsoberoende applikationer.

En av de viktigaste egenskaperna som PHP besitter är dess omfattande stöd för databashantering vilket visar på dess lämplighet för användande mot webben [8].

3.1.2 Utveckling

Den första versionen av PHP utvecklades av Rasmus Lerdorf 1995. Rasmus var intresserad av hur många som tittade på hans CV på hans hemsida och skrev en samling enkla Perl-skript för att räkna antal nerladdningar av sitt CV [8]. Han kallade dessa skript för Personal Home Page Tools. Lerdorf ville ha mer funktionalitet till sina verktyg och skrev en större implementation i C som hade funktionalitet för kommunikation med databaser. Denna version kallade han PHP/FI, Personal Home Page/Forms Interpreter, och innefattade en del grundläggande funktionalitet som finns kvar i språket än idag. Syntaxen var Perl-liknande men mer begränsad och enklare. Det fanns automatisk tolkning av formulärvariabler och koden kunde skrivas inbäddad i html-dokument. Nu kunde enklare dynamiska sidor utvecklas med verktygen. Lerdorf valde sedan att släppa källkoden fri för att andra användare skulle kunna använda den för att hitta buggar och förbättra verktygen.

1997 kom PHP/FI 2.0. Det hade nu uppstått en kult runt om i hela världen med tusentals användare. C:a 1% (50 000) av världens domäner hade PHP installerat.

Den första versionen som någorlunda liknar dagens PHP var PHP 3.0 som skapades av Andi Gutmans och Zeev Suraski. De började på noll och skrev 1997 om allting då de tyckte att PHP/FI 2.0 ej var kraftfullt nog.

En av de största fördelarna med PHP 3.0 var dess möjligheter till utbyggnad. Massvis med utvecklare upptäckte PHP och bidrog med moduler för utökning av språket. Språket bytte nu namn. Indikationen på att det mest var till för personliga hemsidor togs bort. Akronymen bestod dock, men var nu rekursiv och stod för ”PHP: Hypertext Preprocessor”.

Andi Gutmans och Zeev Suraski började 1998 skriva om PHP:s kärna för att bland annat förbättra prestandan för mer komplicerade applikationer. Den nya kärnan kallades ”Zend Engine” och introducerades i mitten av 1999. PHP 4.0, som baserades på denna nya motor och innehöll en rad nya egenskaper, släpptes officiellt i maj 2000. Nu fanns det bland annat stöd för många fler webbservrar, http-sessioner och säkrare sätt att hantera inmatning från användare.

I juli 2004 släpptes den senaste versionen, PHP 5, som utvecklats för att öka prestandan och möjligheterna ännu mer. En av de största förändringarna är stöd för objektorientering. Tanken har varit att vid en konvertering från PHP 4 till PHP 5 ska så lite som möjligt behöva ändras.

Idag har c:a 20% av världens domäner PHP installerat [8].

3.1.3 Speciella konstruktioner

PHP har en ovanlig konstruktion för flödeskontroll som heter `declare`. `declare` används för att utföra en eller flera funktioner efter n antal kodrader. Detta upprepas sedan efter varje n antal kodrader. Här nedan visas ett exempel på hur `declare` fungerar:

```
<?php
function A(){         //Funktionen som anropas då ett tick
    echo ”A”;         //utförts.
}
register_tick_function(”A”);  //Registrerar funktionen A för
                              // tick-anrop
declare(ticks=2){     //Anger att ticks ska utföras efter
                      //varannan sats.
```

```
    echo ”b”;          //Kod som används för att visa hur
    echo ”c”;          //funktionen A anropas efter varje tick
    echo ”d”;
    echo ”e”;
    echo ”f”;
}
?>
```

I exemplet ovan så registreras funktionen `A` i `register_tick_function`, vilket innebär att funktionen `A` kommer att anropas då `ticks` antal satser har utförts. Sedan följer `declare`-blocket, vilket kan vara en rad, där `ticks` sätts till 2. `while`-loopen som följer innehåller ett par anrop till funktionen `echo`, som producerar utskrifter, samt ett anrop till funktionen `sleep` med parametern 1, vilket innebär att processen vilar i 1 sekund. Vid en körning av programmet så kommer ”bcAdeAfA” att skrivas ut. Det sista A:et skrivs ut då klammerparentesen ”}”, som avslutar blocket, räknas som en sats, vilken tillsammans med den sista `echo`-satsen gör att ytterligare ett A skrivs ut. De satser som finns i funktionerna som registrerats i `register_tick_function`, räknas inte in i de satser som aktiverar anropet. Så `echo`-satsen i funktionen `A` räknas inte med.

Funktionaliteten som ges av `declare` fungerar enbart inom det block som `declare` anger. Detta innebär att om `ticks` sätts till 3 och `declare`-blocket bara innehåller en sats så kommer inte de rader som följer `declare`-blocket att kunna utlösa ett anrop av de registrerade funktionerna.

3.1.4 Flödeskontroll

I PHP finns de vanliga konstruktionerna för flödeskontroll `som if, else, elseif, while, do-while, for` och `switch`, men även en del lite mer ovanliga som `foreach` och `declare` (se avsnitt 3.1.3 för mer information om `declare`).

`if, else, elseif, while, do-while, for` och `switch` fungerar på samma sätt som i C. Det går lika bra att skriva `else if` som `elseif`.

I PHP 4 introducerades `foreach`-konstruktionen, liknande den som redan fanns i Perl och andra språk. Med `foreach` ges det möjlighet att iterera över arrayer och objekt. I PHP 4 är

det bara möjligt att använda `foreach` på arrayer och det genereras ett fel om `foreach` försöks tillämpas på någon annan typ. I PHP 5 är det möjligt att iterera över objekt. Det är då alla publika attribut som det itereras över [8].

3.1.5 Typer

I PHP finns det stöd för åtta olika typer. `boolean`, `integer`, `float` och `string` är skalära typer. `array` och `object` är sammansatta. `resource` och `NULL` är två specialtyper. Vilken typ en variabel tillhör sätts normalt inte av programmeraren utan avgörs av PHP-tolken beroende på i vilket sammanhang variabeln används [8].

`boolean`

En `boolean` kan vara antingen `TRUE` eller `FALSE`. Det är möjligt att explicit konvertera andra typer till en `boolean` med hjälp av `(bool)` eller `(boolean)`, men detta är oftast inte nödvändigt eftersom PHP-tolken själv försöker konvertera värden till `boolean` då det krävs [8]. Dessa värden tolkas som falskt:

- `boolean FALSE`
- `integer 0`
- `float 0.0`
- `string ""` och `"0"`
- `array` med noll element
- `object av en klass` med noll klassvariabler
- specialtypen `NULL`

Alla andra värden tolkas som `TRUE`.

`integer`

En `integer` är ett positivt eller negativt heltal som kan representeras decimalt, oktalt eller hexadecimalt. Storleken på en `integer` är plattformsberoende, men vanligtvis är den övre gränsen runt två miljarder. Det finns inget stöd för `unsigned integer`. Vid division av två `integer` returneras en `float` om resultatet inte är ett heltal [8]. Det sker alltså ingen heltalsdivision som i t.ex. C. För att framtvinga en sådan heltalsdivision går att typkonvertera resultatet explicit till `integer`.

Ex:

```
$tal1 = 7/2;                    // $tal1 är nu 3.5
```

```
$tal2 = (integer) (7/2);       // $tal2 är nu 3
```

float

Storleken på en `float` är plattformsberoende, men ligger vanligtvis omkring 1.8e308.

string

En `string` är en följd av tecken där ett tecken motsvarar en byte, vilket möjliggör 256 stycken olika tecken. Detta medför att PHP i grunden inte har stöd för Unicode. PHP använder sig av en Unicode-konverterare för att lösa detta problem. En `string` har i praktiken ingen storleksbegränsning.

En `string` kan skapas på tre olika sätt: enkla citationstecken, dubbla citationstecken och heredoc-syntax.

1. Ett sätt att skapa en `string` i PHP är med enkla citationstecken. Om ett enkelt citationstecken ska ingå i strängen måste det skrivas som ett escape-tecken och alltså föregås av en backslash: `\'`.
2. En `string` som skapats med dubbla citationstecken fungerar på ungefär samma sätt fast det finns fler escape-tecken som kan användas, som t.ex. nyradstecken, dollartecken och tabb-tecken. Dock behöver inte enkla citationstecken skrivas som escape-tecken. Den viktigaste egenskapen med en dubbelciterad `string` är att variabler kan skrivas direkt i strängen, och dessa utvärderas [8].
   Ex:
   ```
   $string1 = ”Mike”;
   $string2 = ”My name is $string1”;
   ```
   `$string2` kommer nu att innehålla ”My name is Mike”.
3. Vid heredoc-syntax (”<<<”) definieras en identifierare som används för att starta och avsluta strängen. Den avslutande identifieraren måste börja på en ny rads första kolumn, och måste bestå av alfanumeriska tecken eller underscore och får inte börja med en siffra.
   Ex:
   ```
   $string1 = <<<END
       Bosse
   END;
   ```

Heredoc-syntax fungerar på samma sätt som en dubbelciterad string vad gällande escape-tecken och utvärdering av variabler inom strängen, förutom att dubbla citationstecken inte behöver skrivas som escape-tecken [8]. Denna funktionalitet finns även i Ruby, se avsnitt 3.3.4.

`array`

En array är i PHP en mappningstabell, där nyckelvärden mappas till värden. En array kan användas som en lista, hashtabell, dictionary, kollektion, stack eller kö. Det är lätt att simulera en trädstruktur då en array kan ha en annan array som värde [8]. En array skapas med array()-konstruktionen, som tar ett antal kommaseparerade par med nyckelvärde och värde.

Ex:

```
$arr = array(”key1” => 123, ”key2” => 234, ”key3” => 345);
echo $arr[”key2”]; // Skriver ut 234
```

Om nyckelvärdet utesluts för en post sätts nyckelvärdet automatiskt till det största integer-nyckelvärdet som finns i arrayen plus ett (1). Finns det inget integer-nyckelvärde i arrayen blir det nya nyckelvärdet 0 [8].

`object`

Ett `object` är en instans av en klass som är en samling variabler och funktioner som jobbar mot dessa variabler, som i t.ex. C++ och Java (Detta gäller från och med PHP5).

`resource`

En variabel av typen `resource` är ett handtag till en extern resurs. Denna resurs kan vara en fil, en databasuppkoppling eller en bild.

3.1.6 Procedurer

Procedurer, eller funktioner som det heter i PHP definieras med hjälp av nyckelordet `function`.

Ex:

```
function func($arg_1, $arg_2, /* ..., */ $arg_n)
{
    echo "En funktion!\n";
    return $ret_val;
}
```

I en funktion kan all giltig PHP-kod finnas, till och med andra funktioner och klassdefinitioner. I PHP 3 måste en funktion definieras före den används. Från och med PHP 4 är detta inte nödvändigt förutom när definitionen av en funktion finns inom en villkorssats, som t.ex. en `if`–sats, eller inom en annan funktion. Om detta är fallet är det tvunget att skriptet varit inne i villkorssatsen, eller anropat den yttre funktionen där den inre funktionen är definierad [8]. Alla funktioner har tillgång till det globala scopet, även om de är definierade inom en annan funktion eller i en villkorssats. Däremot stöds inte funktionsöverlagring, och det är inte möjligt att omdefiniera redan definierade funktioner. Funktionsnamnen är till skillnad från variabelnamn inte skiftlägeskänsliga även om det är vanlig praxis att behandla dem som om de vore det.

3.2 PostScript

Det här kapitlet behandlar språket PostScript. PostScript är ett dokumentbeskrivningsspråk som är konstruerat speciellt för att kunna beskriva komplex grafisk information på ett hårdvaruoberoende sätt. Språket skrivs i postfix format vilket innebär att parametrar är placerade framför de operatorer och procedurer som ska använda dem. PostScript är svårt att kategorisera enligt de vanliga paradigmen. Användandet av procedurer i PostScript medför att språket kan kategoriseras att tillhöra det procedurella paradigmet.

3.2.1 Syfte och filosofi

Syftet med PostScript är att det på ett effektivt sätt ska kunna beskriva och förmedla komplex grafisk information för utskrift på skrivare. PostScript skiljer sig mycket från de fyra övriga skriptspråken med sitt områdesspecifika syfte. Språket är konstruerat så att den grafiska informationen ska kunna beskrivas på ett hårdvaruoberoende sätt [10].

3.2.2 Utveckling

Adobe Systems Incorporated tog i början av 1980-talet fram en modell för att beskriva komplexa bilder. PostScript-språket skapades som en följd av denna modell. Språket gör det möjligt att beskriva grafiska bilder och förmedla dem till PostScriptbaserade enheter som t.ex. skrivare och applikationer som tolkar PostScript [11].

Sedan PostScript skapades så har språket utvecklats som en följd av ny och förbättrad teknologi för att behandla och skapa bilder. Språket har utvecklats genom att paket med utökningar har lagts till språket. Paketen har lagts till språket i grupper som kallas LanguageLevels, vilket kan översättas till språknivåer. När en språknivå läggs till språket måste det innehålla stöd för alla de språknivåerna som tidigare lagts till, utöver den nya funktionaliteten som den innehåller. En PostScript-interpretator som stödjer PostScript upp till språknivå 3 måste därför innehålla stöd för all funktionalitet som finns i språknivåerna 2 och 1. Utöver de officiella språknivåerna så kan en PostScript-enhet även stödja specifika språktillägg [11].

3.2.3 Speciella konstruktioner

Dictionary

PostScript innehåller ett par olika stackar som används för att lagra undan objekt. En av de stackarna är dictionary-stacken, där enbart dictionaryobjekt kan lagras. Ett dictionaryobjekt är ett sammansatt objekt som är uppbyggt likt de associativa arrayaer eller hashtabeller som finns i PHP, Perl, Tcl och Ruby. Dictionaryobjekten innehåller poster av ordnade par, där ett par består av ett nyckelobjekt och ett värdeobjekt [11].

Ett objekt kan i PostScript bara lagras på två olika ställen, antingen på operandstacken eller i ett dictionaryobjekt. Interpretatorn lagrar objekt på operandstacken som ska användas eller har returnerats av procedurer eller operatorer. Så länge programmeraren vet vad som finns på operandstacken så kan objekten där användas direkt utan att de behöver lagras på något annat sätt. Om ett objekt som enbart existerar på operandstacken tas bort därifrån, utan att vara lagrat i ett dictonaryobjekt, finns det inget sätt att få tillbaka det igen [10].

Dictionarystacken används när interpretatorn ska kolla upp ett exekverbart namnobjekt. Ett exekverbart namnobjekt är operatorer eller procedurer, och dessa markeras som sådana i skripten genom att de inte har en ”/” framför sig. En ”/” markerar att namnet tillhör en literal. Exekverbara namnobjekt lagras i dictionaryobjekt med namnet på operatorn eller proceduren som nyckel, och som värde har de ett exekverbart objekt. Dictionarystacken genomsöks efter det exekverbara namnobjektet, från det översta dictionaryobjektet till det nedersta. Om nyckeln hittas exekveras värdeobjektet som det är associerat med direkt utan att det placeras på operandstacken [10].

Dictionarystacken innehåller då interpreteringen börjar tre stycken dictionarys, i språknivå 1 så var det enbart två stycken, som inte kan tas bort från stacken [11]. En av dessa dictionarys, systemdict, innehåller operatorerna som är inbyggda i PostScript. PostScript-program kan lägga till nya dictionaryobjekt på dictionarystacken för att där lagra nya objekt. De inbyggda operatorerna kan överlagras i PostScript genom att ett exekverbart objekt läggs till i ett dictionaryobjekt som ligger högre upp i dictionarystacken än systemdict, tillsammans med samma namnobjekt som för operatorn som ska överlagras. När interpretatorn söker igenom dictionarystacken efter operatorn så kommer den nya operatorn påträffas före den inbyggda operatorn och därför kommer den nya operatorns värdeobjekt exekveras. Den gamla operatorn kommer inte att utföras så länge den nya versionen finns högre upp i dictionarystacken [12].

3.2.4 Flödeskontroll

Selektion

I PostScript så används operatorer för att kontrollera flödet i ett program [11]. Operatorerna använder sig utav procedurer som val för vad som ska göras. Här är ett enkelt exempel:

```
a {b} if
```

PostScript skrivs som, tidigare nämnt, i postfix-format. Interpretatorn läser i det här exemplet `a,` som utvärderas och dess värde läggs på operandstacken. Sedan läses proceduren `{b}` som också läggs till operandstacken. `if`–operatorn tar två operander, ett booleskt objekt och ett procedur-objekt. De båda operanderna tas bort från operandstacken innan operatorn utförs. `if`–operatorn utför proceduren, i det här fallet `{b},` om den booleska operanden, här `a,` har värdet `true.` Om värdet på den booleska operanden inte är `true` så utförs inte proceduren utan interpretatorn fortsätter vidare. PostScript innehåller även operatorn `ifelse` som fungerar på ett liknande sätt som `if,` fast istället för en procedur så tar `ifelse` två procedurer. Om det booleska uttrycket är `true` så utförs den första proceduren som med `if`–operatorn men om det istället är `false` så utförs den andra proceduren [11].

Repetition

PostScript innehåller flera kontrollstrukturer för att styra programflödet iterativt, några av dessa strukturer är: `for`, `forall`, `loop` och `repeat` [11]. `for`-operatorn tar fyra operander, tre tal och en procedur. Det första talet sätter ett initieringsvärde på en räknare, det andra talet

säger hur mycket räknaren ska ökas på efter varje iteration och det tredje talet sätter gränsen för när iterationen ska avbrytas. Den fjärde operanden, proceduren, innehåller den kod som ska utföras under varje iteration. `forall`-operatorn tar två operander där den andra operanden alltid är en procedur som innehåller den kod som ska utföras under varje iteration. Den första operanden kan vara av fyra olika slag: array, packedarray, dictionary eller string. Beroende på vad för typ operanden är av utförs olika saker. För ett dictionaryobjekt hämtas både nyckel och värde ur dictionaryn och läggs på operandstacken medan för de andra tre typerna hämtas bara ett värde och läggs på operandstacken [11]. För alla de olika typerna itererar `forall` igenom alla element i den första operanden och för varje iteration så utförs proceduren. `loop`-operatorn tar bara emot en operand och den måste vara en procedur. Proceduren utförs om och om igen tills den bryter iterationen med hjälp av `exit`-operatorn eller `stop`-operatorn. Alla iterationsoperatorer kan avbrytas genom att de i sina procedurer anropar `exit`-operatorn eller `stop`-operatorn. `repeat`-operatorn liknar `loop`-operatorn men tar en extra operand som talar om hur många gånger proceduren ska utföras.

3.2.5 Typer

I PostScript är all data som ett program kan komma åt objekt. Alla objekt har ett par saker gemensamt: de har en typ, ett antal egenskaper och ett värde. Objekt kan delas in i två kategorier av typer: enkla och sammansatta. Exempel på enkla typerna är: `boolean`, `integer`, `name` och `operator`. Sammansatta objekt kan t.ex. vara av typerna: `array`, `dictionary` och `string` [11]. Objekt i PostScript har alla en och samma fasta storlek, vilket medför att det är enkelt att utföra operationer på dem [10]. Objekt som har en enkel typ lagrar sitt värde inom sin fasta storlek, sammansatta objekt lagrar istället en pekare till sitt värde. Sammansatta och enkla objekt skiljer sig åt i, bland annat, vad som händer när de kopieras. Om ett enkelt objekt kopieras med någon kopieringsoperator så skapas ett nytt objekt med samma värde, typ och attribut som det kopierade objektet [11]. Om istället ett sammansatt objekt kopieras med operatorn `dup` så är det originalobjektets pekare som kopieras och inte det utpekade värdet. En sådan kopiering medför att originalobjektet och det nya objektet delar värde eftersom båda objektens pekare pekar på samma värde. Förändras värdet på det nya objektet så förändras det även för originalobjektet. För att ett sammansatt objekt ska kopieras utan att dela data med originalobjektet, måste först ett nytt objekt skapas av rätt typ. Sedan används operatorn `copy` för att kopiera originalobjektets data till det nya objektet.

Ett arrayobjekt i PostScript är en grupp objekt i följd. Åtkomst till objekten i arrayen sker genom objektets index i arrayen. I PostScript-arrayer kan objekt med olika typer lagras. Indexeringen i arrayen är samma som i språk som C++ och java, första platsen i arrayen har index 0 och objekt nummer n har index n-1 [11]. En array är en sammansatt typ.

Namntypen tillhör kategorin enkla typer. Ett namnobjekt har inget värde, däremot så kan de i ett dictionaryobjekt användas som nyckel vilket gör att namnobjektet förknippas med det objekt som är värde i nyckel/värde-paret i dictionaryobjektet [11]. Om två namnobjekt använder samma sträng av tecken som definition så är de två objekten kopior av varandra.

Operatorobjekt är de inbyggda operationer som finns i PostScript. De finns som standard i systemdict-dictionaryobjektet, som värden i nyckel/värde-paren. Operatorobjekten är associerade med namnobjekt, t.ex. add operatorn är associerad med namnobjektet add.

3.2.6 Procedurer

I PostScript så är en procedur ett exekverbart arrayobjekt eller ett exekverbart packedarrayobjekt. Packedarraytypen liknar arraytypen men är mer kompakt, och är skapad för att kunna användas som procedur [11]. En procedur i PostScript kan inte definiera sina inparametrar. De parametrar som en procedur behöver måste finnas på operandstacken innan proceduren anropas. Eftersom det inte går att definiera parametrar till procedurerna och eftersom parametrarna hämtas av proceduren själv från operandstacken så sker ingen kontroll av vad det är för typ av parametrar som proceduren hämtar. Om sådan kontroll önskas så får den ske innan proceduren anropas eller inne i själva proceduren. Det finns operatorer i PostScript som kan användas för att kontrollera typen på ett objekt. Stacken kontrolleras inte efter objekt när en procedur anropas, vilket medför att en procedur som behöver hämta två parametrar från operandstacken kan anropas när operandstacken är tom, och detta medför ett stackunderflow-error i interpretatorn [11]. Om en grupp objekt har { } runt sig så talar det om för interpretatorn att den gruppen är en procedur [12]. En procedur behöver inte vara associerat med något namnobjekt men om så är fallet så går det bara att komma åt proceduren genom operandstacken.

3.3 Ruby

Ruby är ett helt objektorienterat skriptspråk med bra funktionalitet för bland annat hantering av textfiler och systemadministrationsuppgifter. Syntaxen är enkel och inspirerad av bland annat Eiffel och Ada.

3.3.1 Syfte och filosofi

Syftet med Ruby är att det ska vara lätt att programmera i. Designfilosofin för Ruby är ”The Principal of Least Surprise”. Detta innebär att Ruby ska fungera som programmeraren förväntar sig att det ska fungera till skillnad från t.ex. Perl, där inbyggda operatorer implicit används. Ruby ska genom sin språkdesign leda programmeraren till att skriva läsbar och fungerande kod. Ruby är tänkt att vara ett helt objektorienterat skriptspråk som är lätt att använda [13].

I Ruby finns det tilläggsmoduler som gör det möjligt för Ruby att interagera med webbservern Apache, på samma sätt som t.ex. PHP. I och med detta har Ruby fått ytterligare ett syfte i form av att generera html-dokument vid utveckling av webbapplikationer.

3.3.2 Utveckling

Ruby utvecklades 1993 av Yukihiro Matsumoto, även kallad Matz. Matz kunde sedan tidigare många språk. Han ville ha ett språk som var mer kraftfullt än Perl och mer objektorienterat än Python. Enligt Matz låg objektorienteringen i Python mer ovanpå språket än att det var en del av det. Matz sökte efter ett helt objektorienterat skriptspråk som var lätt att använda, men hittade inget som föll honom i smaken. Han bestämde sig då för att skapa ett nytt skriptspråk. Resultatet blev Ruby. Ett helt objektorienterat skriptspråk där han tog med funktionalitet som han tyckte var bra, som iteratorer, undantagshantering och garbage collection. Han implementerade också Perls egenskaper i ett klassbibliotek. Sedan dess har Ruby växt markant och idag är Ruby vanligare än Python i Japan och delar av Asien [14].

Det har även utvecklats tilläggsmoduler för att Ruby ska kunna interagera med webbservern Apache. Några exempel på dessa är eruby, erb och mod_ruby. Med dessa tillägg kan Rubykod inbäddas i html-dokument på serversidan, vilket ger samma möjligheter vid utveckling av hemsidesystem som ASP, JSP och PHP.

Ruby skrevs ursprungligen för POSIX-system, för att kunna utnyttja systemanrop och bibliotek i olika UNIX-miljöer, men har senare portats till andra plattformar som Windows och Mac OS X [15].

3.3.3 Flödeskontroll

I Ruby finns alla de vanliga flödeskontrollerna, så som `if`–satser och `while`–loopar [15]. Det används inga klammerparanteser utan dessa satsernas block avslutas med ordet end. Till skillnad från t.ex. PHP och Perl, behöver vanliga satser inte avslutas med semikolon. Det räcker med att varje sats ligger på en egen rad. Detta bidrar till Rubys renhet som språk. Genom att sätta en backslash sist på en rad kan en sats sträckas sig över flera rader, som i t.ex. Bash.

Ruby har förutom de vanliga flödeskontrollerna även `until` och `unless` som är negeringar av `while` och `if` [16]. `until` och `unless` finns även i Perl, se avsnitt 3.5.4.

Iteratorer

Istället för fler primitiva loop-konstruktioner använder sig Ruby av iteratorer som är inbyggda i olika klasser. Rubys iteratorer hjälper till att dölja den underliggande implementationen [15].

Rubys variant av `for` skiljer sig något mot hur `for` normalt fungerar i andra språk. I ruby är `for` implementerat i form av en iterator som itererar över ett objekt som representerar en mängd. Enda kravet är att objektet har metoden `each`.

Ex:

```
for song in songlist
    song.play
end
```

Detta översätter Ruby såhär:

```
songlist.each do |song|
    song.play
end
```

Funktionen each finns inbyggd i arrayer och det går även att iterera över en mängd tal (1..10).

Det finns fem olika sätt att avbryta eller ändra flödet i iterationer. Det första, `break`, fungerar precis som det gör i C och hoppar helt ur iterationen. Sedan finns `next` som hoppar till början av nästa iteration, på samma sätt som C:s `continue`. Det tredje sättet att avbryta en iteration är `redo` som hoppar tillbaks till början av pågående iteration. `return` inte bara avbryter iterationen utan också metoden i vilken iterationen finns, som i C [16]. Slutligen kan `retry` användas för att starta om iterationen helt från början [15].

Case

Ruby har också C:s motsvarighet till `switch`, fast något mer avancerad. I Ruby heter denna konstruktion `case`. I stället för, som i C, att varje fall endast kan baseras på ett värde kan ett fall i Ruby vara ett intervall:

```
case input #Villkorsvariabel
  when 1   #Om input är 1
    func1  #Anropa func1
  when 2..9    #Om input är mellan intervallet 2-9
    func2(input)   #Anropa func2 med input som argument
  else     #Annars
    exitProg   #Anropa exitProg
end        #Slut på case-satsen
```

`Case`-satsen returnerar värdet på det uttryck som exekveras och det går alltså att tilldela en variabel värdet av `case`-satsen.

De flesta satser i Ruby returnerar ett resultat av det som utfördes. Detta resultat kan sedan användas som villkor i t.ex. `if`- eller `while`-satser. En tilldelningssats returnerar värdet som tilldelats [15].

3.3.4 Typer

Då Ruby är ett helt objektorienterat språk är allting objekt. De klassiska primitiva typerna representeras alla av objekt med specifika egenskaper. Vilken typ ett objekt tillhör definieras inte av dess klass i Ruby. Dess typ avgörs istället av objektets förmågor och vilka egenskaper

det besitter. Detta brukar i Ruby-sammanhang kallas för duck typing. *”If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck*” [15].

För att särskilja instansvariabler och klassvariabler från övriga variabler måste klassvariablernas namn inledas med ett ”@@” och instansvariabler med ”@”.

Tal

Ruby har stöd för heltal och flyttal. Heltalen kan vara hur stora som helst och begränsas endast av hur mycket ledigt minne som finns tillgängligt [15].

Vid beräkningar av tal som lästs in som strängar är det tvunget att först implicit konvertera talsträngarna till lämplig taltyp [15].

Strängar

Normalt innehåller en sträng i Ruby tecken som går att skriva ut, men det är inte ett krav. En sträng kan också bestå av binär data. Ofta skapas strängar genom ett antal tecken som omsluts av antingen enkla eller dubbla citationstecken. Strängar som omsluts av dubbla citationstecken har fler specialtecken (`\n` o.s.v.) än strängar som omsluts av enkla citationstecken där endast `\\` blir `\` och `\'` blir `'` [15].

Genom att använda %q eller %Q går det att själv välja tecken som ska definiera början och slutet av en sträng. %q har samma egenskaper som enkla citationstecken och %Q som dubbla citationstecken.

Ex:

```
%q!En sträng som omsluts av enkla citationstecken!
```

Om `[`, `{`, `(` eller `<` används som avskiljare fungerar det, förutom med samma tecken, med motsvarande stängningstecken, alltså `]`, `}`, `)` eller `>`.

Ett tredje sätt att skapa strängar är med vad som kallas *here documents*, som fungerar som PHP:s heredoc-syntax, se avsnitt 3.1.5.

Ex:

```
string = <<END_OF_STRING
    Här följer då en sträng som sträcker
```

```
    sig över flera rader och avslutas när
    nästa förekomst av det som står efter
    '<<' upptäcks.
END_OF_STRING
```

Det är möjligt att byta ut vilket Ruby-kodstycke som helst till dess strängvärde genom att omsluta koden med `#{` och `}`. Detta fungerar enbart i en sträng. Om det som ska utvärderas till ett strängvärde endast är en variabel kan klammerparenteserna uteslutas.
Ex:

```
myVar = "123#{5*9}678"
print myVar           # ger utskriften ”12345678”
```

Ruby har en funktion, `eval`, som utvärderar strängar som Ruby-kod.
Ex:

```
kodstring = "print \"hello\""
eval kodstring            # Ger utskriften ”hello”
```

Typkonvertering

I Ruby finns det ett koncept som innebär att ett objekt har möjligheten att konvertera sig själv till ett objekt av en annan klass. Detta kallas för conversion protocols. Detta koncept finns implementerat i tre former [15].

1. Den första formen är metoder som `to_s` (konverterar till sträng) eller `to_i` (konverterar till integer). Dessa är inte särskilt strikta. Om ett objekt kan representeras någorlunda i form av en sträng är det sannolikt att objektet har en `to_s`–metod.
2. Den andra typen av conversion protocol är metoder som `to_str` och `to_int`. Dessa är mycket mer strikta och implemeteras bara då objektet naturligt kan användas på ställen som en integer eller string ska användas.
3. Den tredje typen kallas numerisk tvångskonvertering (numeric coercion) och handlar om hur Ruby hanterar beräkningar av tal.

   Ex: `1+2`

Ruby anropar här metoden `+` på objektet `1` och skickar med fixnum-objektet `2` som inparameter.

Ex: `1+2.3`

Samma `+`–metod tar nu emot float-objektet `2.3` som inparameter. Det är här den numeriska tvångskonverteringen ingriper och anropar metoden `coerce`. `coerce` tar in två tal, en mottagare och en parameter, som inparametrar och returnerar en array som innehåller representationer av de två talen. `coerce`–metoden garanterar nu att de två talen är av samma klass och kan nu adderas, multipliceras, jämföras eller vad som nu ska göras med dem [15].

3.3.5 Procedurer

I Ruby kallas procedurer för metoder. En metod kan vara antingen instansmetod eller klassmetod. En klassmetod kan anropas även om det inte finns något instansierat objekt av klassen [17].

Speciella tecken

Ofta används `?`, `!` och `=` som sista tecken i metodnamnet för att visa att detta är en metod med en viss speciell egenskap. `?` indikerar att metoden är en boolesk frågemetod. ! säger att det är en metod som t.ex. modifierar mottagaren eller har sidoeffekter. T.ex. har `String` både en metod `chop` och en metod `chop!`. Den förstnämnda returnerar en modifierad strängkopia medan den andra modifierar den inskickade strängen. En metod med `=` som sista tecken visar på att metoden tilldelar en variabel ett värde, som setters i Java. Metoden kan nu anropas som en vanlig tilldelning, se avsnitt 3.3.6 om virtuella attribut [15].

Argument

Metoders argument kan göras valfria genom att de tilldelas default-värden i metodhuvudet:

```
def minMetod(arg1=”apelsin”, arg2=”banan”, arg3=”kiwi”)
```

För att kunna ha ett varierande antal argument till en metod sätts det en asterisk framför namnet på det sista argumentet i funktionshuvudet. Det argumentet kommer då att innehålla resterande argument som skickas till metoden:

```
def minMetod(arg1, *rest)
```

`rest` kommer nu vara en array som innehåller alla argument utom det första [15].

Det finns även möjlighet att göra precis tvärtom. Alltså att en array som skickas som argument tas emot i flera variabler. Detta görs genom att sätta en asterisk framför parametern som ska skickas som argument vid metodanropet [15].

Returvärde

Alla metoder i Ruby returnerar ett värde. Det som returneras är resultatet av den sista satsen som exekverades i metoden. Det finns en `return`-sats i Ruby som avlutar exekvering av pågående metod. `return` returnerar värdet av dess argument. Vanligt är att `return` utesluts om det inte behövs [15].

Namnlösa metoder

Ett block med kod kan skickas med som argument till metoder för att kunna anropas vid lämpligt tillfälle. En variabel tilldelas den medskickade koden som sedan kan skickas med till metoden `Proc#call` som anropar kodblocket. Detta kan jämföras med lambda-funktioner i Lisp [15].

Instans- och klassmetoder

I Ruby kan det skapas instansmetoder som tillhör ett specifikt objekt av en klass. Dessa kallas även singeltonmetoder. Metoden definieras utanför klassen och läggs till på det instansierade objektet.

Ex:

```
class Hej                      # Definierar en klass Hej.
end
objekt1 = Hej.new              # Skapar instans av klassen Hej.
def objekt1.enMetod            # Definierar en singeltonmetod enMetod
    print ”en singeltonmetod” # för objekt1.
end
objekt1.enMetod                # Anropar singeltonmetoden på objekt1.
```

Klassmetoder skapas genom att vid definitionen av metoden i klassen skriva klassens namn före metodnamnet. Denna metod kan sedan anropas utan någon instans av klassen.

Ex:

```
class Hej                  # Definierar en klass Hej
    def Hej.enKlassmetod  # Definierar en klassmetod enKlassmetod
        print ”Jag är en klassmetod!”
    end
end
Hej.enKlassmetod           # Anropar klassmetoden
```

3.3.6 Speciella konstruktioner

Virtuella attribut

Tidigare nämndes hur `=` användes för att skapa set-metoder. Detta kan användas till virtuella attribut [15]. Med ett virtuellt attribut ser det utifrån objektet ut som det finns ett attribut, men det är i själva verket metoder som anropas. Låt oss säga att en klass `Person` har ett attribut `length_in_centimeters`. Ett virtuellt attribut length_in_inches kan då skapas för att det ska se ut som det även finns ett sådant attribut.

Ex:

```
class Person
    def initialize   #initieringsmetod, anropas när objekt skapas
        @length_in_centimeters = 180
    end
    def length_in_inches      #Det virtuella attributet
        @length_in_centimeters / 2.5
    end
    #Tilldelningsfunktion för det virtuella attributet
    def length_in_inches=(new_length_in_inches)
        @length_in_centimeters = new_length_in_inches*2.5
    end
    def length_in_centimeters
        @length_in_centimeters #Det ”riktiga” attributet
    end
end
```

Nu är det dolt för utomstående att det egentligen inte finns något attribut `length_in_inches`.

3.4 Tcl/Tk

Tcl (Tool command language) är ett enkelt men kraftfullt plattformsoberoende open source-skriptspråk. Tk (Tool kit) är ett högnivå-verktygspaket för att bygga grafiska gränssnitt.

3.4.1 Syfte och filosofi

Tcl skapdes för att styra andra applikationer och det fanns tre huvudsakliga mål som skulle uppfyllas vid utvecklingen [18]:

1. Språket skulle vara utökningsbart för att applikationerna lätt skulle kunna lägga till sina egenskaper till Tcl:s grundläggande funktionalitet.
2. Det skulle vara enkelt och allmänt, så att språket inte kommer att begränsa applikationers egenskaper.
3. Språket skulle ha ett bra stöd för samverkan då huvudsyftet skulle vara att limma ihop moduler [18].

3.4.2 Utveckling

Tcl:s skapare, John Ousterhout jobbade i början av 80-talet på University of California vid Berkeley med interaktiva designverktyg för integrerade kretsar. Dessa verktyg behövde ett kommandospråk. Grafiska användargränssnitt var inte vanligt på den tiden, utan programmen styrdes genom textkommandon. Eftersom fokus låg på själva designverktygen blev de kommandospråk inte särskilt väl utvecklade eller generella. Detta blev efter ett tag både jobbigt och pinsamt, tyckte Ousterhout [18].

Tcl

Hösten 1987 fick Ousterhout idén till ett inbäddningsbart kommandospråk. Tanken var att lägga lite extra kraft på ett bra interpreterat språk i form av ett bibliotekspaket som sedan skulle kunna användas till många olika applikationer. Språket skulle bestå av ett antal grundläggande beståndsdelar så som variabler, kontrollstrukturer och procedurer. Varje applikation som sedan använde språket skulle lägga till sin egen funktionalitet som en utökning så att språket kunde användas för att styra applikationen. När nu syftet med språket var klart kom han på namnet Tcl, Tool command language [18].

1988 började Ousterhout implementera språket. Han arbetade nu inte längre med designverktyg för integrerade kretsar och arbetet med Tcl var därför nu mera av eget akademiskt intresse.

Tk

Under 80-talet blev användargränssnitt mer och mer populära och Ousterhout var intresserad av detta område. Han hade lagt märke till de flesta användargränssnitts komplexitet. Han var orolig att små forskargrupper med begränsade resurser, som de han själv varit del av, inte skulle ha möjlighet att utveckla denna nya typ av interaktiva system. De projekt som funnits inom området hade varit mycket omfattande och krävt stora investeringar [18].

Ousterhouts tanke var nu att bygga användargränssnitt av återanvändbara komponenter som innehöll den största komplexiteten. För att detta skulle fungera krävdes en kraftfull och flexibel mekanism för att integrera komponenterna. Han gjorde ett test där han skapade ett antal grafiska komponenter som en utökning till Tcl, och använde sedan Tcl för att sätta ihop dessa grafiska komponenter till ett användargränssnitt. Denna utökning kallade han Tk [18].

Tcl/Tk sprids

1990 presenterade Ousterhout en artikel om Tcl på USENIX-konferensen. Många blev nyfikna på Tcl och bad om att få en kopia av Tcl. Ousterhout bestämde sig då för att släppa källkoden fri på Berkeleys FTP, och Tcl började spridas ryktesvägen via Internet.

Don Libes vid National Institute of Standards and technology var på USENIX-konferensen och lyssnade på Ousterhout. Han hade länge velat utveckla ett program för automatisering av interaktiva Unix-applikationer. Efter att ha lyssnat på Ousterhout laddade Libes ner Tcl och skrev en av de mest kända Tcl-applikationerna, Expect på bara tre veckor [18]. Expect är ett administrationsverktyg för automatisering av bl.a. Unix-kommandon utan grafiskt användargränssnitt [19].

1993 hade antalet användare av Tcl/Tk växt till flera tiotusentals användare. En av anledningarna till tillväxten var dess enkelhet i att skapa grafiska användargränssnitt under Unix. Andra alternativ som Motif-verktygen i C var mycket mer komplicerade, men hade ändå mindre funktionalitet. Ett grafiskt användargränssnitt kunde i Tcl/Tk skapas med fem till tio gånger mindre arbete än i Motif [18].

En annan faktor för Tcl/Tk:s popularitet var dess möjligheter till inbäddning. Det råder en ständig diskussion om vilken av dessa faktorer som är viktigast. Vissa säger att Tk är enda

anledningen till att de använder Tcl, medan andra tycker att Tcl som inbäddat skriptspråk, ofta utan Tk, är det de bryr sig om [18].

Utökningar

När språket spreds i början av 90-talet bildades en grupp som bidrog med utökningar och support till nya användare. Mark Diekhans och Karl Lehenbauer var med i denna grupp och skrev TclX som var en av de första Tcl-utökningarna. TclX hade bl.a. funktionalitet som filhantering och tid- och datummanipulering. TclX visade sig innehålla så många viktiga verktyg att många har lagts till i standardutgåvorna av Tcl [18].

Ousterhout fick många förslag om funktionalitet som borde ingå i kommande versioner av Tcl/Tk. Årligen hölls det en Tcl-konferens där Ousterhout ägnade en session att diskutera förslagen på ny funktionalitet. Åhörarna fick sedan rösta på vad de ville ha med i kommande version. Resultatet av dessa omröstningar tog sedan Ousterhout i beaktning när han bestämde vad som skulle ingå i nästkommande version.

Tcl och Sun

1994 fick Ousterhout ett erbjudande om att fortsätta utveckling av Tcl vid Sun Microsystems laboratorier. Många oroade sig nu för att Tcl inte längre skulle vara ett öppen källkodprojekt, men ett av kraven Ousterhout ställde vid anställningen hos Sun Microsystems var att Tcl-kärnan och Tk-biblioteken fick fortsätta att distribueras som öppen källkod. Nu fick han chansen att utveckla Tcl till ett universellt skriptspråk för Internet. Fram tills nu hade Ousterhout skrivit nästan varenda rad kod i Tcl själv. Nu övergick ansvaret för koden mer och mer till anställda i Suns Tcl-grupp och det utvecklades versioner av Tcl för Windows och Macintosh. Stöd för socketprogrammering lades till så att Tcl enkelt kunde användas till olika nätverkslösningar. Även en bytekodkompilator utvecklades som kunde snabba upp exekveringen upp till tio gånger. Idag sker mer än två tredjedelar av alla nerladdningar av Tcl till windowsmaskiner [18].

Under denna tid växte antalet Tcl-utvecklare oerhört. 1997 fanns det flera hundra tusen utvecklare. Antalet nerladdningar ökade från 2 000 per vecka 1995 till över 10 000 per vecka [18].

Scriptics

1997 lämnade Ousterhout Sun Microsystem och tog Tcl med sig. Tillsammans med Sarah Daniels grundade han Scriptics. Efter en månad hade halva Suns Tcl-grupp börjat på Scriptics och de utvecklade TclPro som var en samling utvecklingsverktyg [18].

Scriptics fortsatte att, precis som Sun hade gjort, släppa fria Tcl-distributioner. 1999 släpptes Tcl/Tk 8.1 som bl.a. innehöll stöd för Unicode, trådsäkerhet och ett helt nytt paket för reguljära uttryck.

Scriptics döptes i maj, 2000 om till Ajuba Solutions. Företaget fokuserade mest på Tcl-baserade XML-teknologier och blev sedermera en del av företaget Interwoven. Interwoven var dock inte intresserade av företagets öppen-källkodsfilosofi, utan var ute efter XML-teknologierna. I oktober 2000 grundades Tcl Core Team för att driva utvecklingen av Tcl. Tcl Core Team är inte bundet till någon speciell organisation [18].

3.4.3 Speciella konstruktioner

expr

Tcl har ett inbyggt kommando, `expr`, som utvärderar resultatet av de argument som skickats med som ett tcl-uttryck [20].

Ex:

```
expr 8.2 + 3 #utvärderas till 11.2
```

Resultatet är i stort sett alltid ett numeriskt värde. Denna konstruktion måste användas vid t.ex. numeriska beräkningar. De operatorer som tillåts i ett expr är en delmängd av de operatorer som tillåts i uttryck i språket C, och de har samma betydelse och prioritetsordning som i C. `expr` kan även användas för jämförelse av t.ex. strängar, då med någon av operatorerna `ne` eller `eq` mellan de strängar som ska jämföras.

set

I Tcl används inte `=` som tilldelningsoperator. Tilldelning sker i Tcl istället med hjälp av proceduren `set`. `set` tar ett variabelnamn som inargument, och eventuellt ett värde. Specificeras inte värdet händer ingenting förutom att värdet av variabeln returneras. Existerar inte variabeln innan set anropas skapas den [20].

Ex:

```
set var 5   # Skapar variabeln var och tilldelar
            # den värdet 5 som också returneras.
set var  # Returnerar variabeln var:s värde, alltså 5.
```

Växlar

Tcl använder sig av växlar i sin syntax, liknande bashkommandon, för att t.ex. ta in argument vid skapandet av komponenter.

Exempel från [3]:

```
button .b –text Hello! –font (Times 16) –command {puts hello}
```

Denna rad kod skapar en knapp med texten ”Hello!” i typsnittet Times, storlek 16 som utför kommandot `puts hello` när knappen trycks in. De attribut som inte sätts med hjälp av en växel tilldelas istället defaultvärden [3].

3.4.4 Flödeskontroll

I Tcl finns de vanliga konstruktionerna för flödeskontroll som `if-else`, `switch`, `while`, `for`, `foreach`, och `switch`. Flödeskontrollsatser är i Tcl endast kommandon som allt annat. Villkorsdelen måste omslutas av klammerparenteser vilket förhindrar utvärdering av den när den först påträffas. Flödeskontrollsprocedurer anropar sedan internt funktionen `expr` för att utvärdera villkoret [2].

Repetition

En `while`-sats är utformad ungefär som i C och ser ut så här:

```
while test body
```

Ex:

```
while {$p > 0} {
  set result [expr $result*$base]
  set p [expr $p-1]
}
```

När interpretatorn träffar på hakparenteser anropas interpretatorn rekursivt för att tolka det som finns inom hakparenteserna. Det sista som utvärderas inom hakparenteserna returneras och ersätter hakparenteserna och dess innehåll [2].

Tcl:s `for` är även den lik dess motsvarighet i C och ser ut så här:

```
for start test next body
```

Ex:

```
for {set x 0} {$x < 10} {incr x} {
   set result [expr $result*$x]
}
```

En annan konstruktion för repetition är `foreach` som itererar över varje element i en lista. En variabel tilldelas värdet av det aktuella listelementet som sedan kan användas i body-delen. Så här ser `foreach` ut:

```
foreach varname list body
```

Ex:

```
foreach element $list {
  puts ”Element is: $element.”
}
```

Klammerparentesen som inleder `body`-delarna får inte inleda en ny rad eftersom Tcl använder sig av nyradstecknet för att avsluta kommandon. Klammerparentesen gör så att interpretatorn ignorerar nyradstecknet [2]. Som alternativ till detta kan en backslash sättas sist på raden vilket också ger effekten att nyradstecknet ignoreras.

Nyckelordet `continue` kan användas inom `body`-delen för att avbryta pågående iteration av `body`-delen och går vidare till nästa iteration. `break` kan användas för att avsluta hela `while`-kommandot [21].

3.4.5 Typer

Tcl är mestadels ett typlöst språk. Om det t.ex. är citationstecken runt en sträng siffror förhindrar inte det att strängen används i aritmetiska beräkningar [22]. På källkodsnivå är allt strängar vilka Tcl sedan tolkar som vissa mönster av tecken vilka representerar särskilda typvärden. Dessa värden kan t.ex. vara numeriska, logiska eller listor. Variabler kan vara skalära eller i arrayform. Variabler deklareras inte utan skapas vid tilldelning. För detta används kommandot `set` som skapar en ny variabel och tilldelar den ett värde [22].

I Tcl används parenteser för åtkomst av innehållet i arrayer: `en_array(2)`. Arrayerna är endimensionella, men det går att simulera flerdimensionella arrayer genom att separera indexerna med komma: `en_array(3,2)` [22].

Vad gäller booleska värden innebär `0, false, off` och `no` falska värden medan `true, on, yes och` tal som inte är `0` tolkas som sant [22].

I Tcl är listor en viktig konstruktion. Ett Tcl-skript kan betraktas som en lista med instruktioner. En instruktion kan i sin tur ses som en lista med ord. En lista i Tcl är ett eller flera element som separeras av mellanslag: `apelsin banan kiwi`. Om ett listelement innehåller mellanslag omsluts det med klammerparenteser. Det som egentligen då sker är att elementet som omsluts med klammerparenteser är en lista i sig och alltså en underlista till den yttre listan.
Ex:

```
apelsin banan {gul kiwi}
```

3.4.6 Procedurer

Procedurer skapas i Tcl med kommandot `proc` på detta vis:

```
proc name args body
```

Ex:

```
proc myPuts{text}{
     puts $text
}
```

Proceduren `myPuts` tar här emot ett argument i form av variabeln `text` och skriver sedan ut innehållet i `text` med hjälp av den inbyggda proceduren puts. Det `proc` gör är att lägga till kommandot name i Tcl:s globala namespace [23].

Likt det som beskrevs för Ruby i avsnitt 3.3.5 kan det även i Tcl finnas valfria argument som har defaultvärden.

3.5 Perl

Perl är för tillfället ett av de mest kända och populära språken att programmera i [1]. Det är språket som för de flesta är förstahandsvalet vid val av språk för CGI-script. Språket har väl

inbyggt stöd för stränghantering vilket gör att det passar bra för CGI-skript. I språket finns även starkt stöd för fil- och nätverkshantering. Perl är inte tänkt att användas som ett ”lim-språk”, som många andra skriptspråk är, men det kan ändå användas för det vid behov [2]. Perl kan kategoriseras enligt både de procedurella och objektorienterade paradigmen [24].

3.5.1 Syfte och filosofi

Perl har som filosofi att allting ska gå att göra på mer än ett sätt. Detta har gjort att Perl har fått som motto: “There Is More Than One Way to Do It”. Detta motto har medfört att olika skript som utför samma sak kan se ut på väldigt olika sätt [25]. Denna filosofi skiljer sig stort mot Rubys filosofi där de strävar efter enkel och lättläst kod.

3.5.2 Utveckling

Perl skapades 1986 och var då tänkt användas som ett nätverksinstrument. Perl står för Practical Extraction and Reporting Language och innebörden av namnet stämmer bra med de områden där Perl utmärker sig. Språket utvecklades så att det skulle vara bra på att hantera filer, läsa information från filer och på att strukturera information så att den kunde visas upp för användaren [26].

Perls skapades av Larry Wall för att användas i Unix miljö, men sedan skapandet så har språket portats till många andra operativsystem. Språket, dess kod, bibliotek och dess dokumentation är fritt att använda för den som vill [25].

Mycket av sin syntax och konstruktioner har Perl ärvt från andra språk. Syntaxen liknar till stor del den i C. Perl har även ärvt funktionalitet från Unix-kommandotolkar samt språken sed och awk [25].

Perl har fått en stor spridning till stor del tack vare dess användning i CGI-skript. Det tillsammans med språkets bredd har gjort språket till ett av de mest använda skriptspråken. Språket är känt för att kunna hantera problem inom väldigt många olika områden och denna förmåga har lett till att språket kallas ”the Swiss Army chain saw of languages” [2].

3.5.3 Speciella konstruktioner

Fördefinierade variabler

Perl innehåller en stor mängd fördefinierade variabler som används automatiskt i Perls inbyggda funktioner om inte andra variabler anges som parametrar. Detta implicita användande av variabler gör att koden kan bli svårläst för nybörjare till Perl. Namnen på de fördefinierade variablerna är oftast inte längre än tre tecken långa och består ofta till stor del av icke-alfabetiska tecken. De fördefinierade variablerna har också en engelsk version på sina namn som kan användas genom att `use English;` läggs till i skriptet [25]. De korta namnen på variablerna, och deras ofta implicit användande, medför att Perlskript kan bli korta och kompakta men med nackdelen att de kan kräva stor kännedom om Perl och dess inbyggda variabler för att kunna förstås. Många av de fördefinierade variablerna innehåller information om miljön där Perlskriptet körs samt information om inställningar i interpretatorn [2].

De mest kända och använda fördefinierade variablerna är `$_` och `@_` (för en förklaring av Perls typer se avsnitt 3.5.6.). `$_` är default variabeln som automatiskt används i Perl-funktioner då en skalär variabel behövs [25].

```
@mat = (”korv”, ”potatis”, ”apelsin”);
foreach (@mat) {
    print;
}
```

Koden ovan skriver ut alla element i arrayen `@mat`. Elementen i `@mat` mellanlagras aldrig i någon synlig variabel utan Perl lagrar varje element som hämtas från `@mat` i `$_`. Funktionen `print` använder `$_` som parameter om den inte explicit får en variabel som parameter. Kontrollstrukturen `foreach` hämtar ut elementen från den inskickade arrayen och lagrar dem i `$_` om ingen variabel används explicit för att lagra elementet [25]. Koden nedan visar hur ovanstående kod tolkas av Perl interpretatorn:

```
@mat = (”korv”, ”potatis”, ”apelsin”);
foreach $_ (@mat) {
    print $_;
}
```

Den andra mycket använda fördefinierade variabeln @_ är en array som används för att hålla parametrarna som skickas in till en funktion. Parametrarna kan sedan hämtas från @_.

```
sub printNumTimes{
```

```
    ($text, $num) = @_;
    for($index = 0; $index < $num; $index++){
        print $text;
          }
}
printNumTimes(”Hej”, 5);
```

Koden ovan visar hur två parametrar, som skickats till funktionen `printNumTimes`, hämtas från `@_` och lagras i variablerna `$text` och `$num` [25].

Texthantering

Perl är bra på att hantera text tack vare inbyggda funktioner i språket samt ett starkt stöd för reguljära uttryck. Vid användandet av reguljära uttryck så jämförs en textsträng mot ett mönster för att se om strängen matchar mönstret. Perl använder operatorn `m//` för att se om en sträng matchar ett mönster och returnerar `true` eller `false` beroende på om strängen matchade eller inte. För att `m//`-operatorn inte ska använda sig utav `$_` som teststrängs argument så används `=~` operatorn. Operatorn `=~` talar om för `m//` att den istället för `$_` istället ska använda sig utav variabeln som finns till vänster om `=~`. Perl har också en operator, `!~`, som till funktionalitet fungerar som en negation av `=~` operatorn. Skulle `=~` operatorn byttas ut mot `!~` operatorn vid en jämförelse av en sträng och ett mönster så skulle returvärdet negeras [26].

För att byta en del av en sträng mot en annan sträng så används `s///` operatorn. `s///` operatorn använder sig av samma sätt att matcha strängar som `m//` men istället för att returnera `true` eller `false` vid överensstämmelse så byter `s///` operatorn ut den matchande textsträngen mot en annan textsträng som operatorn har. Exemplet nedan visar hur ordet ”`young`” byts ut mot ordet ”`old`”.

```
$text = "Pretty young.";
$text =~ s/young/old/;
```

En utskrift av variabeln `$text` skulle efter dessa rader bli: ”`Pretty old.`” [26].

Utöver `m//` och `s///` så innehåller Perl ännu en funktion för stränghantering, operatorn `tr///`. Denna operator används för att byta ut ett tecken mot ett annat. I nedanstående exempel ersätts ordet kotte i `$text` med ordet kaffe:

```
$text = "kotte";
$old_char = "ot";
$new_char = "af";
eval "\$text  =~ tr/$old_char/$new_char/";
```

Funktionen eval ovan utvärderar strängen som tas in som parameter på samma sätt som eval i Ruby, se kap 3.3.4.

3.5.4 Flödeskontroll

Selektion

I Perl så används konstruktioner som är väldigt lika de i programspråket C för att styra programflödet. Selektion kan ske med `if`-satser där skillnaderna mot varianten i C är att istället för `else if` skrivs i Perl `elsif`. Blocken som följer en `if`-sats måste ha `{}` runt även om de enbart innehåller en rad. Perl tillåter att uttryck skrivs följt av en `if`-sats, detta innebär att uttrycket utförs enbart om `if`-satset utvärderas till sant fastän uttrycket står före `if`-satsen. Perl innehåller också en konstruktion, `unless`, som till funktion liknar en negerad `if`-sats. Det finns ingen motsvarighet till `elsif` för `unless`-konstruktionen. I Perl finns även konstruktionen `? :` som fungerar på samma sätt som i C, denna tertiära konstruktion är i Perl en operator vilket gör att den kan användas direkt i uttryck [25].

Repetition

Perl använder sig utav konstruktionerna `while`, `until`, `for` och `foreach` för att utföra viss kod flera gånger. Perls `while` liknar mycket den `while` som finns i C, Perl tillåter dock som med `if`-satsen att ett uttryck föregår `while`-satsen. Uttrycket kommer att utföras så länge `while`-satsen utvärderas till sant. `until` fungerar som en negerad `while`-sats, så länge `until`-satsen är falsk så utförs det tillhörande blocket och när det är sant så avbryts repetitionen. `for`-satsen liknar även den sin motsvarighet i C. `foreach` är en konstruktion som tillåter iteration över en lista. Elementen i listan hämtas ut och tilldelas till en variabel som kan användas i det efterföljande blocket. Variabeln kan utlämnas vilket, som i många andra liknande situationer i Perl, medför att det utlämnade elementet kommer att

tilldelas till den inbyggda variabeln `$_` [25]. `for` och `foreach` är samma loop i Perl, vilket innebär att funktionaliteten som här har beskrivits för `for` även kan användas för `foreach` och tvärtom [26]. `while-`, `until-` och `foreach`-satserna kan ha ett `continue`-block efter sig vilket utförs efter varje repetition av loopen.

Perl innehåller tre operatorer som används för att kontrollera flödet i en loop. Dessa är `next`, `last` och `redo`. `next` hoppar över resten av blocket, påbörjar loopen igen och utvärderar villkorssatsen för loopen. Om loopen efterföljs av ett `continue`-block så utförs det innan loopen påbörjas igen. `last` fungerar precis som `break` i C, den avslutar loopen utan att utföra de efterföljande satserna. Inte heller ett efterföljande `continue`-block utförs om last anropats. `redo` hoppar över resten av blocket och även ett efterföljande `continue`-block och börjar om loopen igen men utan att utvärdera vilkorssatsen igen [25].

I Perl så finns det etiketter, så kallade labels, som användas för att ge ett block ett namn. En label kan underlätta användandet av de ovannämnda operatorerna för loopkontroll. Om t.ex. `next` används inne i två eller fler nästlade loopar så kan `next` påbörja den yttre av looparna genom att `next` följs av labeln för den yttre loopen [25].

3.5.5 Procedurer

Procedurer i Perl kallas subrutiner eller funktioner. Som i så mycket annat så har Perl många olika möjligheter för vad som ska anges vid skapandet av en procedur. Nyckelordet `sub` används för att tala om för Perl att den efterföljande deklarationen eller definitionen är en subrutin. Perl tillåter att en subrutin skapas utan att den tilldelas ett namn. Denna icke-namngivna subrutin måste ha ett block där definitionen för den finns och subrutinen måste även tilldelas till en variabel för att det ska vara möjligt att få tillgång till subrutinen vid ett senare tillfälle [25].

Argument till en subrutin tilldelas till den inbyggda arrayen `@_`, från vilken argumenten sen kan hämtas inne i subrutinen. Argument som är arrayer får alla sina element hämtade och även de läggas i `@_`. Prototypen, som talar om vad för argument en subrutin kan ta emot, kan utlämnas i Perl. Utlämnas prototypen innebär det att subrutinen kan ta emot obegränsat antal argument. Prototypen talar om hur många argument en subrutin tar och vad för typ argumenten har, om de är skalära variabler, arrayer eller har någon annan typ [26]. Argument

som har typerna `@` och `%` lägger till alla efterföljande argument till sig själva, detta innebär att om prototypen anger att subrutinen tar emot en skalär variabel och en array och fyra skalära variabler skickas som argument så kommer den första skalära variabeln tas emot som skalär variabel medan de övriga tre läggs till arrayen. För att tala om för Perl att argumentet som skickas in måste vara av en viss typ, och inte bara ombildas till den typen som i exemplet, så ska en ”\” läggas till framför typen på argumentet [25].

3.5.6 Typer

I Perl så börjar variabler med ett tecken som talar om vad för sorts variabel det är. En enkel skalär variabel betecknas med `$`, en array med `@` och en hashtabell med `%` [2]. Värdet som en skalär kan innehålla kan vara antingen numeriskt, en sträng eller en pekare till en annan variabel. Arrayer och hashtabeller innehåller båda skalära värden skillnaden mellan dem är hur de ger åtkomst till dem. Arrayer använder indexoperatorer [] för att komma åt ett visst element medan hashtabeller använder nyckeln i sitt nyckel/värde-par tillsammans med {} för att komma åt ett specifikt element. När ett visst element ska kommas åt i en array eller en hashtabell så används `$`, istället för `@` eller `%`, eftersom det är en skalär som hämtas [26].

Typeglobs är en datatyp i Perl som använder `*` som första tecken. En typglob är del av symboltabellen som Perl använder och i den delen lagras datatyperna som har associerats med ett visst namn t.ex. `$ballong` och `%ballong`. När en typeglob utvärderas så representerar den varje variabel som har samma namn. `*bar` kan representera `$bar`, `@bar`, `%bar` och `&bar` [2]. Koden nedan visar hur en typeglobe tilldelas till en annan:

```
$ballong = ”heliumballong”;
@ballong = ("gul", "bla", "rod");
*bar = *ballong;      #bar blir ett alias för *ballong.
print (”$bar\n”);
```

`*bar` kan nu användas för att komma åt värdena i både `$ballong` och `@ballong`. Typeglobenen måste då använda samma första tecken som den variabel den vill få fram värdet för. Den sista kodraden ovan, `print(”$bar\n”);`, ger utskriften `”heliumballong”` på skärmen vid utvärdering [26].

Perl innehåller från version 5 möjligheter till objektorienterad programmering. Perls klasser är packages. En klass kan innehålla metoder, dessa är i Perl bara subrutiner som finns i ett visst paket. För att utifrån en klass skapa ett objekt så anropas i Perl en konstruktor. Konstruktorn är en metod, oftast kallad `new()`, i klassen som returnerar en referens till ett nytt objekt. Konstruktorn måste inte i Perl döpas till `new()` utan den skulle kunna döpas till vilket giltigt namn som helst [26]. I Perl så används den inbyggda funktionen `bless` för att binda en skalär till en viss klass. När en variabel blivit välsignad (blessed) in i en klass så innebär det att variabeln, eller objektet som det nu har blivit, vet var den ska hämta sina metoder. Vanligtvis används en hashtabell för att lagra data i en klass men det skulle kunna vara t.ex. en skalär eller en array. En klass i Perl kan också innehålla en destruktor för att göra rent efter ett objekt innan det förstörs. Objektet förstörs oavsett om destruktorn existerar. En destruktor i Perl måste döpas till `DESTROY`. Ett objekts destruktor anropas automatiskt, om det existerar, av Perl innan objektet ska förstöras [27]. Nedan följer ett exempel på hur en klass skulle kunna se ut:

```
package TestKlass;

sub new{      #konstruktor
    my $self = {};    #skapar en referens till en anonym hash
    bless($self);     #välsignar referensen till klassen
    return $self;     #returnerar referensen
}
return 1;
```

För att skapa ett objekt av klassen `TestKlass` så anropas `new()` såhär:

```
use TestKlass;
my $testObject = TestKlass->new();
```

Instansvariabler kan läggas till ett objekt genom att de läggs till den anonyma hashtabellen som returneras av `new()`. Om konstruktorn anropas med TestKlass->new(testString => ”test”); så kan instansvariabeln teststring, med värdet `test`, läggas till objektet som returneras av konstruktorn. Konstruktorn måste dock ha funktionalitet för att ta hand om och lagra parametrarna för att detta ska ske [27]. Detta innebär också att objekten av en viss klass kan skilja sig åt om olika datamedlemmar sätts i objekten via konstruktorn.

Inkapslingen av datamedlemmar i Perl går till så att `set-` och `get-`metoder läggs till klassen så att åtkomst till datamedlemmarna ska gå via dessa metoder. Detta hindrar dock inte en användare från att direkt hämta datamedlemmen via objektet:

```
my $object = Person->new();
print $object->{NAME};          #här skrivs namnet på personen ut. Namnet sätts i
                                #konstruktorn.
```

Ett korrekt anrop skulle kunna se ut såhär:

```
my $object = Person->new();
print $object->get_name();
```

4 Testresultat

I detta kapitel redovisas de resultat som framkommit av det test som genomförts. Resultaten som redovisas berör inläsning från fil, sortering, uppstart och avslut, antal rader kod och implementationstid. En beskrivning görs även om testets genomförande, de verktyg som använts samt quicksort-algoritmen.

4.1 Genomförande

För att testa de olika skriptspråkens effektivitet och expressivitet har vi valt att i varje språk skriva ett quicksort-skript där ett filnamn tas som argument. Filen ska innehålla ett antal tal, åtskilda av radbrytning. Skriptet läser in talen och sorterar dem med quicksort-algoritmen, se avsnitt 4.2. Skriptet skriver ut den tid i millisekunder det tog för att först läsa in filen och sedan sortera talen. Vi skrev även ett speciellt testskript i PHP som kör varje språks quicksort-skript totalt 99 gånger med olika filer som argument. Vi var oroliga för att PHP.s resultat skulle påverkas av att det yttre testskriptet även det kördes i PHP. Vi skrev då om det yttre testskriptet i Perl och såg då att PHP:s resultat inte påverkades av vilket språk det yttre testskriptet var skrivet i. Testskriptet tar in det språk som ska testas som argument och skriver ut totaltiden för varje testiteration. Testfilerna innehåller n antal element med värden från 0 till n-1. Varje värde finns endast representerat en gång. Ordningen på talen är slumpade med hjälp av ett Ruby-skript. Det finns 11 olika storlekar på testfilerna och för varje storlek finns det nio olika filer. Med storlek menar vi antal element. Storlekarna är 1 000, 10 000, 20 000, 30 000, 40 000, 50 000, 60 000, 70 000, 80 000, 90 000 och 100 000. Testerna har utförts i både Windows Xp- och Linux Ubuntu-miljö. Resultatet av testerna har samlats i Excel-dokument och sammanställts med medelvärde i tabell- och diagramform.

Det yttre testskriptet mätte den totala tiden för varje sortering, medan de inre språkspecifika skripten själva mätte tiden för inläsning och sortering. Tiden för uppstart och avslut för varje körning räknades ut genom att subtrahera inläsningstiden och sorteringstiden från den totala tiden.

På grund av otillfredsställande försök vid testerna för PostScript så valde vi att, för PostScript, göra liknande tester men med speciella storlekar på testfilerna samt en manuell körning av

testerna. Anledningen till att vi valde att göra på detta vis var att Ghostscript låste sig om antalet element i filen som skulle sorteras var större 28 000. Vi valde därför att skapa ett antal testfiler innehållandes färre element för att PostScript fortfarande skulle få tillräckligt med värden för att kunna användas för en jämförelse. På grund av de stora skillnaderna i resultat samt upplägg av testerna mellan PostScript och de övriga språken vi testat så har vi valt att, för testerna, jämföra de övriga fyra språken som en grupp mot PostScript. Tiderna som rapporterades tillbaka av PostScript är något osäkra för de testfiler med ett lågt antal element. I Linux växlar tiderna mellan noll och ett tiotal millisekunder. I Windows så växlas det inte mellan noll och ett tiotal, men det sker fortfarande en växling mellan noll och ett högre, ofta återkommande, tal. Vi vet ej vad detta beror på men C visade upp samma tendenser som PostScript. Ruby verkade inte kunna mäta tider under 15 millisekunder i Windows då tiden för inläsning av 1 000 element antingen blev 0 eller minst 15.

Vi har även gjort en implementation i C av quicksort-algoritmen för att kunna jämföra mot skriptspråken.

De interpretatorer och kompilatorer vi använt är:

Windows XP

- PHP: PHP 5.1.1
- Tcl: tclsh 8.4.12
- Perl: perl 5.8.7
- Ruby: Ruby 1.8.2
- PostScript: GPL Ghostscript 8.15
- C: Dev-C++ 4.9.9.2

Linux Ubuntu

- PHP: PHP 5.0.5
- Tcl: tclsh 8.4.9
- Perl: perl 5.8.7
- Ruby: ruby 1.8.3
- PostScript: ESP Ghostscript 7.07
- C: gcc 4.0.2

Vid testerna har inga optimeringsflaggor använts vid kompilering eller körning.

Den hårdvara som använts vid testerna som är värd att nämna är:

- Acer Aspire 5021WLMi
- AMD® Turion™ 64 ML-28 processor
- 512MB DDR RAM

Resultaten av testerna finns i bilaga B, Testresultat. I de följande kapitlen sammanfattas resultaten.

4.2 Quicksortalgoritmen

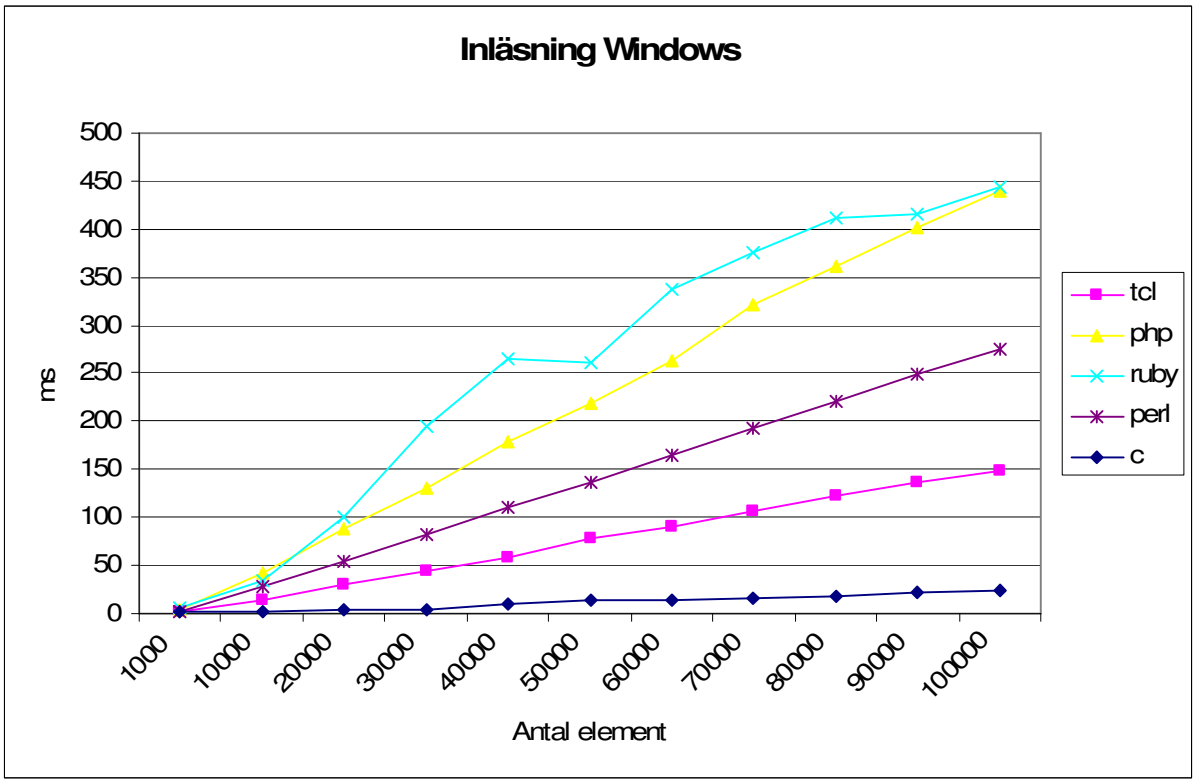
Quicksort är en sorteringsalgoritm som i bästa fall gör O(n log n) jämförelser för att sortera n antal element. Vid sämsta tänkbara scenario görs det $O(n^2)$ jämförelser [28].

Algoritmen för att sortera en array S följer följande fyra steg:

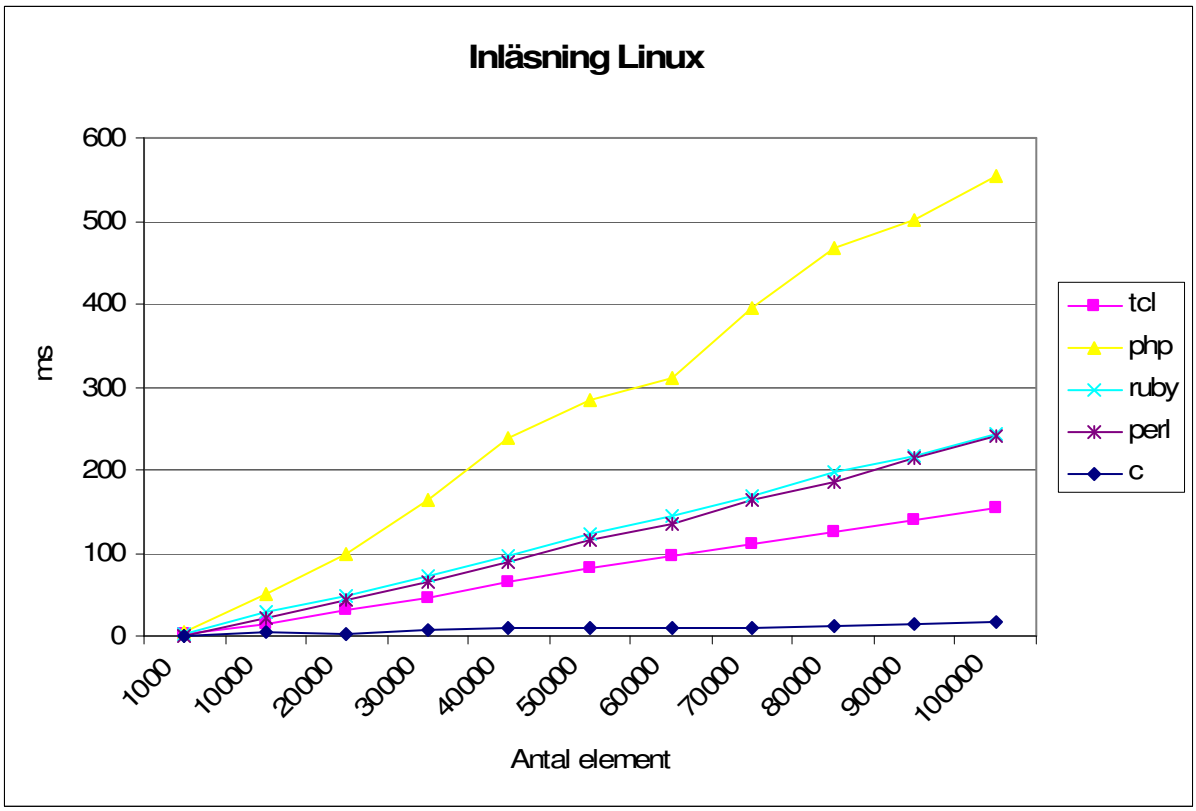
1. Om antalet element i S är 0 eller 1, så returnera.
2. Välj något element v i S som pivotelement.
3. Dela upp elementen i S förutom v i två delar där ena delen, S_1, innehåller element som är mindre än v och den andra delen, S_2, innehåller element som är större eller lika med v.
4. Returnera quicksort(S_1) följt av v, följt av quicksort(S_2).

4.3 Inläsning

Vid inläsning av testfilerna är Tcl snabbast i både Windows och Linux, och det går lika fort på båda plattformarna. Inte heller i PHP eller Perl är det någon större skillnad i hastigheterna på de olika plattformarna. Det enda språket som visar någon skillnad värd att nämna är Ruby, som i Linux nästan halverar tiden för inläsning (444 ms för 100 000 element i Windows mot 243 ms i Linux). Se Figur 1, Inläsning Windows och Figur 2, Inläsning Linux.



Figur 1, Inläsning Windows

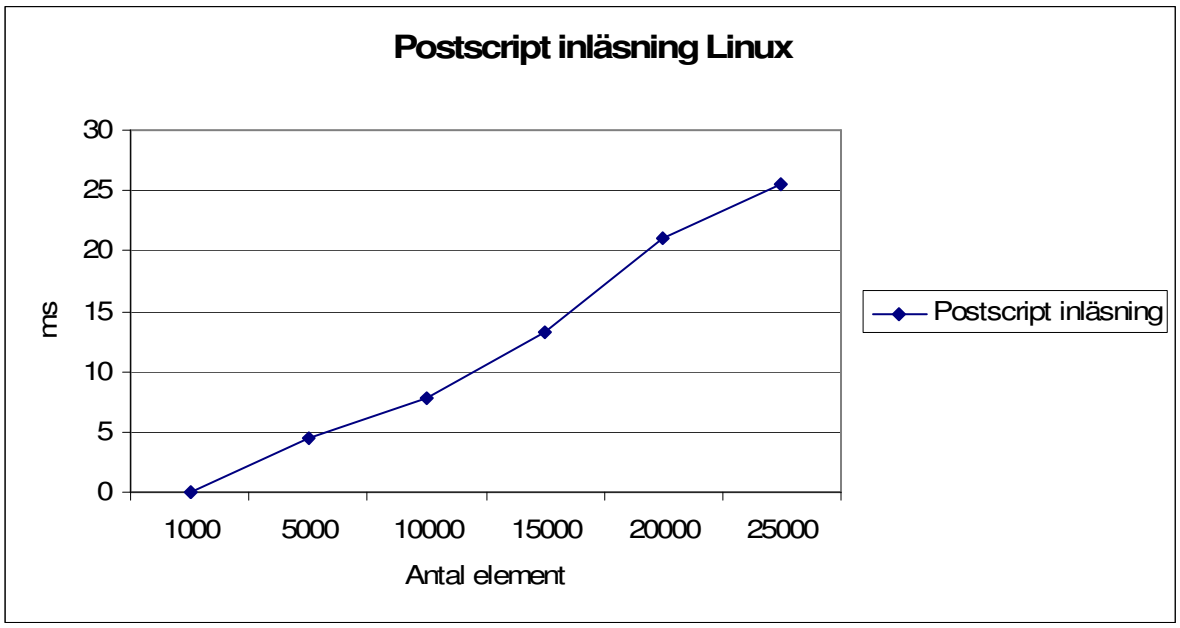


Figur 2, Inläsning Linux

Värt att notera är även att Rubys kurva för inläsning i Windows är väldigt oregelbunden, medan den i Linux är linjär. De andra språken är linjära både i Windows och i Linux. I Linux är PHP märkbart långsammare än de övriga de övriga språken. PHP tar mer än dubbelt så lång tid för inläsning jämfört med Ruby och Perl, och mer än 3 gånger så lång tid som Tcl. I Windows är splittringen bland språken större, Ruby och PHP är långsammast, följt av Perl och ett klart snabbare Tcl.

C är när det gäller inläsningen klart snabbare än skriptspråken både i Linux och i Windows. I Linux är totaltiderna för C ungefär 1/9 av totaltiderna för Tcl, som har den snabbaste totaltiden bland skriptspråken. I Windows är skillnaderna i totaltid mindre mellan C och skriptspråken.

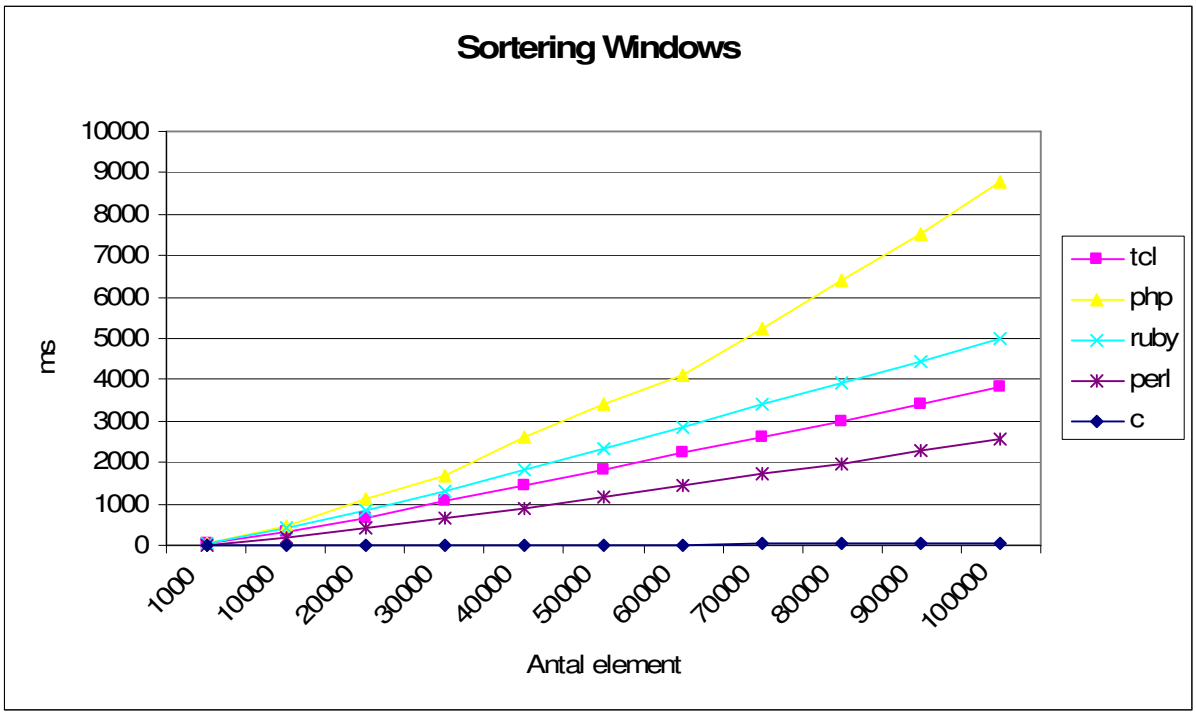
PostScript utmärker sig när det gäller inläsning då språket är snabbare eller lika snabbt som de övriga skriptspråken i inläsningstider upp till den största storleken som PostScript testades på. Detta gäller för båda plattformarna. Dock så måste de problem som nämns om PostScripts tidtagning i 4.1 tas i beaktande. Se Figur 3, Postscript inläsning Linux och Figur 13, Postscript inläsning Windows



Figur 3, Postscript inläsning Linux

4.4 Sortering

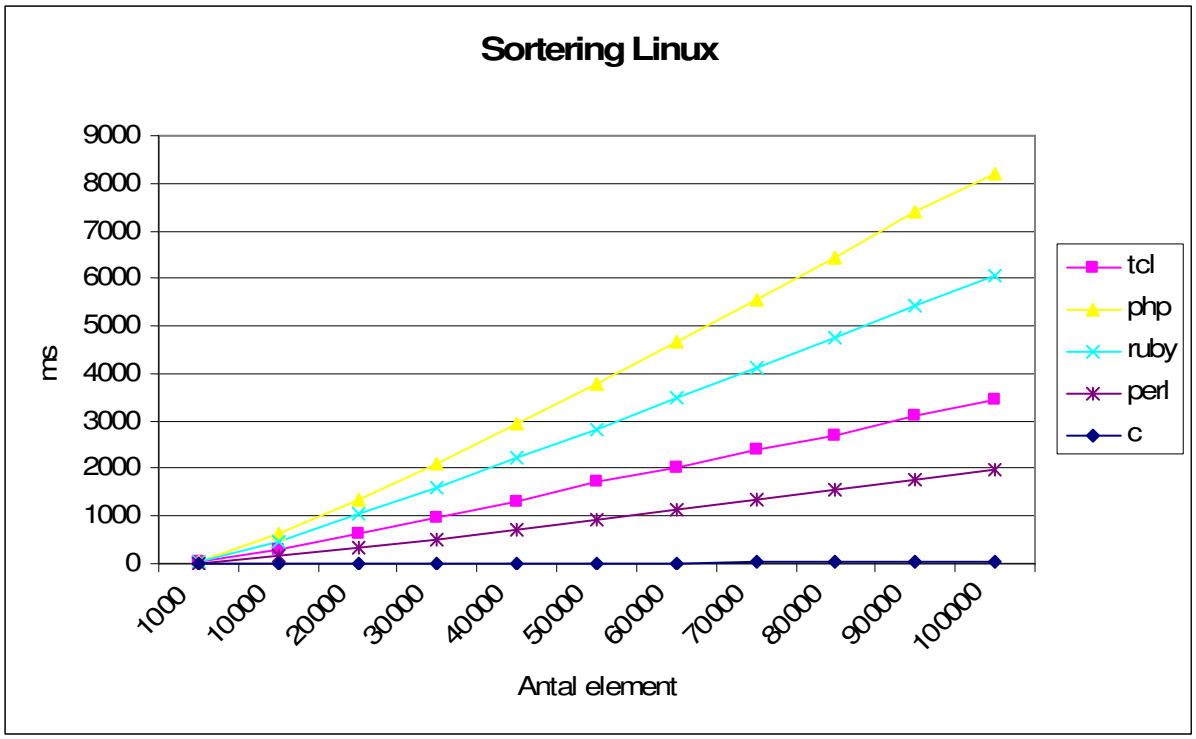
Vid sorteringen, liksom vid inläsningen, är PHP långsammast bland skriptspråken. Språket är något snabbare i Linux än i Windows. PHP:s kurva är i Windows lägre vid ett lågt antal element än för samma antal element i Linux, medan vid högre antal element så är kurvan i Windows högre än den för Linux. Kurvan i Windows är alltså mer exponentiell än kurvan för Linux. Se Figur 4, Sortering Windows och Figur 5, Sortering Linux.



Figur 4, Sortering Windows

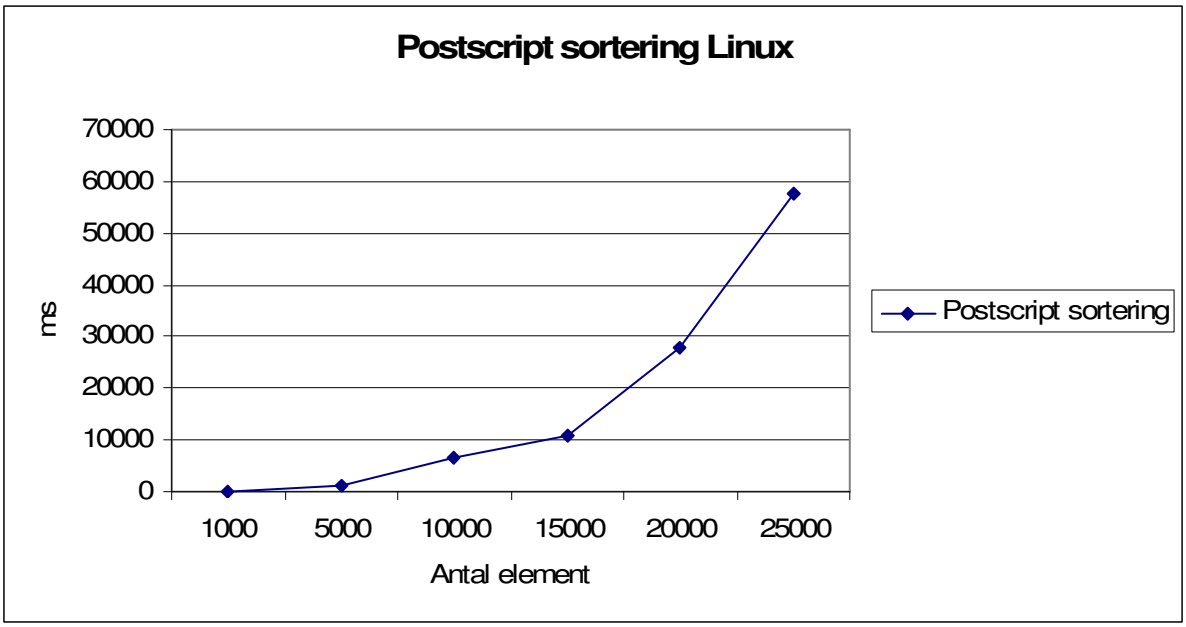
Ruby är näst långsammast, med en snabbare tid i Windows än i Linux. Tcl är snabbare än Ruby och snabbast är Perl. Kurvorna för dessa tre språk är mer eller mindre linjära. Skillnaderna för Tcl och Perl är i stort sett obetydliga mellan de olika plattformarna.

C är även för sorteringen klart snabbare än de övriga språken och förhållandet i tid är för sorteringen ännu större än det är vid inläsningen.



Figur 5, Sortering Linux

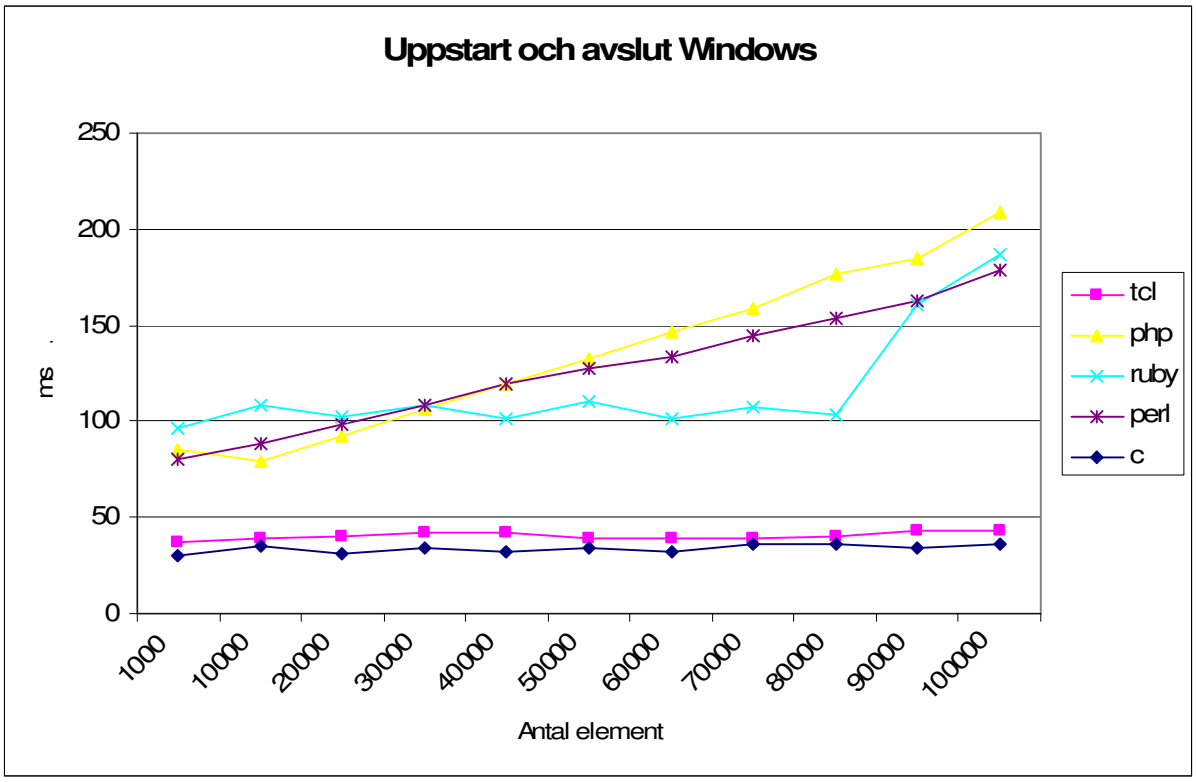
PostScript tar ungefär dubbelt så lång tid på sig vid sorteringen av 1 000 element som gruppen med skriptspråken. Vid större antal element växer skillnaderna mellan PostScript och de övriga språken dramatiskt. T.ex. tar 20 000 element c:a 1 sekund i PHP, medan det i PostScript tar hela 36 sekunder. Se Figur 6, Postscript sortering Linux och Figur 14, Postscript sortering Windows. Värt att nämna här är kanske att PHP är det långsammaste av de övriga skriptspråken.



Figur 6, Postscript sortering Linux

4.5 Uppstart och avslut

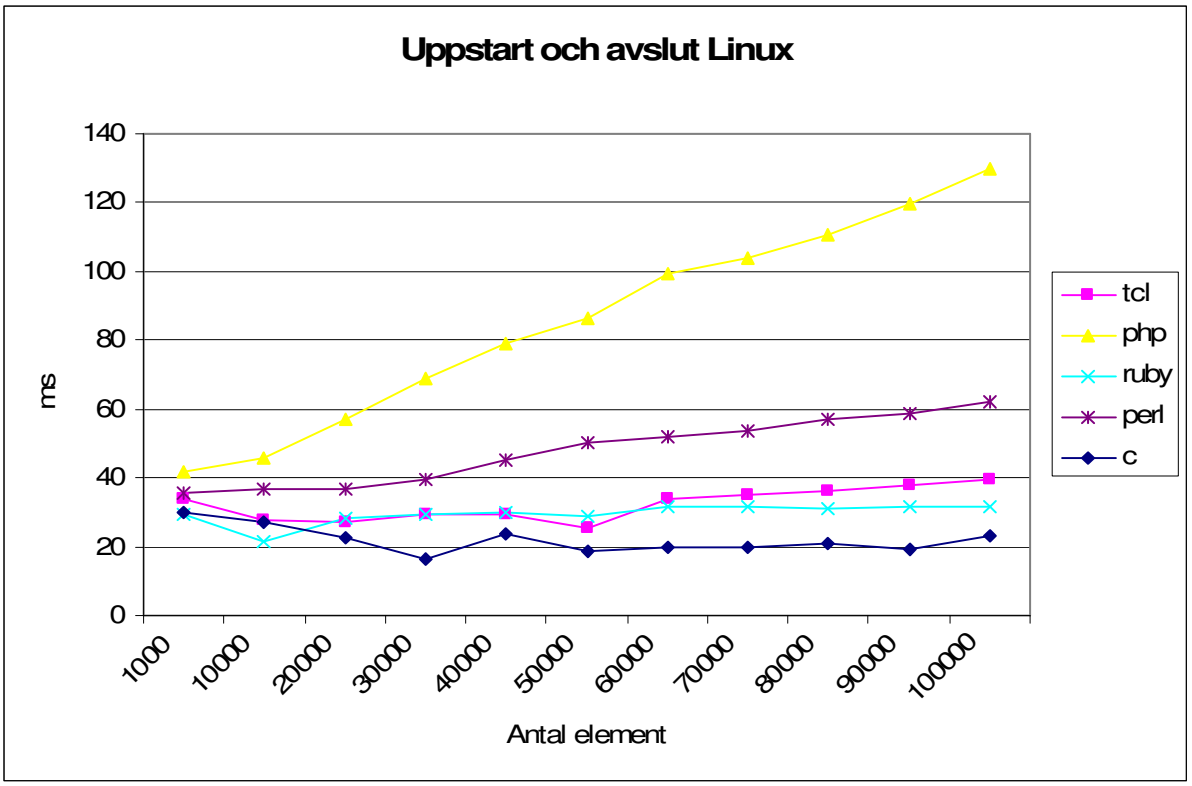
Uppstart och avslut är den tid skripten och programmen använt för varje körning som inte innefattar inläsning och sortering. Denna tid har räknats fram genom att subtrahera tiden för inläsning och sortering från den totala tiden. Det finns i huvudsak två typer av kurvor när det gäller resultaten för uppstart och avslut. Den ena typen är linjärt stigande, medan den andra är (i stort sett) konstant. PHP och Perl har båda två en linjärt stigande kurva. I Linux är det stor skillnad på tiderna mellan dem, där Perl är den snabbare. I Windows följs de båda kurvorna åt från c:a 80 millisekunder vid 1 000 element till c:a 200 millisekunder vid 100 000 element. I Linux utgår de bådas kurvor från c:a 40 millisekunder vid 1 000 element. Vid 100 000 element är dock Perl snabbare och ligger på drygt 60 millisekunder mot PHP:s 130. Se Figur 7, Uppstart och avslut Windows och Figur 8, Uppstart och avslut Linux.



Figur 7, Uppstart och avslut Windows

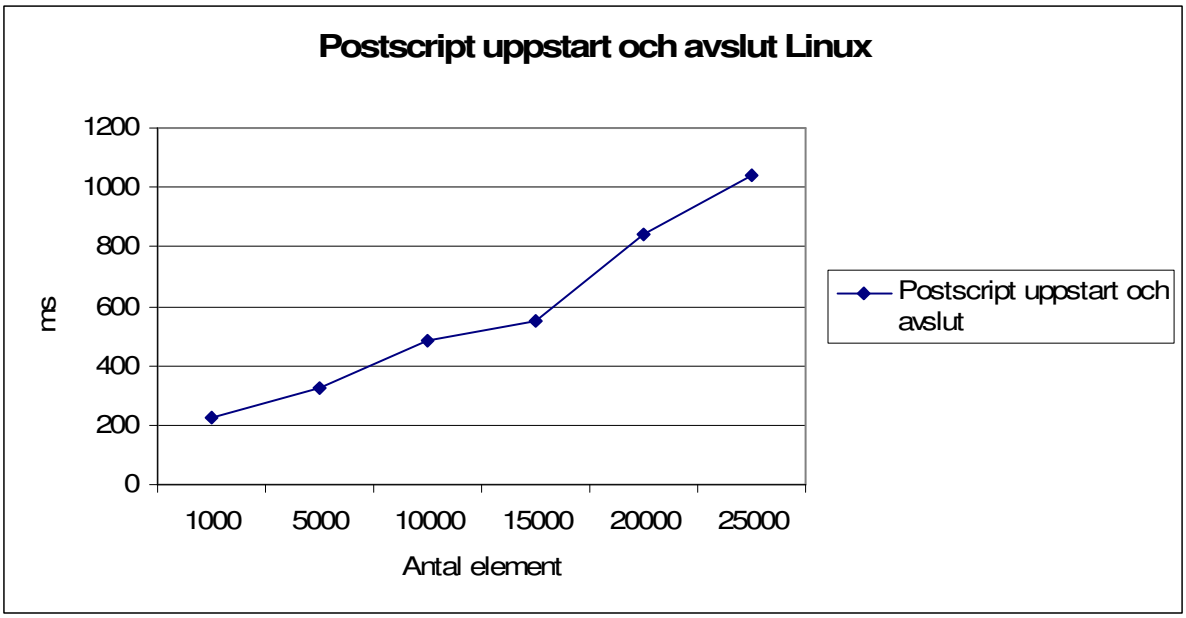
Ruby och Tcl är de språk som har en konstant kurva, med tider runt 30 millisekunder i Linux för alla teststorlekar. I Windows håller sig Tcl:s kurva runt 40 millisekunder medan Ruby här går lite långsammare och ligger runt 100 millisekunder. På de två sista mätvärdena, 90 000 och 100 000 ökar Rubys tider till 161 och 187 millisekunder, vilket är en rätt kraftig ökning.

C är även i uppstart och avslut snabbast med tider runt 20 millisekunder, men får här hård konkurrens. I Linux ligger både Tcl och Ruby väldigt nära C tidsmässigt. I Windows är det bara Tcl som är i närheten.



Figur 8, Uppstart och avslut Linux

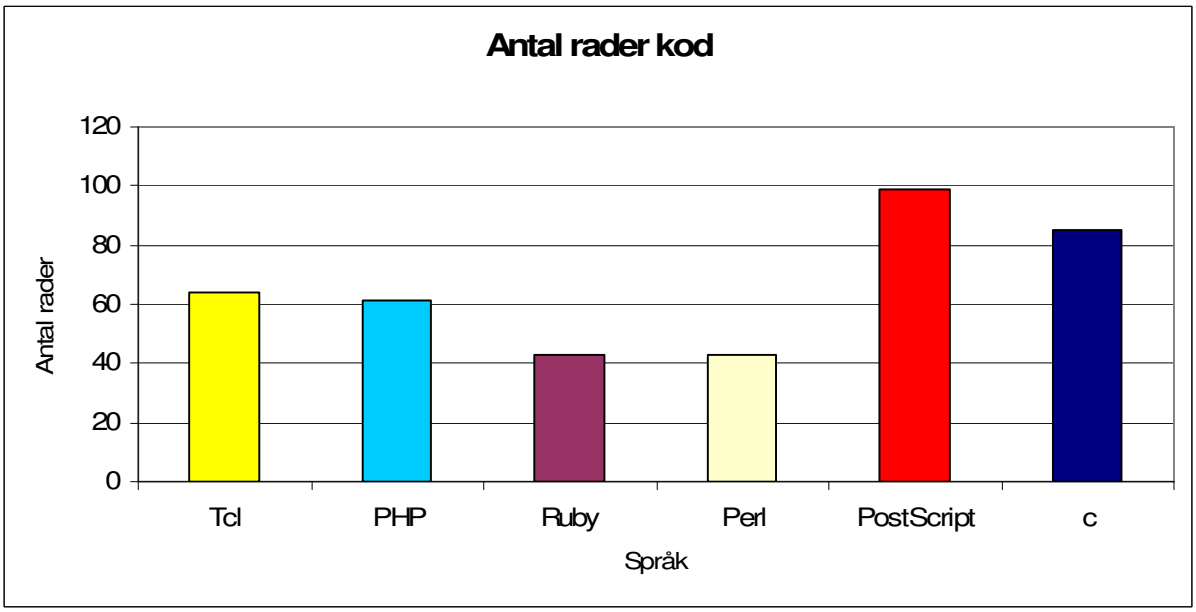
PostScript klarar sig relativt bra i uppstart och avslut i Windows med en hyfsat konstant kurva runt 175 millisekunder. Dock så tar uppstart och avslut för PostScript i Windows dubbelt så lång tid som de närmaste skriptspråken Ruby, Perl och PHP. I Linux är kurvan linjärt stigande från drygt 200 millisekunder vid 1 000 element till drygt 1 000 millisekunder vid 25 000 element. Se Figur 9, Postscript uppstart och avslut Linux och Figur 16, Postscript uppstart och avslut Windows.



Figur 9, Postscript uppstart och avslut Linux

4.6 Antal rader kod och implementationstid

Ruby och Perl är de skriptspråk som krävde minst antal rader kod, med 43 rader vardera. Tcl och PHP krävde dryga 60 rader, C 85 rader, och PostScript 99 rader, se Figur 10, Antal rader kod. Det är samma antal rader kod i Linux som i Windows.



Figur 10, Antal rader kod

Tiden det tog att implementera quicksort i de olika språken är inte exakt mätt, utan enbart uppskattad. Rangordning efter tid med kortast tid först följer här:

1. Ruby
2. PHP
3. Perl
4. Tcl
5. C
6. PostScript

5 Analys

I detta kapitel analyseras och diskuteras skriptspråken och deras testresultat från kapitel 4. Språken jämförs och diskussion förs om vad för anledningar det kan finnas som förklaring till testresultaten. En egen definition för skriptspråk görs utifrån deras egenskaper som tidigare diskuterats i uppsatsen. Skriptspråken klassificeras också utifrån deras utveckling och historia.

5.1 Tester

Vid inläsning kan vi konstatera att Tcl är snabbast, följt av Perl, i både Windows och Linux. PHP tar däremot relativt lång tid på sig vid inläsning. Skillnaden i tid mellan PHP och de övriga språken tror vi kan bero på att PHP har som syfte att producera text snarare än att läsa in text från fil. PHP är väl anpassat för att hämta data från databaser, vilket gör att PHP själv slipper ha hand om själva filinläsningen. Det kan vara så att den här funktionaliteten har prioriterats högre än att effektivt kunna läsa från fil. Anledningen att Perl är effektivt vid filinläsning tror vi har att göra med deras syfte som språk, att kunna hantera text med hjälp av reguljära uttryck vilket gör att filinläsning får en hög prioritet så att den kraftfulla texthanteringen kan komma till sin rätt. Att Tcl är så pass snabbt är för oss något förvånande då dess ursprungliga användningsområde från början var att styra andra applikationer snarare än att behandla data. Vad gäller PostScript är det en positiv överraskning gällande filinläsning. Upp till de mätvärden som är möjliga, håller PostScript jämna steg med Tcl. Med tanke på de övriga resultat PostScript uppnått, får detta ses som uppseendeväckande. Det är svårt att se någon anledning eller förklaring till varför det är så pass effektiv inläsning i PostScript. Ruby står sig väl i konkurrensen vid inläsning i Linux och är lika snabb som Perl. I Windows däremot är Ruby långsammast av alla de jämförda språken och har en väldigt ojämn kurva. En möjlig teori varför det är på detta sätt är att Ruby ursprungligen utvecklades för UNIX-miljöer och att portningen till Windows blev lika effektiv.

Jämfört med C är skriptspråken långsamma vid sortering. Detta tror vi beror på att databehandling av detta slag inte är vad skriptspråken utvecklats för. Av skriptspråken är ändå Perl det språk som visar på högst effektivitet. Av den känsla vi fått vid implementeringen av testskripten, tycker vi att Perl och Tcl håller sig på en lägre språknivå än de övriga. Möjligtvis

är detta en förklaring till varför Perl och Tcl var snabbast vid sorteringen. PHP gör oss återigen besvikna, och tar t.ex. fyra gånger så lång tid som Perl vid sortering av 100 000 element. Tunga beräkningar har aldrig varit PHP:s syfte vilket kan förklara de undermåliga resultaten. PostScript visar här klart och tydligt att denna typ arbete inte är någon som språket är väl anpassat för, vilket antagligen beror på dess användning av stackar vid datalagring.

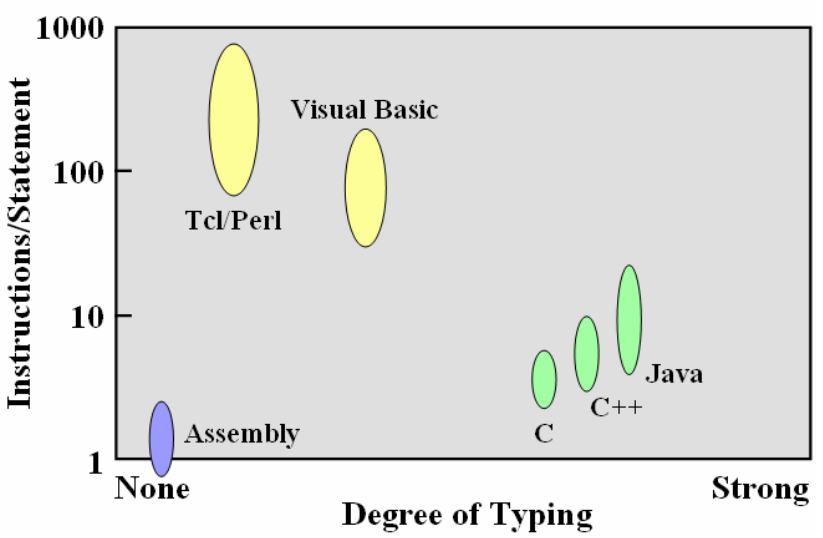
Angående uppstart och avslut klarar sig de flesta av språken bra jämfört med C i Linux. PHP är återigen sämst, om vi bortser från PostScript. En förklaring till detta kan vara att PHP oftast används som en integrerad modul i en webbserver. Det krävs då ingen uppstart av en egen process för PHP. I Windows är det endast Tcl som håller jämna steg med C. Perl går klart sämre i Windows och följer PHP:s linjärt stigande kurva. Ruby, Perl och PHP har redan vid de låga testvärdena en dubbelt så hög uppstart och avslutstid jämfört med i Linux. Rubys kurva utmärker sig i Windows genom att till en början gå konstant, för att vid två största testvärdena (90 000 och 100 000) kraftigt stiga från 104 via 161 till 187 millisekunder.

5.2 Expressivitet

För att få en uppfattning om språkens expressivitet sinsemellan, har vi tittat på antal rader kod som krävts för att skriva testskripten. Ett lägre antal rader kod tyder på en högre expressivitet.

Vid analys av antal rader kod bör det tas i beaktande att de som skrivit testskript och testprogram inte är experter, utan till och med är nybörjare på vissa av språken. Det är alltså möjligt, och även troligt, att antal rader kod hade sett annorlunda ut om språkspecifika experter hade skrivit testskripten och testprogrammet. Eftersom programmet eller skripten som använts är relativt små är det möjligt att skillnader i expressivitet mellan språken inte utmärker sig så mycket som om uppgiften hade varit större.

Enligt Ousterhout [3], har systemspråk som C, C++ och Java 1 till 10 % av expressiviteten som skriptspråk som Tcl och Perl, se Figur 11. Detta kan vi inte styrka med hjälp av de tester vi gjort. I en jämförande undersökning av 80 olika program, gjord år 2000 av Lutz Prechelt [29], visas det att systemspråken C, C++ och Java har mellan 30 till 50 procent av skriptspråkens expressivitet. Skriptspråken i undersökningen var Tcl, Rexx, Python och Perl.



Figur 11, Expressivitet enligt Ousterhout

Våra resultat visar mer på att C har någonstans mellan 50 och 70 procent av expressiviteten för de skriptspråk vi undersökt. Denna jämförelse gäller inte PostScript som krävde fler kodrader än C.

Vi tror att skriptspråken har en större expressivitet än vad våra resultat visar på. Som vi redan nämnt kan testet vara otillfredsställande för expressivitetsutvärdering, samt att våra kunskaper inom språken får anses begränsade. Vi tror inte att skriptspråken har så mycket större expressivitet än systemspråken som Ousterhout påstår, detta grundar vi på det sätt som Ousterhout har samlat in data till sin undersökning samt den något oklara redovisningen av data. Däremot känns Prechelts resultat mer trovärdiga då han bättre redovisar hur undersökningsdatan samlats in samt påvisar tydligare hur mätningarna har gått till och granskats.

5.3 Utveckling

Skriptspråken vi har undersökt har växt fram på två olika sätt. Det ena sättet är att språket har utvecklats med ett klart syfte att redan från början vara ett språk, dessa kallar vi för språkorienterade. Det andra sättet är då ett språk växer fram från verktyg skapade för en specifik uppgift, denna kategori kallar vi för verktygsorienterade. Till de språkorienterade språken hör PostScript och Ruby. Perl och PHP är enligt vår definition verktygsorienterade. Tcl anser vi hamnar i gränslandet mellan de båda kategorierna.

PostScript utvecklades för att oberoende av hårdvara kunna beskriva komplex grafik. PostScripts utveckling har gjort det väldigt specifikt till uppgiften den ska lösa, vilket lett till att språket inte har fått någon spridning utanför dess användningsområde. Detta kan då ha lett till att språket inte har utvecklats till att bli ett brett språk som är användbart inom fler områden.

Ruby utvecklades för att dess skapare, Matz, var missnöjd med de redan existerande skriptspråken, som han inte tyckte uppfyllde hans behov. Eftersom Ruby var ett i stort sett komplett språk redan när det släpptes till allmänheten, har språket inte vidareutvecklats i samma utsträckning som t.ex. PHP som skapats senare men ändå genomgått större förändringar. Det som istället har utvecklats för Ruby är olika tilläggsmoduler som används t.ex. för att interagera med en webbserver. Det är alltså inte språket i sig som utvecklas, utan dess möjligheter att användas inom olika områden.

I PHP har utvecklingen sett annorlunda ut. Språket var ursprungligen en samling Perl-skript som användes för att räkna antalet nerladdningar av ett CV. Källkoden för dessa släpptes fri och spreds över Internet av personer som fann skripten användbara. Ny funktionalitet lades till och förslag på förbättringar från användarna drev utvecklingen av skripten till ett eget språk. Genom åren har språket genomgått stora förändringar med bland annat en omskrivning av hela språket vid ett tillfälle och en omskrivning av språkets kärna för förbättrad prestanda vid ett annat. Språket har allt sedan det skapades haft sitt främsta användningsområde inom webbutveckling, vilket lett till att språket fått en bred funktionalitet med en specialisering mot databashantering.

Perls utveckling påminner om PHP:s då språket i början var ett verktyg anpassad för en speciell uppgift. Verktygen vidareutvecklades och likt PHP fann andra personer verktygen användbara vilket ledde till utökningar av verktygen. Perl har fått fler användningsområden än PHP vilket kan bero att språket tidigt fick funktionalitet inom andra områden än det som det skapades för. Utvecklarna av Perl har också haft som mål att språket ska vara praktiskt och så komplett som möjligt vilket medfört att funktionalitet inom många områden har inkluderats. Likt PHP så har Perl genomgått omskrivningar av hela språket och fått stora tillägg.

Tcl:s utveckling är som sagt en blandning av de två utvecklingskategorierna. Tcl utvecklades av Ousterhout då han behövde ett kommandospråk som inte var specifikt för en uppgift utan kunde återanvändas. De tidiga versionerna av Tcl, var mycket små, men dock var det ett eget språk. I och med detta skulle det kunna kategoriseras som språkorienterat. Att språket var så litet från början har ändå medfört att det har fått en liknande utveckling som språken som tillhör den verktygsorienterade kategorin. Trots att Ousterhout i stort sett skrev varenda rad kod i Tcl själv fram till 1994, var utvecklingen driven av användarna och deras önskemål.

Av de språk vi undersökt är Ruby det enda helt objektorienterade språket. De övriga tillhör ursprungligen det procedurella paradigmet. Flera av dessa språk har dock fått tillägg i språken som gör att de nu stödjer även det objektorienterade paradigmet. Perl och PHP har båda fått stöd för objektorientering, medan det ur Tcl utvecklats flera språk där stöd för objektorientering finns. PostScript är enbart procedurellt.

5.4 Definition av skriptspråk

Vad definierar ett skriptspråk och vad särskiljer skriptspråk från klassiska systemspråk?

Av de egenskaper som definierar vad ett skriptspråk är, så är en av de viktigaste att programmeraren inte explicit behöver kompilera koden för att den ska bli körbar. Många av dagens skriptspråk har som en del av interpreteringen en fas där koden kompileras till ett språk på en lägre nivå, exempelvis bytekod. Detta görs antingen för att underlätta interpreteringen eller av effektivitetsskäl. För en del skriptspråk finns det möjlighet att kompilera koden till ett körbart program, men detta är då i de fall vi sett tillagt i efterhand som ett verktyg för att optimera koden och göra den mer effektiv. När kod skriven i ett skriptspråk kompileras kan det bero på att det används inom ett användningsområde som kräver hög effektivitet och som skriptspråket från början inte hade som syfte att användas för. Att programmeraren inte explicit behöver kompilera koden i ett skriptspråk före körning anser vi vara en av de saker som mest särskiljer skriptspråk från de klassiska systemspråken. Även Java som kan köras på olika plattformar, så länge det finns en Java Virtual Machine (JVM), skiljer sig från skriptspråken då Java kräver ett separat kompileringssteg. Dock så tycker vi att Java, med sin bytekod, ligger närmare skriptspåken än vad de övriga klassiska systemspråken gör.

De skriptspråk vi undersökt använder sig alla av dynamisk typbindning, vilket verkar vara en egenskap som alla skriptspråk har. Dynamisk typbindning innebär att variabler binds till en viss typ när de tilldelas ett värde. Variabelns typ kan under körning ändras om det tilldelas värden av en annan typ än den variabeln har. Detta skiljer sig mot de språk som använder sig av statisk typbindning där variabeln tilldelas en typ, vilken sedan inte kan ändras under körning [30].

5.5 Syntax

De flesta av de språk vi tittat på skiljer sig åt i hur deras syntax ser ut. Det språk som särskiljer sig mest från de andra är PostScript, som med sin postfixnotation har ett annat upplägg på sin grammatik. T.ex. hamnar parametrarna till en funktion före funktionsnamnet vid anropet. De andra språken har en mer traditionell grammatik där funktionsnamnet följs av dess parametrar, och numeriska operatorer ligger mellan dess operander. Perl och PHP har stora likheter i syntaxen med C. Av dessa kan Perl skilja sig mest från C-liknande kod, tack vare sin filosofi om att allting ska gå att göra på mer än ett sätt. Även Perls många inbyggda parametrar gör att dess syntax särskiljer sig. Detta gör att det kan krävas ett stort kunnande om sammanhanget för att förstå Perls syntax. Inte heller Ruby skiljer sig så mycket från C-liknande syntax. De största skillnaderna är avsaknaden av semikolon för att avsluta satser och klammerparenteser vid block då istället `end` används för att stänga ett block. Tcl är det språk som utmärker sig mest i sin syntax förutom PostScript. Språkets syntax påminner mycket om bash-kommandon i Linux där ett kommando använder sig av växlar för att ta in argument. I Tcl används denna syntax för att välja vilka attribut på en komponent som ska tilldelas värden.

5.6 Utvecklingstid jämfört med expressivitet

En rangordning av språkens skript efter hur många rader kod de har, med minst antal rader kod först, ser ut såhär:

1. Ruby och Perl
2. PHP
3. Tcl
4. C
5. PostScript

En jämförelse mellan ovanstående rangordning och rangordningen av skripten och programmens utvecklingstider, se avsnitt 4.6, visar att språken med högre expressivitet har en kortare utvecklingstid än de med låg expressivitet. De båda rangordningarna ser i stor sett lika ut, med skillnaderna att Perl har flyttat sig ner på utvecklingstidsrangordningen jämfört med expressivitetsrangordningen. En anledning till att Perl-skriptet tog längre tid än Ruby-skriptet, som har lika många kodrader, att utveckla kan vara Rubys syfte att vara enkelt att skriva och förstå jämfört med Perls något komplexa syntax med mycket inbyggda variabler.

PHP:s framskjutna position i utvecklingstid kan till viss del bero på programmerarnas tidigare kunskaper och vana inom språket. Det motsatta kan sägas om utvecklingstiden för PostScript. Inte bara var det ett nytt språk utan även en annorlunda notation, postfixnotation, vilket krävde ett ovant sätt att tänka.

Utvecklingstiden för C känns hög med tanke på att programmerarna hade tidigare erfarenheter av språket. Till del berodde detta på problem vid tidsmätningen av inläsning och sortering, men det belyser ändå skriptspråkens effektivare utvecklingstakt.

6 Slutsats

Skriptspråk har det senaste decenniet fått en ökad spridning tack vare sin lämplighet för användning mot webben. Denna spridning har lett till en ökad använding av skriptspråk inom många områden där tidigare enbart klassiska systemspråk använts. Även allt bättre prestanda på datorer har bidragit till skriptspråkens spridning, då effektivitetskraven är lättare att uppfylla även för skriptspråken.

Av vår analys har vi kommit fram till att den största skillnaden mellan skriptspråken och de klassiska systemspråken är att skriptspråken inte har någon separat steg för kompilering så som systemspråken har. Många skriptspråk kompilerar skripten till ett språk på en lägre nivå som sedan interpreteras, medan andra är rent interpreterade och läser och exekverar skripten rad för rad. Typiskt för skriptspråken är också att de använder dynamisk typbindning för variabler. Vi tror att avsaknaden av ett separat kompileringssteg tillsammans med dynamisk typbindning bidrar till att utvecklingstiden för skriptspråk är lägre än för klassiska systemspråk. Eftersom utvecklarna inte behöver kompilera koden får de snabbare respons på sitt arbete, och den dynamiska typbindningen tillåter utvecklarna att koncentrera sig mer på vad de ska göra med variablerna än vilken typ de ska ha.

Undersökningen av skriptspråkens utvecklingen har lett oss till att kategorisera språken i två grupper: språkorienterade och verktygsorienterade. De språkorienterade skriptspråken var redan från början tänkta som kompletta språk medan de verktygsorienterade har växt fram och sakta utvecklats till kompletta språk från att från början varit skapat som ett verktyg. PostScript och Ruby hamnar i den grupp vi kallar språkorienterade. Perl och PHP hamnar i den verktygsorienterade gruppen. Tcl ligger mellan de två grupperna då språket har egenskaper och en utveckling som passar in i båda kategorierna. Språken i den verktygsorienterade gruppen har haft en mer omfattande utveckling och genomgått större förändringar än språken i den språkorienterade gruppen.

De tester vi gjorde av språken visar på en stor skillnad mellan systemspråken, i det här fallet C, och skripspråken vad gäller effektivitet vid exekvering. Systemspråken är klart snabbare vid exekvering än skriptspråken. Av skriptspråken är Perl effektivast tätt följt av Tcl.

PostSkript klarade av testerna sämst förutom vid inläsningen från fil, där PostScript är snabbast av skriptspråken.

Gällande expressivitet visar våra tester att systemspråken har 50-70 % av skriptspråkens expressivitet. Dessa resultat ligger långt ifrån Ousterhouts undersökning där systemspråken har 1-10 % av skriptspråkens expressivitet. Prechelts undersökning ligger närmare våra resultat, då hans tester visar att systemspråken har 30-50 % av skriptspråkens expressivitet.

Referenser

[1] TIOBE Programming Community Index, http://www.tiobe.com/tpci.htm, 2006-02-20

[2] Barrow, David. The world of scripting languages, 2000, Wiley

[3] Ousterhout, John K. Scripting: Higher Level Programming for the 21st Century. Publicerad i IEEE Computer Magazine, mars 1998.

[4] Surveyer, Jaques. The Irony of JavaScript's Success, http://www.webreference.com/programming/javascript/j_s/column8/, 2005-09-21

[5] Morin, Rich och Brown, Ricki. Scripting Languages, A cross-OS perspective, http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/, 2005-10-18

[6] Kay, Russell. Follow the Script, http://www.computerworld.com/developmenttopics/development/story/0,10801,101392,00.html, 2005-09-21

[7] Hatcher, Paul. JavaScript professional projects, 2003, Premier Press

[8] PHP: Hypertext Preprocessor, http://www.php.net, 2005-09-06

[9] Brogdon, Darrel. Securing a PHP Installation, http://www.onlamp.com/pub/a/php/2001/03/29/php_admin.html, 2006-01-20

[10] Reid, Glenn C, PostScript language program design Adobe Systems Incorporated, 1988, Addison-Wesley

[11] Adobe Systems Incorporated, PostScript Language Reference, third edition, 1999, Addison-Wesley

[12] Adobe Systems Incorporated, PostScript Language Tutorial and Cookbook, 1985, Addison-Wesley

[13] Feldt, Johnson och Neumann. Ruby Developers Guide 2002, Syngress

[14] What is the history of Ruby, http://www.rubygarden.org/faq/entry/show/5, 2005-09-25

[15] Thomas, Dave. Programming Ruby, the pragmatic programmer's guide 2005, Pragmatic Bookshelf

[16] Slagell, Mark, Ruby user's guide. http://www.rubyist.net/~slagell/ruby/control.html, 2005-10-14

[17] The Ruby Language FAQ, http://www.rubycentral.com/faq/rubyfaq-8.html, 2005-11-16

[18] Ousterhout, John. History of Tcl, http://www.tcl.tk/about/history.html, 2005-11-04

[19] Expect homepage, http://expect.nist.gov/, 2005-11-04

[20] Tcl Built-in Commands – expr manual page, http://www.tcl.tk/man/tcl8.4/TclCmd/expr.htm, 2006-01-18

[21] Tcl manual pages, http://www.tcl.tk/man/tcl8.4/TclCmd/while.htm, 2005-11-08

[22] Nelson, Christopher. Tcl/Tk Programmers Reference. 2000, Osbourne/McGraw-Hill

[23] Intro to Tcl: Procedures, http://www.lib.uchicago.edu/keith/tcl-course/topics/procedures.html, 2005-11-21

[24] About Perl – perl.org, www.perl.org/about.html, 2006-01-12

[25] Brown, Martin C., Perl: The Complete Reference, second edition, 2001, Osborne/McGraw-Hill

[26] Steven Holzner, PERL Black Book, Second edition, 2001, Coriolis

[27] Simon Cozens, Peter Wainwright, Beginning Perl, first edition, 2000, Wrox Press

[28] Weiss, Mark Allen. Data structures & algorithm analysis in C++, second Edition 1999, Addison Wesley

[29] Prechelt, Lutz. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl, 2000

[30] Sebesta W. Robert, Concepts of Programming Languages, fifth edition, 2002, Addison-Wesley

A Testskript

A.1 Testskript i PHP

```
<?php
/*********************************************************************/
/*Quicksortfunktionen                                                */
/*********************************************************************/
function quicksort($arr){
    $length = count($arr);
    if($length <= 1){
    return $arr;
    }else{
        $first = $arr[0];
        $middle = $arr[round($length/2)];
        $last = $arr[$length-1];
        if(($first > $last && $first < $middle) || ($first < $last
&& $first > $middle)){
            $pivot = $first;
        }else if(($last > $first && $last < $middle) || ($last <
$first && $last > $middle)){
            $pivot = $last;
        }else if(($middle > $last && $middle < $first) || ($middle <
$last && $middle > $last)){
            $pivot = $middle;
        }else{
            $pivot = $first;
        }
    $smaller = array();
    $greater = array();
    $equal = array();
    for($i=0;$i<$length;$i++){
        if($arr[$i]<$pivot){
            $smaller[count($smaller)] = $arr[$i];
            }else if($arr[$i]>$pivot){
```

```
            $greater[count($greater)] = $arr[$i];
            }else{
                $equal[count($equal)] = $arr[$i];
            }
        }
    }
    return array_merge(quicksort($smaller), $equal,
quicksort($greater));
}
/*****************************************************************/
/* Läser in en array från filen $filename                        */
/*****************************************************************/
function getArrayFromFile($filename){
    if(($arr = file($filename)) == false){
    echo "File \"$filename\" doesn't exist. Aborting...\r\n";
    exit();
    }
    $arr2 = array();
    for($i=0;$i<count($arr);$i++){
    $arr2[$i] = (int)$arr[$i];
    }
    echo count($arr) . "\t";
    return $arr2;
}
/*****************************************************************/
/* "Main"-delen                                                  */
/*****************************************************************/
if(count($argv) == 2){
    $filename = $argv[1];
    $before_file = microtime(true);
    $arr = getArrayFromFile($filename);
    $after_file = microtime(true);
    $difference_file = $after_file - $before_file;
    echo (round($difference_file*1000)) . "\t";
    $before = microtime(true);
    $arr2 = quicksort($arr);
    $after = microtime(true);
```

```
    $difference = $after - $before;
    echo (round($difference*1000)) . "\t";
}else{
    echo "Wrong number of arguments. Usage: php quicksort.php
<filename>\r\n";
}
/*****************************************************************/
?>
```

A.2 Testskript i PostScript

```
%!
/$QSortDict 10 dict def
/QSortArray  {    % [ array ] => [ array' ]
    dup
    qsortrange
} bind def
/qsortrange  {    % [ array ] => ---
    dup length dup 1 le
    { pop pop }      % done
    {            % else
        2 eq% check for simple case
        {        %   if
            aload 3 1 roll
            2 copy gt
            { exch 3 -1 roll astore pop }
            { pop pop pop }
            ifelse
        }
        {        %   else (not simple)
            PartitionArray1
            qsortrange
            qsortrange
        }
        ifelse
    } ifelse
} bind def
/PartitionArray1 {    % [a] => [ a-left ] [ a-right ]
```

```
//$QSortDict begin

/a exch def
/r a length 1 sub def
/v a r get def
/i 0 def
/j r def     % First time only
/firstTime true def
{                % Main loop
    i 1  j
    {    % Left loop
        a 1 index get
        v ge 1 index j eq or { exit } if
        pop
    } for
    /i exch def
    firstTime
    { i r eq     % is the target the largest number?
        { exit }          % Yes: leave
        { /j r 1 sub def } % No: Reset j
        ifelse
    } if
    j -1 i   {    % Right loop

        a 1 index get
        v le 1 index i eq or { exit } if
        pop
    } for
    /j exch def
    i j lt
    {            % if: swap a[i] & a[j]
        a dup i get
        a dup j get
        i exch put
        j exch put
    }
    { exit } % else: leave
```

```
        ifelse
    } loop
    a dup i get      % exch a[i] & a[r]
    r exch put
    a i v put
    a 0 i getinterval     % Leave subarrays on stack
    i r ne
    { a i 1 add r i sub getinterval }
    { [ ] }
    ifelse
    end          % $QSortDict
} bind def
/starttimer {usertime /mytimenow exch def} def
/stoptimer {usertime mytimenow sub /mytime exch mytime
add def} def
/myfile3 (test_output.txt) (w+) file def
/resettimer {/mytime 0 def} def
/reporttimer {mytime (\r\nElapsed time: ) print 20 string cvs print
( ms.\r\n) print flush} def
/stopwatchon {resettimer starttimer} def
/stopwatchoff {stoptimer reporttimer} def
/myfile (test1.txt) (r) file def
/mystring 10 string def
/mymaxsize 25000 def
/size mymaxsize def
/myarray size array def
/myfile2 (test1.txt) (r) file def
stopwatchon
0 1 size 1 sub {myfile2 mystring readline { pop /mysize exch cvi def
myarray mysize mystring cvi put } {exit} ifelse } for
stopwatchoff
myarray
stopwatchon
QSortArray
stopwatchoff
quit
```

A.3 Testskript i Ruby

```
# ######################################################################
# Sorterar en array med quicksort-algoritmen
# ######################################################################
def quicksort( arr )
    return arr if arr.size <= 1
    first = arr[0]
    middle = arr[(arr.size/2).round]
    last = arr[arr.size-1]
    # Väljer pivotelement att sortera efter
    if (first > last && first < middle) || (first < last && first >
middle)
        m = first
        elsif (last > first && last < middle) || (last <   first &&
last > middle)
        m = last
    elsif (middle > last && middle < first) || (middle < last &&
middle > last)
        m = middle
        else
            m = first
            end
return quicksort( arr.select { |i| i < m } ) + arr.select { |i| i ==
m } + quicksort( arr.select { |i| i > m } )
end
# ######################################################################
# Läser från filen filename och returnerar en array med dess
innehåll
# ######################################################################
def getArrayFromFile( filename )
    arr = []
    test = File.open(filename)
    test.each { |line|
        arr.push(line.to_i)
    }
    print arr.length.to_s + "\t"
    test.close
```

```
    return arr
end
# ######################################################################
# "Main"-del
# ######################################################################
if ARGV[0] != nil
    filename = ARGV[0]
else
    filename = "50000-1.txt"
end
$stdout.flush
time_file1 = Time.now
arr = getArrayFromFile( filename )          # Hämtar array från fil
time_file2 = Time.now
             #Beräknar tidsåtgång
time_file3 =
((time_file2.min*60000000+time_file2.sec*1000000+time_file2.usec) -
(time_file1.min*60000000+time_file1.sec *
1000000+time_file1.usec))/1000
  # Skriver ut hur lång tid inläsningen från fil tog i millisekunder
print time_file3.to_s + "\t"
$stdout.flush
time1 = Time.now
arr2 =  quicksort(arr)                          # Sorterar arrayen
time2 = Time.now
         #Beräknar tidsåtgång
time3 = ((time2.min*60000000+time2.sec*1000000+time2.usec) -
(time1.min*60000000+time1.sec*1000000+time1.usec))/1000
# Skriver ut hur lång tid som har gått i millisekunder
print time3.to_s + "\t"
$stdout.flush
# ######################################################################
```

A.4 Testskript i Perl

```
#! perl
#######################################################################
# Quicksortfunktionen                                                 #
```

```
#####################################################################
sub qsort {
    @_ or return ();
    $first = @_[0];
    $last = @_[@_ - 1];
    $middle_pos = int((@_/2) +.5 );
    $middle = @_[$middle_pos-1];
    my $p;
    if(($first > $last && $first < $middle) || ($first < $last &&
$first > $middle)){
        $p = $first;
    }elsif(($last > $first && $last < $middle) || ($last < $first &&
$last > $middle)){
        $p = $last;
    }elsif(($middle > $last && $middle < $first) || ($middle < $last
&& $middle > $first)){
        $p = $middle;
    }else{
        $p = $first
    }
    print "piv=$p\n";
    (qsort(grep $_ < $p, @_), grep($_ == $p, @_), qsort(grep $_ >
$p, @_));
}
#####################################################################
# "Main"-delen                                                      #
#####################################################################
if($#ARGV == 0){
    $FILE_NAME = @ARGV[0];
}
else{
    print "Wrong number of arguments. Usage: perl quicksort.pl
<filename>\n";
    exit "error";
}
use Time::HiRes qw(gettimeofday);
($before_file_Seconds, $before_file_MicroSeconds) = gettimeofday;
```

```
open(FILE, "< $FILE_NAME") || die "Could not open file\n";
for($i = 0; <FILE>; $i++){
    @arr[$i] = substr($_, 0, -1);
}
print @arr . "\t";
($after_file_Seconds, $after_file_MicroSeconds) = gettimeofday;
$after_file_Time = ($after_file_Seconds * 1000000) +
$after_file_MicroSeconds;
$before_file_Time = ($before_file_Seconds * 1000000) +
$before_file_MicroSeconds;
print(int((($after_file_Time - $before_file_Time)/1000)) . "\t" );
($beforeSeconds, $beforeMicroSeconds) = gettimeofday;
@test = qsort(@arr);
($afterSeconds, $afterMicroSeconds) = gettimeofday;
$afterTime = ($afterSeconds * 1000000) + $afterMicroSeconds;
$beforeTime = ($beforeSeconds * 1000000) + $beforeMicroSeconds;
print(int((($afterTime - $beforeTime)/1000)) . "\t" );
```

A.5 Testskript i Tcl

```
#!/tcl/bin/tclsh
#####################################################################
# Läser talen in från filen fname                                   #
#####################################################################
proc getArrayFromFile {arr fname} {
    upvar $arr arr2
    set fp [open $fname r]
    set data [read $fp]
    close $fp
    set count 0
    set data [split $data "\n"]
    foreach line $data {
        set arr2($count) $line
        incr count
    }
    puts -nonewline "$count\t"
}
#####################################################################
```

```
# Quicksortfunktionen                                                    #
#########################################################################
proc quicksort {arr l p} {
    upvar $arr arr2
    set first $arr2($l)
    set last $arr2($p)
    set middle $arr2([expr ( $p - $l ) / 2 ])
    if {$first > $last && $first < $middle || $first < $last &&
$first > $middle} {
        set x $first
    } elseif {$last > $first && $last < $middle || $last < $first
&& $last > $middle} {
        set x $last
    } elseif {$middle > $first && $middle < $last || $middle <
$first  && $middle > $last} {
        set x $middle
    } else {
        set x $first
    }
    set i $l
    set j $p
    while {$j > $i} {
        while {$x > $arr2($i)} {
            incr i
        }
        while {$x < $arr2($j)} {
            incr j -1
        }
        if {$i <= $j} {
            set w $arr2($i)
            set arr2($i) $arr2($j)
            set arr2($j) $w
            incr i
            incr j -1
        }
    }
    if {$l < $j} {quicksort arr2 $l $j}
```

```
    if {$i < $p} {quicksort arr2 $i $p}
}
#########################################################################
# "Main"-delen                                                          #
#########################################################################
if {$::argc == 1} {
    set fname $argv
    array set arr {}
    set t_file [time {getArrayFromFile arr $fname}]
    set index [string first " " $t_file]
    set t_file2 [expr [string range $t_file 0 [expr $index - 1]] /
1000]
    puts -nonewline "$t_file2\t"
    set arr_size [array size arr]
    set time1 [clock seconds]
    set t [time { quicksort arr 0 [expr $arr_size - 1]}]
    set index [string first " " $t]
    set t2 [expr [string range $t 0 [expr $index - 1]] / 1000]
    puts -nonewline "$t2 \t"
} else {
    puts "Wrong number of arguments. Usage: tclsh quicksort.tcl
filename"
}
#########################################################################
```

A.6 Testprogram i C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <math.h>
int readFile(int* arr, char* fname){
  FILE *infile;
  char line[100];
  int lcount = 0;
  int arr_counter = 0;
  if((infile = fopen(fname, "r")) == NULL) {
```

```
    printf("Error Opening File.\n");
    exit(1);
  }
  while( fgets(line, sizeof(line), infile) != NULL ) {
    arr[lcount++] = atoi(line);
  }
  fclose(infile);
  return lcount;
}
int partition(int y[], int f, int l){
    int up,down,temp, last = y[l], first = y[f], middle, piv;
    middle = y[(int)floor((l-f)/2.0)];
    if((first > last && first < middle) || (first < last && first >
middle)){
        piv = first;
    }else if((last > first && last < middle) || (last < first &&
last > middle)){
        piv = last;
    }else if((middle > last && middle < first) || (middle < last &&
middle > first)){
        piv = middle;
    }else{
        piv = first;
    }
    up = f;
    down = l;
    do {
        while (y[up] <= piv && up < l){
            up++;
        }
        while (y[down] > piv  ){
            down--;
        }
        if (up < down ){
            temp = y[up];
            y[up] = y[down];
            y[down] = temp;
```

```
        }
    }while (down > up);
    temp = piv;
    y[f] = y[down];
    y[down] = piv;
    return down;
}
void quicksort(int x[], int first, int last) {
    int pivIndex = 0;
    if(first < last) {
        pivIndex = partition(x,first, last);
        quicksort(x,first,(pivIndex-1));
        quicksort(x,(pivIndex+1),last);
    }
}
void printArray(int* arr, int size){
     int counter = 0;
     while(counter<size){
         printf("%i\n", arr[counter++]);
     }
}
int main(int argc, char *argv[]){
  int arr[100001];
    int size, i;
    double test;
    clock_t start, ends;
    if(argc < 2){
        printf("Wrong number of arguments!");
    }else{
        start = clock();
        size = readFile(arr, argv[1]);
        ends = clock();
        test = ((double) (ends - start) / CLOCKS_PER_SEC*1000);
        printf("%d\t%.0f\t", size, test);
        start = clock();
        quicksort(arr, 0, size-1);
        ends = clock();
```

```
        test = ((double) (ends - start) / CLOCKS_PER_SEC*1000);
        printf("%.0f\t", test);
         return 0;
    }
  system("PAUSE");
  return 0;
}
```

A.7 Yttre testskript i PHP

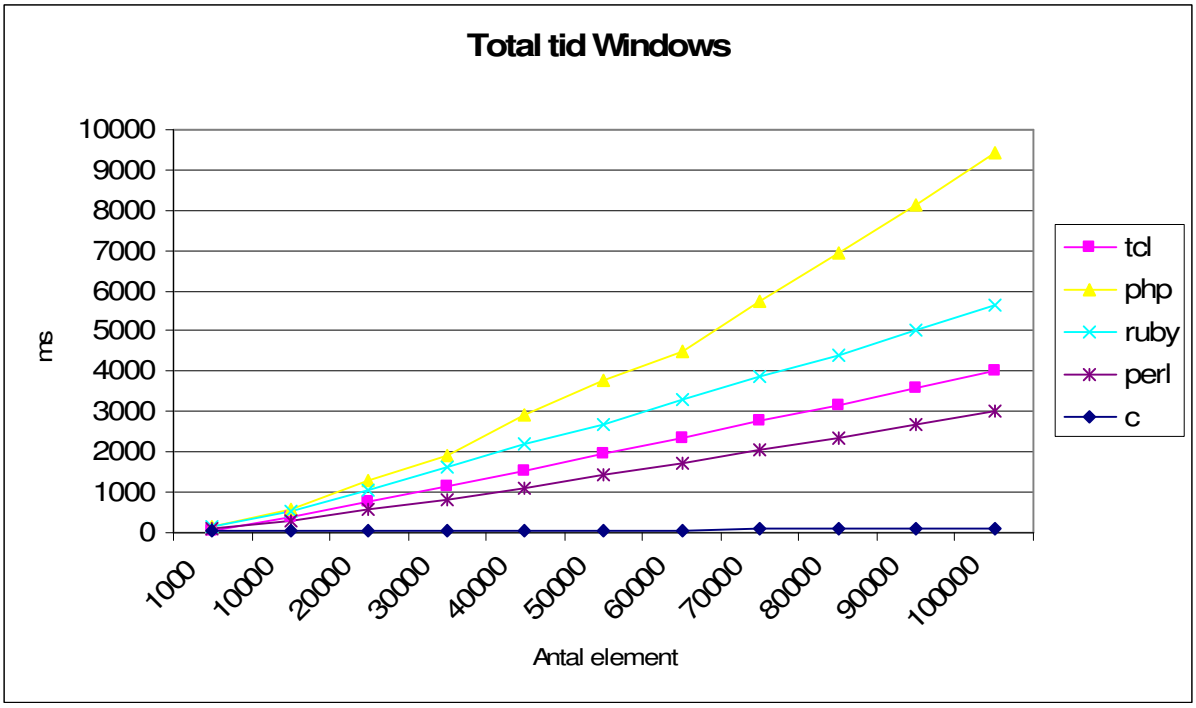
```
<?php

$total_before = microtime(true);

if(count($argv) == 2){
    echo "                  **********************************\r\n";
    echo "                  **** Scripting language test ****\r\n";
    echo "                  **********************************\r\n";
    $language = $argv[1];
    if($language == "perl"){
        $interpreter = "perl";
    $ending = "pl";
    }else if($language == "php"){
        $interpreter = "php";
    $ending = "php";
    }else if($language == "ruby"){
        $interpreter = "ruby";
        $ending = "rb";
    }else if($language == "tcl"){
        $interpreter = "tclsh";
        $ending = "tcl";
    }else if($language == "postscript"){
        $interpreter = "gs";
        $ending = "ps";
    }else if($language == "c"){
        $interpreter = "";
```

```
    $ending = "exe";
    }else{
        echo "Incompatible argument \"$language\". Aborting...\r\n";
        exit();
    }

    $testfiles = array(1000, 10000, 20000, 30000, 40000, 50000,
60000, 70000, 80000, 90000, 100000);

    if($language != "postscript"){
    foreach($testfiles as $size){
        for($i = 1; $i<10; $i++){
            $file = $size . "-" . $i . ".txt";
            $before = microtime(true);
            system($interpreter." quicksort.".$ending." ".$file);
            $after = microtime(true);
            $difference = $after - $before;
            echo "\t".(round($difference*1000))."\t$language\r\n";
            }
        }
    }else{
        $testfiles = array(1000, 5000, 10000, 15000, 20000, 25000);
        foreach($testfiles as $size) {
            $before = microtime(true);
            system("\gs\gs8.15\bin\gswin32.exe -sstdout =
test_ps$size.txt -q quicksort$size.ps");
            $after = microtime(true);
            $difference = $after - $before;
            echo "\t".(round($difference*1000))."\t$language\r\n";
        }
    }
}else{
    echo "Wrong number of arguments. Usage: php testscript.php
<language>\n";
}

    $total_after = microtime(true);
```

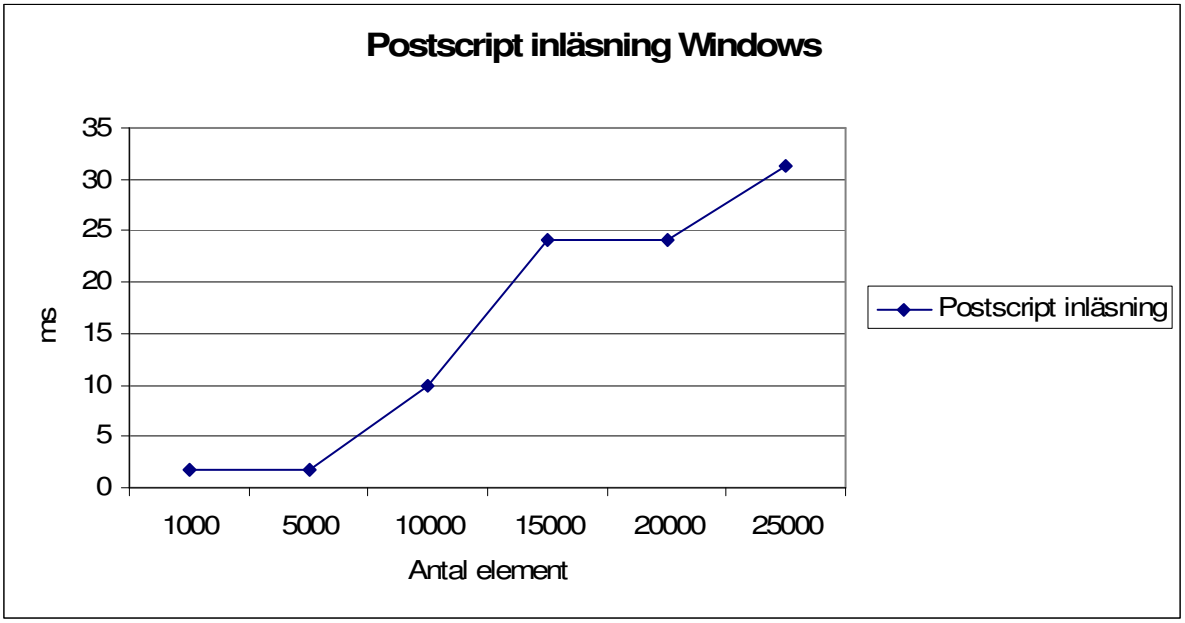
```
    $total_difference = $total_after - $total_before;
    echo "Total $language test took " .
(round($total_difference*1000)) . "ms\r\n";
?>
```

B Testresultat

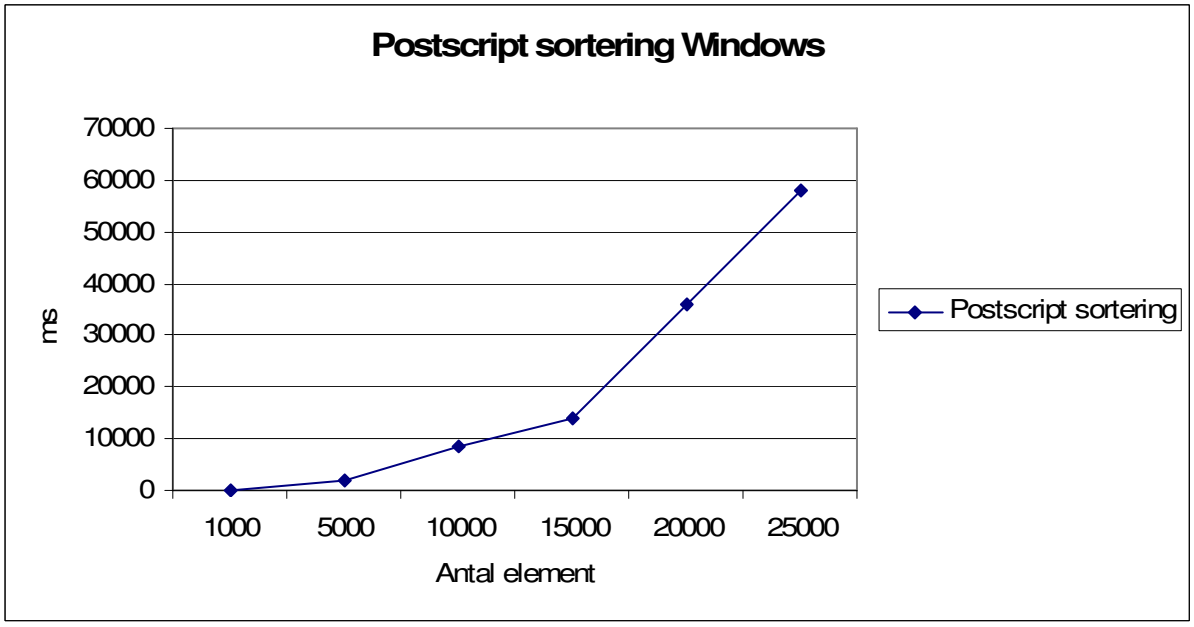
Alla tider som nämns i testresultaten är i millisekunder.



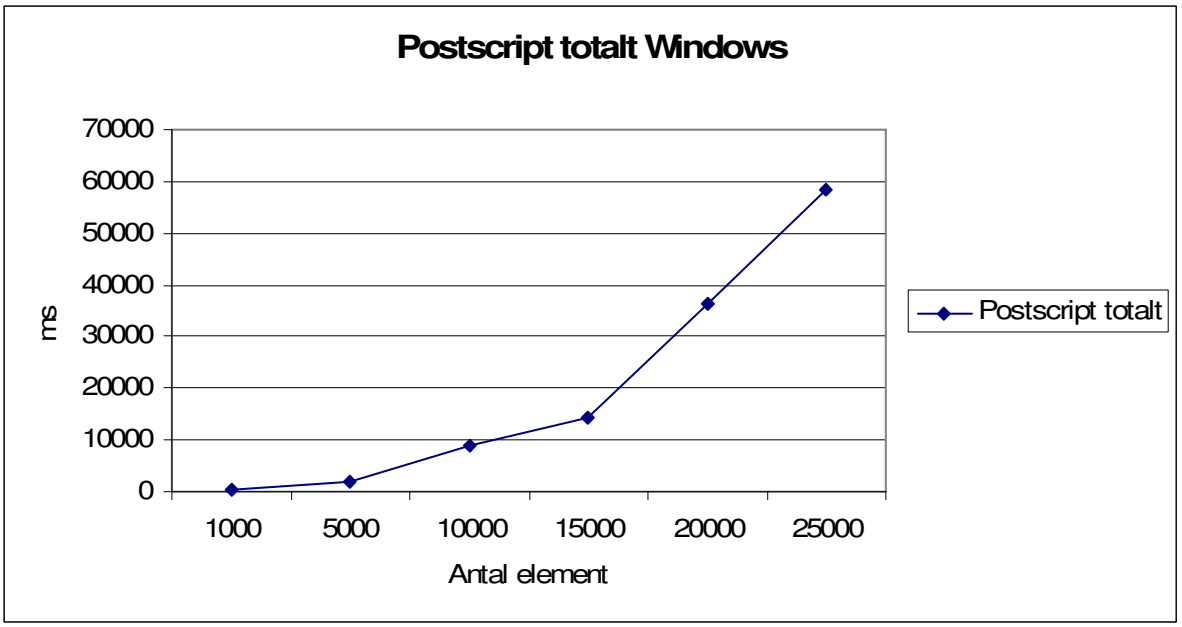
Figur 12, Total tid Windows



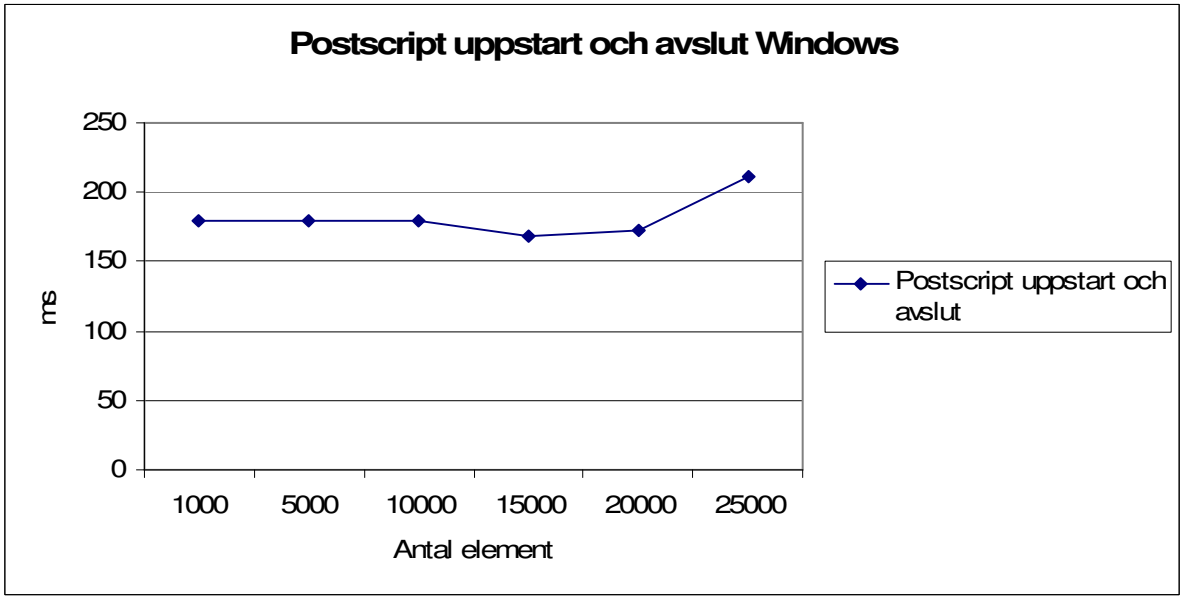
Figur 13, Postscript inläsning Windows



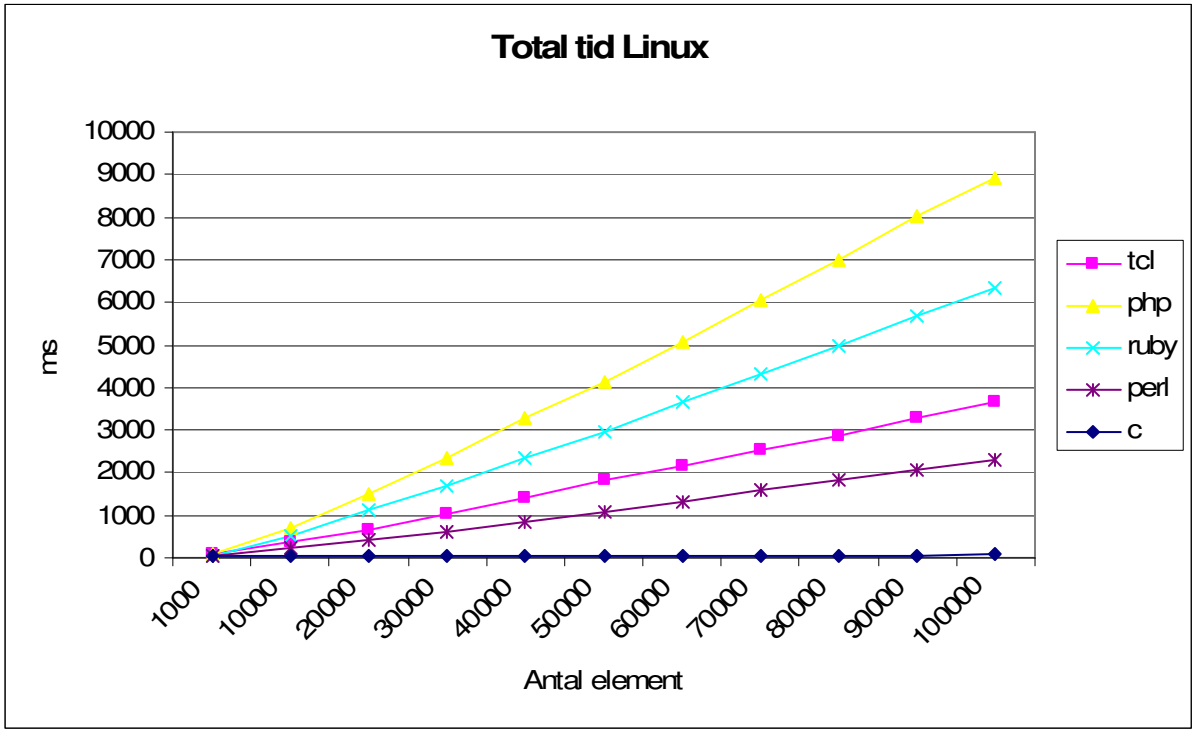
Figur 14, Postscript sortering Windows



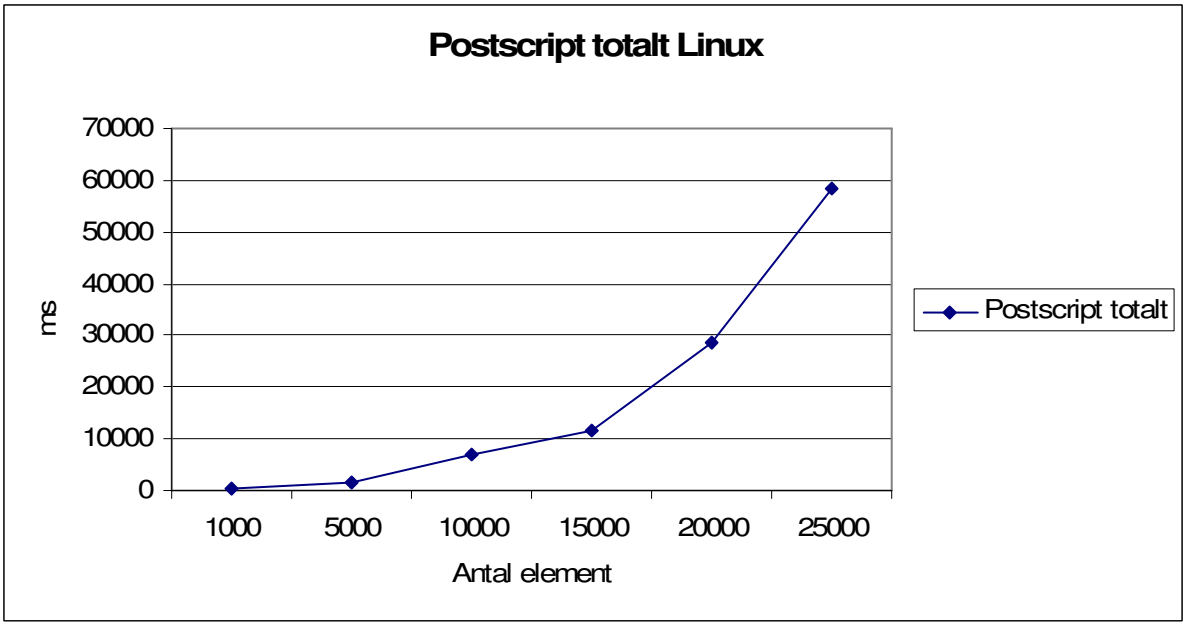
Figur 15, Postscript totalt Windows



Figur 16, Postscript uppstart och avslut Windows



Figur 17, Total tid Linux



Figur 18, Postscript total Linux

Inläsning Windows

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	2	14	30	45	58	78	91	106	122	136	149	830
php	4	41	88	131	178	218	263	322	361	402	440	2448
ruby	7	33	101	194	265	261	337	375	411	417	444	2845
perl	2	27	55	82	110	137	164	192	221	248	274	1513
c	2	2	5	5	10	15	15	17	19	22	24	135

Tabell 1, Testresultat Windows

Sortering Windows

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	27	319	676	1051	1440	1841	2224	2621	2994	3406	3828	20427
php	34	473	1102	1677	2611	3410	4091	5246	6399	7540	8794	41379
ruby	36	402	856	1313	1827	2316	2840	3398	3910	4453	5004	26354
perl	15	185	403	634	887	1164	1429	1707	1983	2268	2558	13234
c	0	2	7	10	14	16	19	24	26	29	36	183

Tabell 2, Sortering Windows

Total tid Windows

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	65	372	746	1138	1540	1958	2354	2766	3156	3586	4020	21702
php	123	594	1283	1915	2908	3760	4501	5726	6937	8126	9443	45318
ruby	140	545	1059	1615	2193	2687	3279	3880	4424	5031	5635	30487
perl	98	300	556	825	1117	1428	1726	2044	2358	2679	3011	16142
c	32	39	43	49	57	66	66	77	81	85	97	691

Tabell 3, Total tid Windows

Uppstart och avslut Windows

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	37	39	40	42	42	39	39	39	40	44	43	444
php	85	79	93	107	119	132	147	159	176	185	209	1491
ruby	97	109	103	108	101	110	101	107	104	161	187	1288
perl	80	88	98	109	120	127	134	144	154	163	178	1395
c	30	35	31	34	33	35	32	36	36	34	36	373

Tabell 4, Uppstart och avslut Windows

Postscript inläsning Windows

Antal elemment	1000	5000	10000	15000	20000	25000
Tid	2	2	10	24	24	31

Tabell 5, Postscript inläsning Windows

Postscript sortering Windows

Antal elemment	1000	5000	10000	15000	20000	25000
Tid	61	1863	8684	13953	36111	58071

Tabell 6, Postscript sortering Windows

Postscript totalt Windows

Antal elemment	1000	5000	10000	15000	20000	25000
Tid	241	2045	8873	14146	36308	58314

Tabell 7, Postscript totalt Windows

Postscript uppstart och avslut Windows

Antal elemment	1000	5000	10000	15000	20000	25000
Tid	179	180	179	168	173	211

Tabell 8, Posstscript uppstart och avslut Windows

Inläsning Linux

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	2	15	30	45	65	82	96	110	126	141	155	867
php	4	51	100	163	237	285	310	394	468	502	553	3068
ruby	2	29	47	73	96	123	145	169	197	218	243	1341
perl	1	21	43	66	90	116	135	163	187	215	242	1276
c	0	6	2	7	10	10	10	10	11	16	18	99

Tabell 9, Inläsning Linux

Sortering Linux

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	47	311	618	959	1325	1743	2012	2382	2704	3093	3466	18659
php	45	610	1351	2098	2956	3780	4664	5558	6427	7384	8221	43093
ruby	38	480	1036	1607	2227	2822	3477	4134	4762	5414	6060	32056
perl	14	157	335	524	724	928	1138	1359	1566	1783	1992	10521
c	1	3	12	10	10	17	20	26	29	32	32	192

Tabell 10, Sortering Linux

Total tid Linux

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	83	354	675	1033	1419	1850	2141	2527	2865	3272	3661	19880
php	91	707	1508	2330	3272	4151	5074	6056	7005	8006	8904	47104
ruby	69	530	1112	1710	2352	2975	3653	4334	4989	5664	6335	33721
perl	50	214	415	630	859	1094	1324	1576	1810	2056	2296	12324
c	31	36	37	33	44	45	50	55,56	61	67	73	532

Tabell 11, Total tid Linux

Uppstart och avslut Linux

Språk	1000	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	Total tid
tcl	34	27	27	29	29	25	34	35	36	38	39	355
php	42	46	57	69	79	86	99	104	111	120	130	943
ruby	29	21	28	30	30	29	31	32	31	32	32	324
perl	35	36	37	40	45	50	52	54	57	59	62	527
c	30	27	23	16	24	19	20	20	21	19	23	241

Tabell 12, Uppstart och avslut Linux

Postscript inläsning Linux

Antal elemment	1000	5000	10000	15000	20000	25000
Tid	0	4	8	13	21	26

Tabell 13, Postscript inläsning Linux

Postscript sortering Linux						
Antal elemment	**1000**	**5000**	**10000**	**15000**	**20000**	**25000**
Tid	50	1319	6517	10997	27932	57487

Tabell 14, Postscript sortering Linux

Postscript totalt Linux						
Antal elemment	**1000**	**5000**	**10000**	**15000**	**20000**	**25000**
Tid	279	1649	7011	11562	28795	58554

Tabell 15, Postscript total Linux

Postscript uppstart och avslut Linux						
Antal elemment	**1000**	**5000**	**10000**	**15000**	**20000**	**25000**
Tid	229	326	486	552	841	1041

Tabell 16, Postscript uppstart och avslut Linux

Antal rader kod	
Tcl	64
PHP	61
Ruby	43
Perl	43
PostScript	99
c	85

Tabell 17, Antal rader kod