

Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java

A chapter for *Advances in Computers*, Volume 58

Lutz Prechelt (prechelt@computer.org)
Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe
Germany
<http://wwwipd.ira.uka.de/~prechelt/>

August 18, 2002

Abstract

Four scripting languages are introduced shortly and their theoretical and purported characteristics are discussed and related to three more conventional programming languages. Then the comparison is extended to an objective empirical one using 80 implementations of the same set of requirements, created by 74 different programmers. The limitations of the empirical data are laid out and discussed and then the 80 implementations are compared for several properties, such as run time, memory consumption, source text length, comment density, program structure, reliability, and the amount of effort required for writing them. The results indicate that, for the given programming problem, “scripting languages” (Perl, Python, Rexx, Tcl) are more productive than conventional languages. In terms of run time and memory consumption, they often turn out better than Java and not much worse than C or C++. In general, the differences between languages tend to be smaller than the typical differences due to different programmers within the same language.

Contents

1	On scripting languages	4
1.1	What is a scripting language?	4
1.2	Language comparisons: State of the practice	5
1.3	The structure of this study	6
2	The languages	7
2.1	C, C++, Java	7
2.2	Perl	8
2.3	Python	9
2.4	Rexx	10
2.5	Tcl	10
3	Overview of the empirical study	12
3.1	Short overview	12
3.2	The programming task ‘phonecode’	12
3.3	Origin of the C, C++, and Java programs	14
3.4	Origin of the Perl, Python, Rexx, and Tcl programs	14
3.5	Number of programs	15
3.6	Validity: Are these programs comparable?	15
3.6.1	Programmer capabilities	16
3.6.2	Work time reporting accuracy	16
3.6.3	Different task and different work conditions	17
3.6.4	Handling a misunderstood requirement	18
3.6.5	Other issues	19
3.6.6	Summary	19
3.7	Evaluation conditions	19
3.8	External validity: To what degree can these results be generalized?	20
3.8.1	Stability of results over time	20
3.8.2	Generalizability of results to other programming tasks	21
3.9	Evaluation tools: Data plots and statistical methods	21
4	Results for execution time	22
4.1	Total run time: z1000 data set	22
4.1.1	Results	22
4.1.2	Discussion	24
4.2	Run time for initialization phase only: z0 data set	25
4.2.1	Results	25
4.2.2	Discussion	27
4.3	Run time for search phase only	27
4.3.1	Results	27
4.3.2	Discussion	29
5	Results for memory consumption	29
5.1	Results	29

<i>CONTENTS</i>	3
5.2 Discussion	30
6 Results for program length and amount of commenting	31
6.1 Results	31
6.2 Discussion	33
7 Results for program reliability	33
7.1 Results for “standard” inputs	33
7.2 Discussion	35
7.3 Results for “surprising” inputs	35
7.4 Discussion	35
8 Results for programmer work time	37
8.1 Results	37
8.2 Discussion	38
8.3 Validation	39
9 Discussion of program structure	43
9.1 Results for scripts	43
9.2 Discussion	43
9.3 Results for non-scripts	43
9.4 Global discussion	45
10 Testing two common rules of thumb	46
10.1 The run time versus memory consumption tradeoff	46
10.2 Are shorter programs also more efficient?	48
11 Metrics for predicting programmer performance	49
11.1 Results for objective metrics	49
11.2 Results for programmer self-rating	50
12 Conclusions	50
12.1 Further reading	53
A Appendix: Specification of the phonecode programming problem	53
A.1 The procedure description	53
A.2 Task requirements description	55
A.3 The hint	58
B Appendix: Raw data	58
Bibliography	61

1 On scripting languages

Historically, the state of the programming practice has continually moved towards ever higher-level languages; for instance there is a mainstream that started with machine code and then progressed first to assembler, then to Fortran and Cobol, then to C and Pascal, then to C++, and then to Java.

Whenever the next step on this ladder was about to occur, the new candidate mainstream language was initially accused of not being practical. The accusations always claimed an unacceptable lack of runtime and memory efficiency and sometimes also that the language was too complicated for programmers. However, the following happened in each case: After a while (when compilers and runtime systems had matured, computers had gotten still faster, and programmers had had some time to learn the languages and related design methods and programming styles) it turned out that the runtime and memory efficiency problems were neither as big nor as relevant as had been believed and that the higher language level had rather beneficial effects on programmer efficiency. Consequently, the older mainstream representatives have slowly but surely started to go out of business — at least since the advent of C++.

This chapter attempts to evaluate whether or in what respect scripting languages represent the next step on this ladder and how far they have progressed towards refuting the usual counterarguments. The evaluation will ground in an empirical comparison of 80 separate implementations of the same specification (each written in either of 4 scripting languages or 3 “conventional” languages) and relate the empirical results to claims made by advocates or opponents of each scripting language and to a theoretical analysis of language properties to be plausibly expected.

1.1 What is a scripting language?

What makes a language a scripting language, anyway? And what is the opposite of a scripting language? There are at least two possible perspectives from which these questions can be answered: Either in terms of features of the language or in terms of its purpose and use.

The protagonists of the most successful scripting languages are mostly rather pragmatic folks and do not care much about the answers at all. *Larry Wall*: “*a script is what you give the actors. A program is what you give the audience.*” When forced to answer, they mostly relate to language features and say

- scripting languages usually lack an explicit compilation step and perhaps even provide an interactive interpreter dialog mode.
- Scripting languages are a blend of those features that make a programmer productive and attempt to leave out everything else;
- in particular, they tend to have automatic memory management and powerful operations tightly built in (rather than relying on libraries) and

- they tend not to have strong typing rules and access rules that restrict the programmer from doing certain things.

The more philosophical approach to scripting languages proposes a dichotomy between scripting languages (also called systems integration languages or glue languages) on the one hand and systems programming languages on the other. Their difference is claimed first and foremost to be a difference of purpose and any differences in language characteristics to be mere consequences of that. The most prominent representative of this school of thought is John Ousterhout, the creator of Tcl. In [18] he describes

- scripting languages aren't intended for writing applications from scratch; they are intended primarily for plugging together components (which are usually written in systems programming languages).
- Scripting languages are often used to extend the features of components but they are rarely used for complex algorithms and data structures.
- In order to simplify the task of connecting components, scripting languages tend to be typeless.
- Applications for scripting languages are generally smaller than applications for system programming languages, and the performance of a scripting application tends to be dominated by the performance of the components.

As we will see in the language descriptions below, each of the scripting languages mixes and weighs these ideas rather differently.

However, if we view scripting languages as the potential next step on the ladder of ever higher-level languages described in the introduction, their most striking feature is certainly the by-and-large lack of static typing: Whereas previous languages introduced more and more elaborate mechanisms for statically declaring types, data structures, and access rules to be checked before the program is executed, scripting languages use hardly any type declarations and consequently do type checking only at run time; they also check fewer aspects.

1.2 Language comparisons: State of the practice

The scientific and engineering literature provides many comparisons of programming languages — in different ways and with different restrictions:

Some comparisons are purely theoretical discussions of certain language constructs. The many examples range from Dijkstra's famous letter "Go To statement considered harmful" [6] to comprehensive surveys of many languages [4, 19]. These are non-quantitative and usually partly speculative. Some such works are more or less pure opinion-pieces. There is also plenty of such discussion about scripting languages, though most of it does not claim to be scientific.

Some comparisons are benchmarks comparing a single implementation of a certain program in either language for expressiveness or resource consumption, etc.; an example is [11]. Such comparisons are useful, but extremely narrow and hence always slightly dubious: Is each of the implementations adequate? Or could it have been done much better in the given language? Furthermore, the programs compared in this manner are sometimes extremely small and simple.

Some are narrow controlled experiments, e.g. [7, 15], often focusing on either a single language construct, e.g. [14, p.227], or a whole notational style, e.g. [14, p.121], [22].

Some are empirical comparisons based on several and larger programs, e.g. [9]. They discuss for instance defect rates or productivity figures. The problem of these comparisons is lack of homogeneity: Each language is represented by different programs and it is unclear what fraction of the differences (or lack of differences) originates from the languages as such and what fraction is due to different programmer backgrounds, different software processes, different application domains, different design structures, etc.

1.3 The structure of this study

The present work provides both subjective/speculative and objective information comparing 4 popular scripting languages (namely Perl, Python, Rexx, and Tcl) to one another and to 3 popular representatives of the last few generations of high-level mainstream languages (namely C, C++, and Java).

It has the following features:

- The comparison grounds in objective information gathered in a large empirical study. The speculative discussion is guided and limited by the results obtained empirically.
- In the empirical study, the same program (i.e. an implementation of the same set of requirements) is considered for each language. Hence, the comparison is narrow but homogeneous.
- For each language, we analyze not a single implementation of the program but a number of separate implementations by different programmers. Such a group-wise comparison has two advantages. First, it smoothes out the differences between individual programmers (which threaten the validity of any comparison based on just one implementation per language). Second, it allows to assess and compare the *variability* of program properties induced by the different languages.
- Several different aspects are investigated, such as program length, amount of commenting, run time efficiency, memory consumption, and reliability.

The chapter has the following structure. It will initially introduce the individual scripting languages: Their most salient properties, their likely strengths and weaknesses. Then we will look at the programming task underlying the empirical data and what it can plausibly tell us

about other applications of the languages. The next section describes the origin of the program implementations and discusses the validity that we can expect from their comparison. Each subsequent section then compares one attribute of the programs across the language groups and relates the findings to the claims stated in the language description, explaining any contradictions where possible.

2 The languages

This chapter assumes the reader is at least roughly familiar with some or all of C, C++, and Java. Consequently, these will be described only briefly. However, each of the scripting languages is introduced in a little more detail, including a small program example that illustrates a few of the basic characteristics; see Figure 1 for the description.

```

obtain filename fn from first command line argument;
open the file named fn as filehandle f for reading;
for each line in f:
  read next line from f into variable s;
  remove all '-' characters from s;
  remove all '"' characters from s;
  print s to standard output;
end;
```

Figure 1: Example program in pseudocode: Print the contents of a file with all dashes and doublequotes removed. All three subtasks (reading file, printing to standard output, removing certain characters from many strings) also occur in the programming task used in the empirical study.

Most of the material in each description is taken or derived from various kinds of language documentation and from statements made by third parties (often including proponents of other scripting languages).

2.1 C, C++, Java

C is a small, relatively low-level high-level language that became immensely popular during the 1980s with the widespread adoption of Unix operating systems, which are traditionally written largely in C. The terse syntax of C has once looked rather strange, but has now become so mainstream that it appears almost canonical — C++, Java, Perl and many other languages borrow heavily from C's syntax. Since C does hardly impose any overhead on the programmer (in terms of memory and runtime), one can write extremely efficient programs in C. The lack of certain high-level language features plus the presence of easily misused features such as the preprocessor make the design soundness and maintainability of a C program highly dependent on the knowledge and discipline of the programmer.

C++, which became popular starting around 1990 and has undergone two major revisions and extensions since then, is more or less a superset of C, adding object-oriented language

features (classes and inheritance) plus many other extensions such as template types, operator overloading, an exception mechanism, and others. It retains most of the strengths and weaknesses of C, except that it is much larger and more powerful — which makes it perhaps even more reliant on well-trained programmers. Like C programs, well-written C++ programs can be very efficient and have good access to low-level features of the machine.

Java is also an object-oriented language that resembles C, but compared to C++ it is much smaller: It attempts to leave out many constructs that are difficult to control — but also lacks some that are quite useful. Java provides no access to low-level pointers, memory management is automatic (garbage collection) and the language enforces full type-safety (e.g. for safe execution in a “sandbox” on other people’s computers). The language Java is tightly bundled with a rather large standardized library that promotes many good design practices. Java is usually not compiled into machine code but rather into a machine-independent and rather high-level “bytecode”, which is then further compiled usually at load time (“just-in-time compiler”, JIT). Java’s characteristics support writing clean and well-designed programs, but make obtaining highly efficient code more difficult for both the programmer and the compiler than C and C++ do.

2.2 Perl

Version 1 of Perl came out in December 1987 and at that time, the name was meant to be an acronym meaning “Practical Extraction and Report Language”, where the term “practical” is explained as “*The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).*” This ambition includes lots of built-in support for reading text files, scanning and massaging strings, and formatting text output. However, Perl quickly grew way beyond that by incorporating direct access to many of Unix’s system calls and thus became what may be the most powerful language for system administration tasks on Unix systems. The language became highly popular with Version 4 released in March 1991. The current major version, 5, was first released October 1994 and “introduced everything else, including the ability to introduce everything else” [10]. In particular, it introduced object-oriented programming features.

Perl is a very powerful language (a quote from the Perl documentation: “*The three principal virtues of a programmer are Laziness, Impatience, and Hubris.*”), which has a number of interesting consequences. For instance, most things, even at the level of individual statements, can be expressed in more than one manner in Perl. This is also the semi-official Perl motto “*There’s more than one way to do it*”, also known as TMTOWTDI (often pronounced Tim Towdy). For instance, there are different constructs and styles that minimize either the time for writing a program, the time to run it, the amount of memory required by it, or the time to understand it. Following the motto, Perl has a rather large core language and a large library of built-in operations. Both are very well thought-out, but the extremely pragmatic approach of Perl’s design makes it rather odd in places (another quote from the documentation: “*Perl actually stands for “Pathologically Eclectic Rubbish Lister”, but don’t tell anyone I said that.*” In fact however, the potential ugliness of any aspect of Perl has carefully been balanced with the power that can be derived from it. Nevertheless, Perl makes it relatively easy to shoot

oneself in the foot or to produce programs that are very short but also very difficult to understand. For this reason, somewhat like for C++, writing good Perl programs depends more strongly on the knowledge and discipline of the programmer than in most other languages.

```
open(F, $ARGV[0]);
while ($s = <F>) { # read next line, terminating at eof
    $s =~ s/ - | "/g; # in s replace - or " by nothing, globally
    print $s;
}
```

Figure 2: Example program in Perl, normal style. <F> means reading the next line from a text file; s/pattern1/pattern2/ is a general regular-expression pattern replacement operator.

```
#!/bin/perl -p
s/ - | "/g;
```

Figure 3: Example program in Perl, minimalistic style. Execution option -p provides implicit iteration over all lines of each file provided as a command line argument plus implicit printing of the results.

See the small example program in Figure 2, which illustrates a few aspects of the terse syntax and powerful constructs found in Perl. The alternative version of the same program (shown in Figure 3) shows one of the many constructs for writing super-short programs and represents one example of TMTOWTDI.

The best starting points for tons of further information on Perl, including lists of books and articles, are <http://www.perl.org/> and <http://www.perl.com/>. A fairly comprehensive introduction is [24].

2.3 Python

Python (the name derives from the old british TV comedy show Monty Python's Flying Circus) first became publicly available in 1991. The self-description says "*Python is an interpreted, interactive, object-oriented programming language that combines remarkable power with very clear syntax.*"

This statement is in fact modest, because it is not only the syntax that is clear about Python but rather all of the language's concepts. Python is defined in such a way as to minimize the potential confusion of a programmer trying to learn it and also the strain on his or her memory. Most language constructs are nicely orthogonal so that no special rules are required for describing what happens if one combines them. Python resembles Java in that most of the many built-in features are not actually part of the language itself, but rather reside in many separate modules to be used (or ignored) explicitly and individually, which makes incremental learning of the language somewhat easier.

As a result of this design, Python programs cannot be quite as short as minimalistic Perl programs, but on the average Python programs are easier to read.

```
import sys, fileinput, re
for s in fileinput.input(sys.argv[1]): # iterate line-by-line
    s = re.sub("-|\\\"", "", s) # replace - or " by nothing in s
    print s, # comma avoids additional newline
```

Figure 4: Example program in Python. Only indentation is required to indicate block structure.

The small example program in Figure 4 illustrates some of these points: Although it is effectively only three statements long, it accesses three different modules from the standard library. The module approach allows, for instance, to choose between three different kinds of regular expressions (from the modules `regexp`, `regex`, and `re`).

The best starting point for tons of further information on Python, including lists of books and articles, is <http://www.python.org/>. A fairly comprehensive introduction is [16].

2.4 Rexx

All other scripting languages discussed here either started in an academic setting (Tcl) or were initially the work of a single individual (Perl, Python). In contrast, Rexx is a creation of IBM. It was historically available on various mainframe operating systems and consists of a system-independent core language plus system-specific extensions. It has now also been ported to other systems, in particular Windows and Unix systems (AIX, Linux) and at least one open source implementation is available (Regina).

Rexx is probably the least interesting of the languages discussed here in several respects: It is not as widely available on mainstream platforms as are the other languages; its user community is neither as large nor as dynamic; it has neither the conceptual and syntactical cleanliness of Python or Tcl, nor the extreme power and versatility of Perl. Here is the (somewhat dull) description of Rexx given on its homepage <http://www.ibm.com/software/ad/rexx/>: “IBM REXX is a powerful procedural programming language that lets you easily write programs and algorithms that are clear and structured. It includes features that make it easy to manipulate the most common symbolic objects, such as words, numbers, and names.”

Due to the smaller user community, there are also only 4 Rexx programs in our empirical study which makes for too small a dataset for properly judging Rexx’s behavior. Therefore we will limit the discussion of Rexx to a few words here and there.

2.5 Tcl

Tcl (Tool Command Language, pronounced “tickle”) was invented by John Ousterhout, then at the University of California in Berkeley, and was first released in 1989. Since then it has had quite a lively career leading to its current major version 8: Ousterhout (and Tcl) left Berkeley and went to Sun in 1994, the Tcl Consortium took over Tcl in 1997, but was abandoned in 1999. Meanwhile in 1998, Ousterhout had founded Scriptics, which developed the commercial TclPro tools, but still kept the basic Tcl language implementation free and

open source. In 2000, Scriptics renamed itself to Ajuba Solutions and later merged with Interwoven, which made TclPro open source software, too. Then ActiveState took over its development and made it commercial again, but basic Tcl is still free (also being maintained by ActiveState now as of early 2002).

Tcl has by far the smallest core language of all contestants; it basically consists of a general comment, command, and block syntax and evaluation semantics but has no actual language primitives whatsoever. Even constructs as fundamental as if-then-else or variable assignment are not part of the core language but rather take the form of built-in library operations called command words. See figure 5 for the Tcl version of our small example program.

```
set f [open [lindex $argv 0]]
while {1} {
    gets $f s
    if {[eof $f]} break
    regsub -all -- {[-\"]} $s {} s; # replace all - and " in s by
    puts $s                        # nothing, giving s
}
```

Figure 5: Example program in Tcl. Section enclosed in square brackets denote command substitution and are replaced at run time by the result of executing what is inside. Braces are strong quotes, forming one long word as-is to be evaluated later.

The Tcl core is so small that its syntax and semantics are quite precisely described by only 11 paragraphs of text. The Tcl distribution is complemented by a large built-in library of such command words covering most of the large base functionality that one might expect from a scripting language. This approach is both curse and glory at the same time.

On the positive side, since almost everything in Tcl is an extension, Tcl makes for a good meta-language, allowing extensions of essentially every kind and in every direction. Consequently, a number of special-purpose languages built on top of Tcl exist, e.g. tclX for Unix system calls and many other extensions, Expect for adding automation to interactive user interfaces, Incr Tcl for object-oriented programming, etc.

Furthermore, since almost everything in Tcl is an extension anyway, Tcl smoothly allows for two-way embedding with other programs: Say you have written a large program system in C++, then you can embed it with Tcl, provide simple adapter wrappers for all relevant functions, and then call all these functions from Tcl scripts as if they were built into Tcl. It also works the other way round: Your system can call Tcl scripts to simplify some of its own tasks. In particular, this is an easy way for adding a clean, powerful, standard-based interactive command language to a system and is why Tcl calls itself the Tool Command Language.

On the negative side, many program constructs are necessarily awkward and inconvenient. For instance, even integer arithmetic and infix notation are available only through a command word (called “expr”) so that one has to write e.g. [expr \$i+1] rather than just \$i+1.

Furthermore, Tcl is inefficient by design. In Tcl, everything is a String — and that includes the program text itself, even while it is running. Consequently, a Tcl interpreter has a hard

time avoiding to do certain things over and over in a loop that yield the same results over and over. For instance, it may convert the fixed string “1234” into the integer number 1234 each time this value is passed to a function as an argument.

The best starting point for lots of further information on Tcl, including lists of books and articles, is <http://www.scriptics.com/> (now also known as <http://tcl.activestate.com>).

3 Overview of the empirical study

This section describes the programming task solved by the participants of the empirical study and the number and origin of the programs (which come from two fairly different sources). It then discusses the validity of the study: To what degree must we expect that differences in programmer ability and other factors have distorted the results? Finally, it explains the statistical evaluation approach and format of the graphs used in the subsequent results parts of this chapter.

3.1 Short overview

The programming task was a program called *phonocode* that maps telephone numbers into strings of words according to a German dictionary supplied as an input file and a fixed digit-to-character encoding. It will be described in Section 3.2.

The programs analyzed in this report come from two different sources. The Java, C, and C++ programs were produced by paid student subjects in the course of a controlled experiment, the others were produced by a broad variety of subjects under unobserved conditions after a public call for volunteers sent to Usenet newsgroups and were submitted by Email. Find some more details in sections 3.3 and 3.4. Section 3.5 discusses the number of programs from each language that were collected for the study

In a study of this kind, there are always concerns with respect to the validity of the comparison. In our case, we find that the conditions of the study probably put the script language groups at some advantage compared to the non-script groups with respect to the following criteria: possibly slightly higher average programmer capability, more flexible work conditions, better up-front information about evaluation criteria and about specific task difficulties, a better program testbed. Fortunately, all of these differences can be expected to be only modest, though. They should make us ignore small differences between the groups’ performance when discussing the results and focus on major differences only. Section 3.6 provides a thorough discussion of these issues.

3.2 The programming task ‘phonocode’

The programs analyzed in this study represent solutions of the following task.

We want to encode telephone numbers as words to make them easier to remember. Given the following mapping from letters to digits

E	J N Q	R W X	D S Y	F T	A M	C I V	B K U	L O P	G H Z
e	j n q	r w x	d s y	f t	a m	c i v	b k u	l o p	g h z
0	1	2	3	4	5	6	7	8	9

and given a dictionary like this

```
Bo"
da
Doubletten-Konkatenations-Form
je
mir
Mix
Mixer
Name
neu
o"d
so
su"ss
superkalifragilistisch
Tor
```

and given a list of "telephone numbers" like this

```
5624-82
0721/608-4067
10/783--5
/378027586-4259686346369
1078-913-5
381482
```

write a program that encodes the phone numbers into all possible sequences of words according to the mapping like this

```
5624-82: mir Tor
5624-82: Mix Tor
10/783--5: je Bo" da
10/783--5: neu o"d 5
/378027586-4259686346369: superkalifragilistisch
381482: so 1 Tor
```

Building the encoding sequentially left-to-right, a single digit from the original number may be placed at a point where no matching word is available.

The character mapping is fixed and can be coded right into the program.

The dictionary is supplied as a text file and should be read into memory (maximum size 75000 words). Words may contain dashes and double quotes (signifying umlaut characters) which are to be ignored for the encoding. A word may be up to 50 characters long.

The phone numbers are supplied as a text file of unlimited size and must be read and processed line by line. Phone numbers may contain dashes and slashes which are to be ignored for the encoding. A phone number may be up to 50 characters long.

The output encodings must be printed exactly as shown (except for their order within each number's block). Nothing at all is printed for numbers that cannot be encoded.

For details on the task and how it was communicated to the participants, see Appendix A.

3.3 Origin of the C, C++, and Java programs

All C, C++, and Java programs were produced in 1997/1998 during a controlled experiment that compared the behavior of programmers with and without previous PSP (Personal Software Process [12]) training. All of the subjects were Computer Science master students. They chose their programming language freely. A few participants of the experiment used languages other than C, C++, or Java, such as Modula-2 and Sather-K. The subjects were told that their main goal should be producing a correct (defect-free) program. A high degree of correctness was ensured by an acceptance test. The sample of programs used here comprises only those that passed the acceptance test. Several subjects decided to give up after zero, one, or several failed attempts at passing this test.

Detailed information about the subjects, the experimental procedure, etc. can be found in [21].

For brevity, I will often call these three languages *non-script languages* and the corresponding programs *non-scripts*.

3.4 Origin of the Perl, Python, Rexx, and Tcl programs

The Perl, Python, Rexx, and Tcl programs were all submitted in late 1999 by volunteers after I had posted a "Call for Programs" on several Usenet newsgroups (comp.lang.perl.misc, de.comp.lang.perl.misc, comp.lang.rexx, comp.lang.tcl, comp.lang.tcl.announce, comp.lang.python, comp.lang.python.announce) and one mailing list (called "Fun with Perl", fwp@technofile.org).

For four weeks after that call, the requirements description and test data were posted on a website for viewing and download. The participants were told to develop the program, test it, and submit it by email. There was no registration and I have no way of knowing how many participants started to write the program but gave up.

Detailed information about the submission procedure can be found in Section 3.2.

Table 1: For each non-script programming language: Number of programs originally prepared (progs), number of subjects that voluntarily participated a second time one year later (second), number of programs that did not pass the acceptance test (unusable), and final number of programs used in the study (total). For each script programming language: Number of programs submitted (progs), number of programs that are resubmissions (second), number of programs that could not be run at all (unusable), and final number of programs used in the study (total).

language	progs	second	unusable	total
C	8	0	3	5
C++	14	0	3	11
Java	26	2	2	24
Perl	14	2	1	13
Python	13	1	0	13
Rexx	5	1	1	4
Tcl	11	0	1	10
Total	91	6	11	80

3.5 Number of programs

As shown in Table 1, the set of programs analyzed in this study contains between 4 and 24 programs per language, 80 programs overall. Note that the sample covers 80 different programs but only 74 different authors, as a six of them submitted two versions – two of these in a different language (namely Java) than the first.

As many as 91 programs were actually collected, but 11 of them had to be discarded for the evaluation because they did not work properly.

Furthermore, the results for C and Rexx will be based on only 5 or 4 programs, respectively, and are thus rather coarse estimates of reality. Given the large amount of variation between individual programs even within the same language, statistically significant results can rarely be obtained with such a small number of data points. We will hence often disregard these languages in the discussion as they would contribute more confusion than information.

For all of the other languages there are 10 or more programs, which is a broad-enough base for reasonably precise results.

3.6 Validity: Are these programs comparable?

The rather different conditions under which these programs were produced raise an important question: Is it sensible and fair to compare these programs to one another or would such a comparison say more about the circumstances than it would say about the programs? Put differently: Is our comparison valid? (More precisely: internally valid) The following subsections discuss problems that threaten the validity. The most important threats usually occur between the language groups script and non-script; a few caveats when comparing one particular script language to another or one non-script language to another also exist and will be discussed where necessary.

3.6.1 Programmer capabilities

The average capabilities of the programmers may differ from one language to the other.

It is plausible that the Call for Programs has attracted only fairly competent programmers and hence the script programs may reflect higher average programmer capabilities than the non-script programs. However, two observations make me estimate this difference to be small. First, with some exceptions, the students who created the non-script programs were also quite capable and experienced with a median programming experience of 8 years (see [21]). Second, some of the script programmers have described themselves as follows:

“Most of the time was spent learning the language not solving the problem.”

“Things I learned: [...] Use a language you really know.”

“First real application in python.”

“It was only my 4th or 5th Python script.”

“I’m not a programmer but a system administrator.”

“I’m a social scientist.”

“I am a VLSI designer (not a programmer) and my algorithms/coding-style may reflect this.”

“This is my first Tcl prog. I’m average intelligence, but tend to work hard.”

“Insight: Think before you code. [...] A lot of time was lost on testing and optimising the bad approach.”

Taken together, I expect that the script and non-script programmer populations are roughly comparable — at least if we ignore the worst few from the non-script group, because their would-be counterparts in the script group have probably given up and not submitted a program at all. We should keep this in mind for the interpretation of the results below.

Within the language groups, some modest differences between languages also occurred: In the non-script group, the Java programmers tend to be less experienced than the C and C++ programmers for two reasons. First, most of the noticeably most capable subjects chose C or C++, and second, nobody could have many years of Java experience at the time, because the experiment was conducted in 1997 and 1998, when Java was only about three years old.

Within the script group, my personal impression is that the Perl subjects tended to be more capable than the others. The reasons may be that the Perl language appears to irradiate a strange attraction to highly capable programming fans and that the “fun with Perl” mailing list on which I posted the call for programs appears to reach a particularly high fraction of such persons.

3.6.2 Work time reporting accuracy

Even if the capabilities of the programmers are comparable, the work times reported by the script programmers may be inaccurate.

In contrast to the non-script programs from the controlled experiment, for which we know the real programming time accurately, nothing kept the script programmers from heavily “rounding down” the working times they reported when they submitted their program. Some of them also reported they had had to estimate their time, as either they did not keep track of it during the actual programming work or they were mixing too much with other tasks (“*many breaks to change diapers, watch the X-files, etc.*”). In particular, some apparently read the requirements days before they actually started implementing the solution as is illustrated by the following quotes:

“Design: In my subconscious for a few days”

“The total time does not include the two weeks between reading the requirements and starting to design/code/test, during which my subconscious may have already worked on the solution”

“The actual time spent pondering the design is a bit indeterminate, as I was often doing other things (eating cheese on toast, peering through the snow, etc).”

Nevertheless, there is evidence (described in Section 8) that at least on the average the work times reported are reasonably accurate for the script group, too: The old rule of thumb, saying the number of lines written per hour is independent of the language, holds fairly well across all languages.

3.6.3 Different task and different work conditions

The requirements statement, materials provided, work conditions, and submission procedure were different for the script versus non-script group.

The requirements statement given to both the non-script and the script programmers said that correctness was the most important aspect for their task and also that the algorithms must be designed such that they always consider only a fraction of the words from the dictionary when trying to encode the next digit. However, the announcement posted for the script programmers (although not the requirements description) also made a broader assignment, mentioning programming effort, program length, program readability/modularization/maintainability, elegance of the solution, memory consumption, and run time consumption as criteria on which the programs might be judged.

This focus difference may have directed somewhat more energy towards producing an efficient program in the script group compared to the non-script group. On the other hand, two things will have dampened this difference. First, the script group participants were explicitly told “*Please do not over-optimize your program. Deliver your first reasonable solution*”. Second, in the non-script group highly inefficient programs were filtered out and sent back for optimization in the acceptance test, because the test imposed both a time and memory limit¹ not present in the submission procedure of the script group.

¹64 MB total, 30 seconds maximum per output plus 5 minutes for loading on a 143 MHz Sparc Ultra I.

There was another difference regarding the acceptance test and reliability measurement procedures: Both groups were given a small dictionary (`test.w`, 23 words) and a small file of inputs (`test.t`) and correct outputs (`test.out`) for program development and initial testing, plus a large dictionary (`woerter2`, 73113 words). The acceptance test for the non-script group was then performed using a randomly created input file (different each time) and a medium-large dictionary of 20946 words. A failed acceptance test cost a deduction of 10 Deutschmarks from the overall compensation paid for successful participation in the experiment, which was 50 Deutschmarks (about 30 US Dollars).

In contrast, the script group was given both the input file `z1000.in` and the corresponding correct outputs `z1000.out` that are used for reliability measurement in this report and could perform as many tests on these data as they pleased.

Possessing these data is arguably an advantage for the script group with respect to the work time required. Note, however, that the acceptance test in the non-script group automatically flagged and reported any mistakes separately while the script group had to perform the comparison of correct output and actual output themselves. The web page mentioned that the Unix utilities `sort` and `diff` could be used for automating this comparison.

A more serious problem is probably the different working regime: As mentioned above, many of the script group participants thought about the solution for several days before actually producing it, whereas the non-script participants all started to work on the solution right after reading the requirements. This is probably an advantage for the script group. On the other hand, for more than two thirds of the non-script group one or several longer work breaks (for the night or even for several days) occurred as well.

Summing up we might say that the tasks of the two groups are reasonably similar, but any specific comparison must clearly be taken with a grain of salt. There was probably some advantage for the script group with respect to work conditions: some of them used unmeasured thinking time before the actual implementation work. Hence, only severe results differences should be relied upon.

3.6.4 Handling a misunderstood requirement

There was one important statement in the requirements that about one third of all programmers in both groups misunderstood at first (see Section A.3), resulting in an incorrect program. Since many these programmers were not able to resolve the problem themselves, help was required. This help was provided to the non-script programmers as follows: When they failed an acceptance test due to this problem, the respective sentence in the requirements was pointed out to them with the advice of reading it extremely carefully. If they still did not find the problem and approached the experimenter for further help, the misunderstanding was explained to them. All of these programmers were then able to resolve the problem. Actually correcting this mistake in a faulty program was usually trivial.

For the script programmers, no such interaction was possible, hence the requirements description posted on the web contained a pointer to a “hint”, with the direction to first re-read the

requirements carefully and open the hint only if the problem could not be resolved otherwise. The exact wording and organization is shown in Appendix A below.

The easier access to the hint may have produced an advantage (with respect to work time) for the script-group, but it is hard to say whether or to which extent this has happened. On the other hand, a few members of the script group have reported having had a hard time understanding the actual formulation of the hint. My personal impression based on my observations of the non-script group and on the feedback I have received from participants of the script group is that the typical work time penalty for misunderstanding this requirement was similar in the script and non-script group.

3.6.5 Other issues

The non-script programmers had a further slight disadvantage, because they were forced to work on a computer that was not their own. However, they did not complain that this was a major problem for them. The script programmers used their own machine and programming environment.

The Rexx programs may experience a small distortion because the platform on which they were evaluated (a Rexx implementation called “Regina”) is not the platform on which they were originally developed. Similarly, the Java programs were evaluated using a much newer version of the JDK (Java Development Kit) than the one they were originally developed with. These context changes are probably not of major importance, though.

3.6.6 Summary

Overall, it is probably fair to say that

- due to the design of the data collection, the data for the script groups will reflect several relevant (although modest) a-priori advantages compared to the data for the non-script groups and
- there are likely to be some modest differences in the average programmer capability between any two of the languages.

Due to these threats to validity, we should discount small differences between any of the languages, as these might be based on weaknesses of the data. Large differences, however, are likely to be valid.

3.7 Evaluation conditions

The programs were evaluated using the same dictionary `woerter2` as given to the participants. Three different input files were used: `z1000` contains 1000 non-empty random phone numbers, `m1000` contains 1000 arbitrary random phone numbers (with empty ones allowed), and `z0` contains no phone number at all (for measuring dictionary load time alone).

Table 2: Compilers and interpreters used for the various languages. Note on Java platform: The Java evaluation uses the JDK 1.2.2 Hotspot Reference version (that is, a not performance-tuned version). However, to avoid unfair disadvantages compared to the other languages, the Java run time measurements will reflect two modifications where appropriate: First, the JDK 1.2.1 Solaris Production version (with JIT) may be used, because for short-running programs the tuned JIT is faster than the untuned Hotspot compiler. Second, some programs are measured based on a special version of the `java.util.Vector` dynamic array class not enforcing synchronization. This is similar to `java.util.ArrayList` in JDK 1.2, but no such thing was available in JDK 1.1 with which those programs were written.

language	compiler or execution platform
C	GNU gcc 2.7.2
C++	GNU g++ 2.7.2
Java	Sun JDK 1.2.1/1.2.2
Perl	perl 5.005_02
Python	python 1.5.2
Rexx	Regina 0.08g
Tcl	tcl 8.2.2

Extremely slow programs were stopped after a timeout of 2 minutes per output plus 20 minutes for loading the dictionary — however, three quarters of all programs finished the whole `z1000` run with 262 outputs in less than 2 minutes.

All programs were executed on a 300 MHz Sun Ultra-II workstation with 256 MB memory, running under SunOS 5.7 (Solaris 7); the compilers and interpreters are listed in Table 2

3.8 External validity: To what degree can these results be generalized?

Even if the results discussed below are nicely valid as they stand, there is still the question whether they are also relevant: Will similar differences between the languages be observed in many practical situations? When and where can the results be generalized to other circumstances, that is, how much external validity do they exhibit?

3.8.1 Stability of results over time

The results regarding execution time and memory consumption depend not only on the language and the capabilities of the programmer, but also on the quality of the compiler or interpreter and the runtime system used.

In this respect it should be noted that while the translation and execution technology for C and C++ is rather mature, a lot of progress is still possible for the other languages — most strongly probably for Java and Tcl, whose design requires rather complex optimization technology for producing optimal efficiency. And indeed the language communities of all of these languages are working busily on improving the implementations — most strongly certainly for Java, but also quite actively for Python and Perl, perhaps a little less for Tcl and Rexx.

Therefore, in those cases where Java or the script languages impose an execution time or memory penalty compared to C or C++, this penalty can be expected to shrink over time.

3.8.2 Generalizability of results to other programming tasks

Little can be inferred directly from the results of the study that reliably applies to very different programming tasks, e.g. requiring much larger programs, involving a lot of numerical computations, handling much larger amounts of data or much coarser-grained data, having an interactive user interface, etc.

However, the class of programs represented by the phonocode task is rather large and common, especially in an area for which script languages are often used, namely transcribing text files from one format into another, perhaps including some semantic processing underways. If these transcriptions are such that the rules are not fixed in the program, but rather also rely on some kind of mapping instructions or vocabulary described by a run-time data set, then such programs often resemble the phonocode task fairly well. A frequent example of such programs could be the conversion from one XML-based specification language into another that has similar expressive power but different constructs.

At least for this class of programs, we can expect to find results similar to this study when comparing these languages.

3.9 Evaluation tools: Data plots and statistical methods

The plots and statistical methods used in the evaluation are described in some detail in [21]; we only give a short description here.

The main evaluation tool will be the multiple boxplot display, see for example Figure 7 on page 23. Each of the “lines” represents one subset of data, as named on the left. Each small circle stands for one individual data value. The rest of the plot provides visual aids for the comparison of two or more such subsets of data. The shaded box indicates the range of the middle half of the data, that is, from the first quartile (25% quantile) to the third quartile (75% quantile). The “whiskers” to the left and right of the box indicate the bottom and top 10% of the data, respectively. The fat dot within the box is the median (50% quantile). The “M” and the dashed line around it indicate the arithmetic mean and plus/minus one standard error of the mean.

Many interesting observations can easily be made directly in these plots. For quantifying some of them, I will also sometimes provide the results of statistical significance tests: Medians are compared using the Wilcoxon Rank Sum Test (Mann-Whitney U-Test) and in a few cases means will be compared using the t-Test. All tests are performed one-sided and all test results will be reported as p -values, that is, the probability that the observed differences between the samples are only due to random variations and either no difference between the underlying populations does indeed exist or there is even a difference in the opposite direction.

At some points I will also provide confidence intervals, either on the differences in means or on the differences in logarithms of means (that is, on the ratios of means). These confidence intervals are computed by Bootstrapping. They will be chosen such that they are open-ended, that is, their upper end is at infinity. Bootstrapping is described in more detail in [8].

Note that due to the caveats described in Section 3.6 all of these quantitative statistical inference results can merely indicate trends; they should not be considered precise evidence.

For explicitly describing the variability within one group of values we will use the *bad/good ratio*: Imagine the data be split in an upper and a lower half, then the bad/good ratio is the median of the upper half divided by the median of the lower half. In the boxplot, this is just the value at the right edge of the box divided by the value at the left edge. In contrast to a variability measure such as the standard deviation, the bad/good ratio is robust against the few extremely high values that occur in our data set.

4 Results for execution time

4.1 Total run time: z1000 data set

4.1.1 Results

The global overview of the program run times on the z1000 input file is shown in Figure 6. Except for C++, Java and Rexx, at least three quarters of the programs run in less than one minute. The most important lesson to learn from this plot is that in all languages there are a few programs that are not just slower than the others but rather are immensely much slower. Put differently: With respect to execution time, an incompetent programmer can shoot him or herself in the foot quite badly in any language.

In order to see and discriminate all of the data points at once, we can use a logarithmic plot as shown in Figure 7. We can make several interesting observations:

- The typical (i.e., median) run time for Tcl is not significantly longer than that for Java (one-sided Wilcoxon test $p = 0.21$) or even for C++ ($p = 0.30$).
- Don't be confused by the median for C++. Since the distance to the next larger and smaller points is rather large, it is unstable. The Wilcoxon test, which takes the whole sample into account, confirms that the C++ median in fact tends to be smaller than the Java median ($p = 0.18$).
- The median run times of Python are smaller than those of Rexx ($p = 0.024$), and Tcl ($p = 0.047$). The median run times of Perl are also smaller than those of Rexx ($p = 0.018$), and Tcl ($p = 0.002$).
- Except for two very slow programs, Tcl and Perl run times tend to have a smaller variability than the run times for the other languages. For example, a one-sided bootstrap

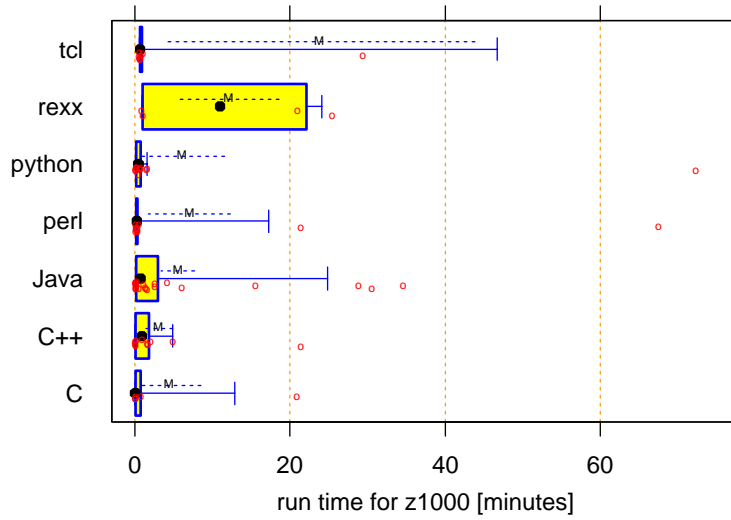


Figure 6: Program run time on the z1000 data set. Three programs were timed out with no output after about 21 minutes. One Tcl program took 202 minutes. The bad/good ratios range from 1.5 for Tcl up to 27 for C++.

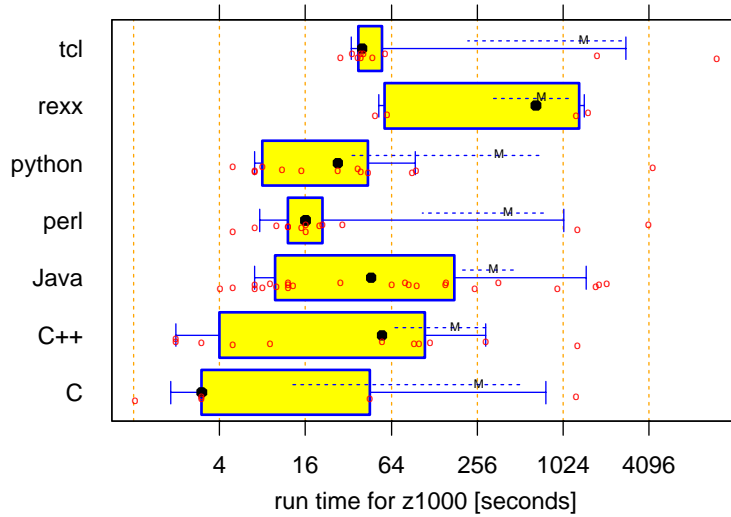


Figure 7: Program run time on the z1000 data set. Equivalent to Figure 6, except that the axis is logarithmic and indicates seconds instead of minutes.

test for differences in interquartile range of logarithmic run times (i.e. differences in box width in Figure 7) between Perl and Python indicates $p = 0.15$.

Remember to interpret the plots for C and Rexx with care, because they have only few points.²

If we aggregate the languages into only three groups, as shown in Figure 8, we find that the run time advantage of C/C++ is not statistically significant: Compared to Scripts, the C/C++ advantage is accidental with probability $p = 0.15$ for the median and with $p = 0.11$ for the log mean (via t-test). Compared to Java, the C/C++ advantage is accidental with $p = 0.074$ for the median and $p = 0.12$ for the log mean.

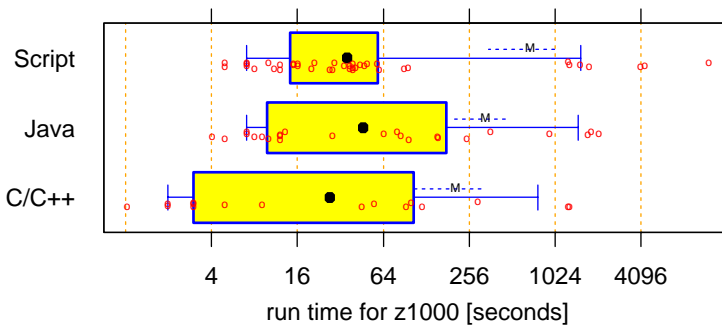


Figure 8: Program run time on the z1000 data set on logarithmic axis; just like Figure 7, except that the samples for several languages are aggregated into larger groups. The bad/good ratios are 4.1 for script, 18 for Java and 35 for C/C++.

The larger samples of this aggregate grouping allow for computing reasonable confidence intervals for the differences. A bootstrap-estimated confidence interval for the log run time means difference (that is, the run time ratio) indicates that with 80% confidence a script will run at least 1.29 times as long as a C/C++ program (but with higher log run time variability, $p = 0.11$). A Java program must be expected to run at least 1.22 times as long as a C/C++ program. There is no significant difference between average Java and Script run times.

4.1.2 Discussion

- The run time penalty of a script relative to a C or C++ program is a lot smaller for the phonocode problem than probably most of us would have expected. There are two reasons for this. First, scripts can generally be quite efficient if their most sensible implementation can make heavy use of powerful builtin operations, because these

²Regarding the performance of Rexx, participant Ian Collier pointed out that the otherwise high performance Regina interpreter suffers from its fixed hashtable size for the phonocode problem, because the default size of 256 is too small. Increasing this value to 8192 (which requires recompiling Regina) reduced the run time of Collier's Rexx program from 53 seconds down to 12. Further increasing the value would probably reduce the run time still more, because a hash table for 70000 elements should best have more than 70000 entries.

operations are written in lower-level languages and have usually been thoroughly optimized. Second, as we can see in the plots, each of the non-script program groups can be distinguished quite clearly into an efficient and an inefficient subgroup. No such effect is present for Perl, Python, or Tcl. As we will see in Section 9, the reason for this lies in the program design: While the scripts more or less all use the same basic algorithm design, the non-script programs use either of two designs that are radically different in terms of execution time.

- The slower execution of Tcl compared to Perl and Python corroborates the speculative statement from the language introduction that the design of Tcl makes it hard to obtain efficient implementations.
- The smaller variance in execution time (as described by the bad/good ratio) for the script languages in general and for Perl and Python in particular can be considered an advantage of the script languages: They appear to be less dependent on the programmer for obtaining reasonable performance than are the non-script languages. This may be quite important in practice. The point here is one of risk: Yes, a well-optimized C or C++ program is likely to be faster than any script, but is there a high-enough probability that you really get one?
- Given Perl's TMTOWTDI principle, the smaller bad/good ratio of Perl compared to Python is a surprise. Thinking of a good reason why the bad/good ratio of Tcl should be so much smaller than that of Python is also difficult. Perhaps these both facts are indications that the Perl and Tcl programmers were somewhat better trained in their language than the Perl programmers.

Summing up, the script programs are not only much faster than expected, compared to the non-script programs, they also exhibit better predictability of the execution time where different programmers are concerned.

4.2 Run time for initialization phase only: z0 data set

4.2.1 Results

We can repeat the same analysis for the case where the program only reads, and stores the dictionary; almost all programs also reverse-encode the words as telephone numbers in this phase to accelerate further execution. Figure 9 shows the corresponding run time.

We find that C and C++ are clearly faster in this situation than all other programs. The fastest script languages are again Perl and Python. Rexx and Tcl are again slower than these but now Java is faster.

For the aggregate grouping (Figure 10) we find that, compared to a C/C++ program, a Java program will run at least 1.3 times as long and a script will run at least 5.5 times as long (at the 80% confidence level). Compared to a Java program, a script will run at least 3.2 times as long.

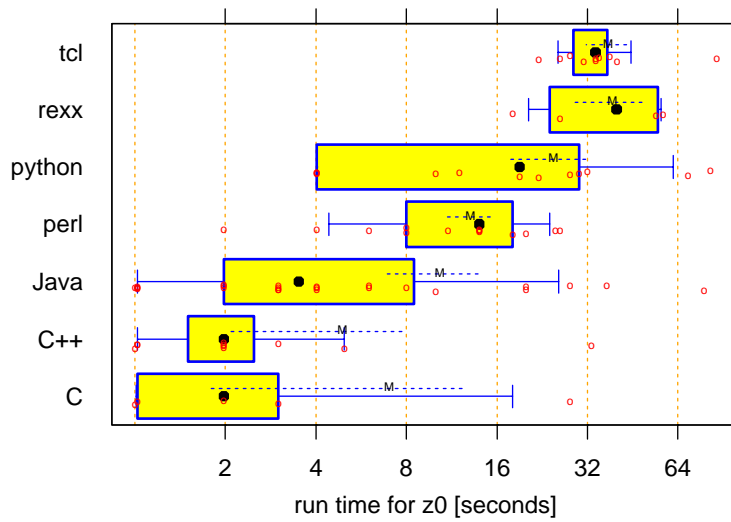


Figure 9: Program run time for loading and preprocessing the dictionary only (z0 data set). Note the logarithmic axis. The bad/good ratios range from 1.3 for Tcl up to 7.5 for Python.

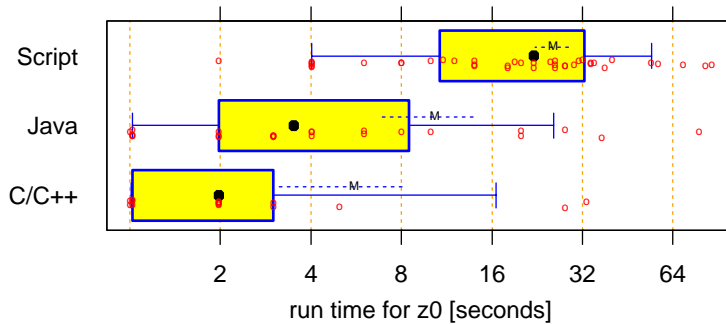


Figure 10: Program run time for loading and preprocessing the dictionary only (z0 data set); just like Figure 7, except that the samples for several languages are aggregated into larger groups. The bad/good ratio is about 3 for Script and C/C++ and 4.3 for Java.

4.2.2 Discussion

These results are obviously somewhat more of the kind that one might have expected: All scripting languages are slower than Java and Java is slower than both C and C++.

No explanation, however, will be given here, as there are three very different components in the run time for this phase and I performed no analysis trying to decompose them:

- Reading the dictionary file.
- Converting each word into the corresponding bare telephone number. (As we will see in Section 9, the eventually slower non-script programs were the fastest here, as they converted the first letter of each word only.)
- Storing the words in a data structure that allows to retrieve them quickly based on the corresponding phone number.

4.3 Run time for search phase only

4.3.1 Results

Finally, we may subtract this run time for the loading phase (z0 data set) from the total run time (z1000 data set) and thus obtain the run time for the actual search phase only. Note that these are time differences from two separate program runs. Due to the measurement granularity, a few zero times result. These were rounded up to one second. Figure 11 shows the corresponding run times.

We find that very fast programs occur in all languages except for Rexx and Tcl and very slow programs occur in all languages without exception. More specifically:

- The median run time for Tcl is longer than that for Python ($p = 0.10$), Perl ($p = 0.012$), and C ($p = 0.099$), but shorter than that of Rexx ($p = 0.052$).
- The median run times of Python are smaller than those of Rexx ($p = 0.007$), and Tcl ($p = 0.10$). They even tend to be smaller than those of Java ($p = 0.13$).
- The median run times of Perl are smaller than those of Rexx ($p = 0.018$), Tcl ($p = 0.012$), and even Java ($p = 0.043$).
- Although it doesn't look like that, the median of C++ is not significantly different from any of the others (two-sided tests yield $0.26 < p < 0.92$).

The aggregated comparison in Figure 12 indicates no significant differences between any of the groups, neither for the pairs of medians ($p > 0.14$) nor for the pairs of means ($p > 0.20$).

However, a bootstrap test for differences of the box widths indicates that with 80% confidence the run time variability of the Scripts is smaller than that of Java by a factor of at least 2.1 and smaller than that of C/C++ by a factor of at least 3.4.

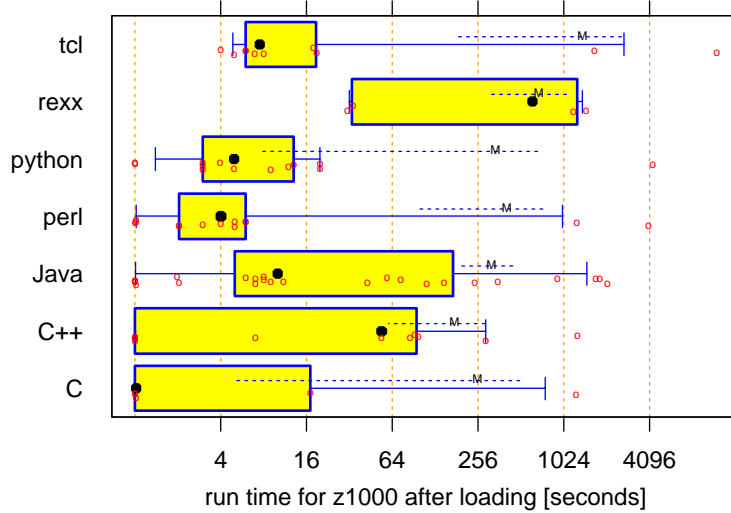


Figure 11: Program run time for the search phase only. Computed as time for z1000 data set minus time for z0 data set. Note the logarithmic axis. The bad/good ratios range from 2.9 for Perl up to over 50 for C++ (in fact 95, but unreliable due to the imprecise lower bound).

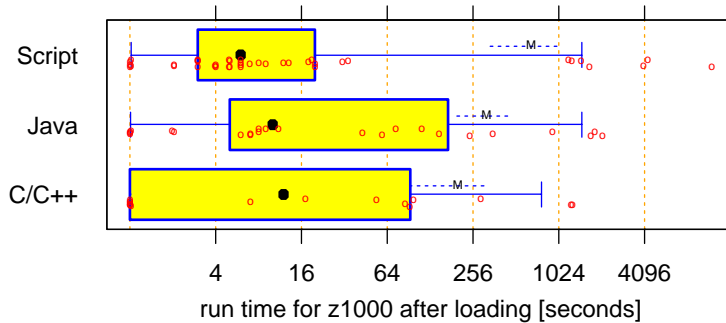


Figure 12: Program run time for the search phase only. Computed as time for z1000 data set minus time for z0 data set. This is just like Figure 11, except that the samples for several languages are aggregated into larger groups. The bad/good ratio is about 7 for Script, 34 for Java, and over 50 for C/C++ (in fact 95, but unreliable due to the estimated lower bound).

4.3.2 Discussion

Some people may be quite surprised at this, but given the right kind of work to do, the average script may be just as fast as the average non-script. What may even be more important in practice is the fact that, just as for the overall run time, the variability of script run times is so much smaller.

5 Results for memory consumption

How much memory is required by the programs?

5.1 Results

Figure 13 shows the total process size at the end of the program execution for the z1000 input file.

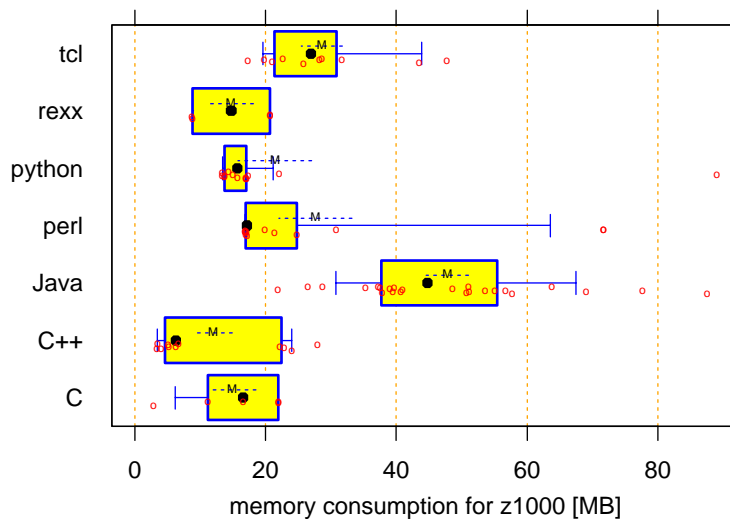


Figure 13: Amount of memory required by the program, including the interpreter or run time system, the program itself, and all static and dynamic data structures. The bad/good ratios range from 1.2 for Python up to 4.9 for C++.

Several observations are interesting:

- The most memory-efficient programs are clearly the smaller ones from the C and C++ groups.

- The least memory-efficient programs are clearly the Java programs.
- Except for Tcl, only few of the scripts consume more memory than the worse half of the C and C++ programs.
- Tcl scripts typically require clearly more memory than other scripts.
- Python scripts typically require less memory than both Perl and Tcl scripts.
- Especially for Python, but also for Perl and Tcl, the relative variability in memory consumption tends to be much smaller than for C and in particular C++.
- A few (but only a few) of the scripts have a horribly high memory consumption compared to the others.
- On the average (see Figure 14) and with a confidence of 80%, the Java programs consume at least 32 MB (or 297%) more memory than the C/C++ programs and at least 20 MB (or 98%) more memory than the script programs. The script programs consume only at least 9 MB (or 85%) more than the C/C++ programs.

Summing up, the memory consumption of Java is typically more than twice as high as that of scripts, and scripts are not necessarily worse than a program written in C or C++, although they do beat the most parsimonious C or C++ programs.

5.2 Discussion

We can conclude

- Neither scripts nor Java programs can beat a memory-efficient C or C++ program. This is partially due to the much lower level (and hence more controlled) memory management in these languages and partially due to the simpler data structure for one

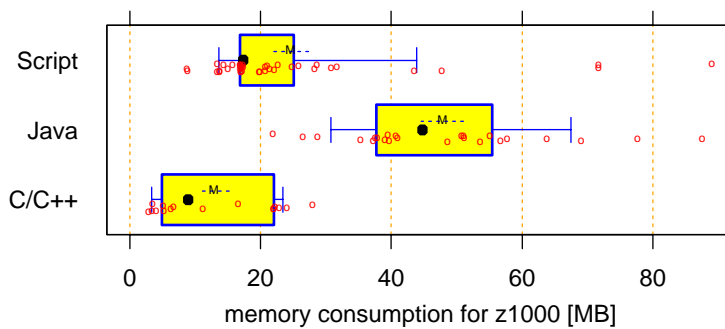


Figure 14: Like Figure 13, except that the languages are aggregated into groups. The bad/good ratios are 1.5 for Script and for Java and 4.5 for C/C++.

of the two major program designs used in the non-script languages (as described in Section 9). As we will see in Section 10.1, the latter point involves a time/memory tradeoff.

- Just like for run time, the variability in memory consumption from one programmer to the next tends to be substantially smaller in the script languages compared to the non-script languages, which may help reduce risk in software development.
- The very high memory consumption of the Java programs, where even the best ones are worse than most scripts, probably has two main reasons. For one, the improvements in the Java execution platforms have so far focused on execution time. However, garbage collection mechanisms (as present in Java) usually involve strong time/memory tradeoffs. Java run time systems can probably be made a lot more memory-efficient than they are today with only a small run time penalty. Second, at the time when these programs were written, Java was still quite young and a memory-conscious programming style was not yet widely known.

Summing up, for the phonecode problem one cannot generally speak of a memory penalty due to using script languages.

6 Results for program length and amount of commenting

How long are the programs?

How much commenting do they contain?

6.1 Results

Figure 15 shows the number of lines containing anything that contributes to the semantics of the program in each of the program source files, e.g. a statement, a declaration, or at least a delimiter such as a closing brace (end-of-block marker).

We see that non-scripts are typically two to three times as long as scripts. Even the longest scripts are shorter than the average non-script and even the shortest non-scripts are longer than the average script. Furthermore, the variability of program length for the scripts, at least for Perl and Python, is also smaller than for the non-scripts.

At the same time, scripts tend to contain a significantly higher density of comments (Figure 16), with the non-scripts averaging a median of 22% as many comment lines or commented lines as statement lines and the scripts averaging 34% ($p = 0.020$ when we compare the language groups rather than individual languages).

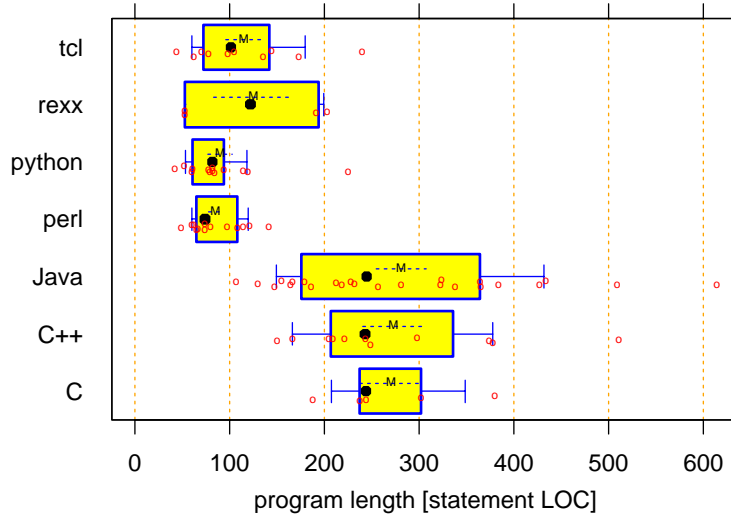


Figure 15: Program length, measured in number of non-comment source lines of code. The bad/good ratios range from 1.3 for C up to 2.1 for Java and 3.7 for Rexx.

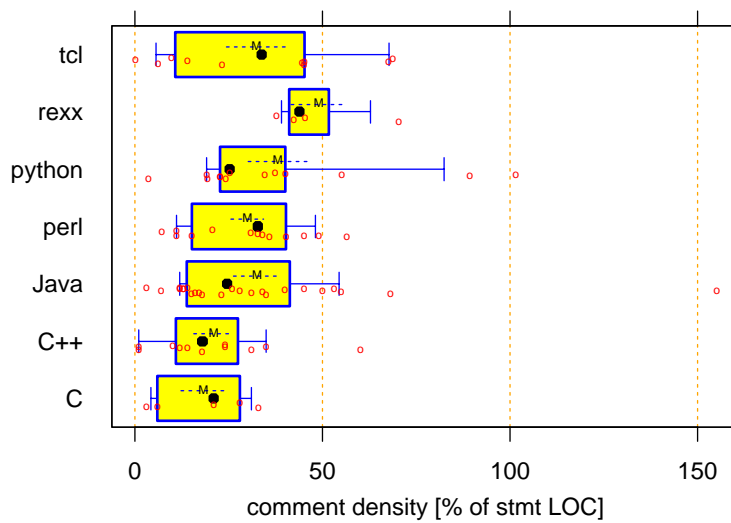


Figure 16: Percentage of comment lines plus commented statement lines, relative to the number of statement lines. The bad/good ratios range from 1.3 for Rexx up to 4.2 for Tcl.

6.2 Discussion

These results are quite spectacular. Scripts are not just shorter than “real” programs, they are very much shorter. Looking into the program source texts, one finds three reasons for that: First, the lack of declarations in the scripts; second, the generally more powerful (and thus compact) constructs for everyday operations such as opening and reading files (see the program examples in Section 2); and finally a different algorithm structure, which we will discuss further below in Section 9.

Many people believe that scripts tend to be less readable than non-scripts. Looking at some fixed amount of program source code, this may well be true: Scripts typically have more program logic packed into the same number of lines and will hence often appear more convoluted. However, the above results show that there are two effects that may result in an overall readability that is as good or even better than that of non-scripts. First, the much lower amount of program text in a script makes it much easier to have a complete overview of a program. Second, it may be that the denser logic in scripts provokes higher efforts at commenting the program, further reducing any readability penalty of script languages, if such a thing exists. However, we should take into account that the higher density of commenting may at least in part be due to the different programmer populations in the two major language groups.

7 Results for program reliability

Do the programs conform to the requirements specification?
How reliable are they?

7.1 Results for “standard” inputs

Each of the programs in this data set processes correctly the simple example dictionary and phone number input file that was given (including a file containing the expected outputs) to all participants for their program development.

However, with the large dictionary `woerter2` and the partially quite strange and unexpected “phone numbers” in the larger input files, not all programs behaved entirely correctly. The percentage of outputs correct is plotted in Figure 17.

5 programs (1 C, 1 C++, 1 Perl) produced no correct outputs at all, either because they were unable to load the large dictionary or because they were timed out during the load phase. 2 Java programs failed with near-zero reliability for other reasons and 1 Rexx program produced many of its outputs with incorrect formatting, resulting in a reliability of 45 percent. The only language with all flawless programs is Tcl.

If we ignore the above-mentioned highly faulty programs and compare the rest (that is, all programs with reliability over 50 percent, hence excluding 13% of the C/C++ programs, 8% of the Java programs, and 5% of the script programs; Figure 18) by language group, we find

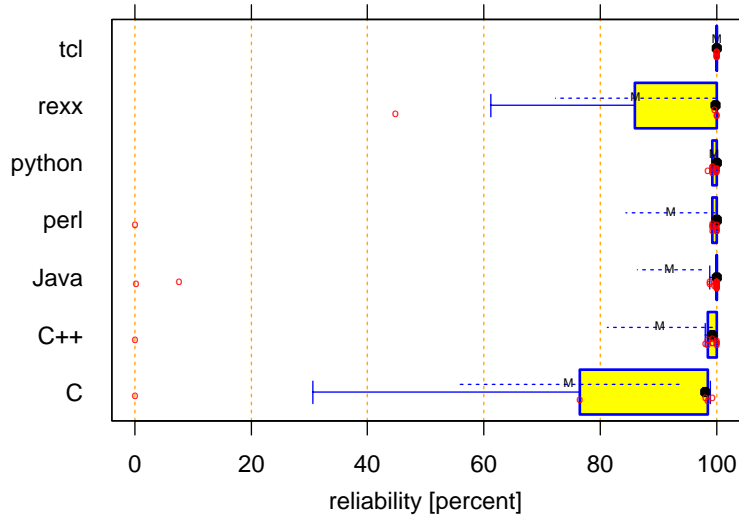


Figure 17: Program output reliability in percent for the z1000 input file.

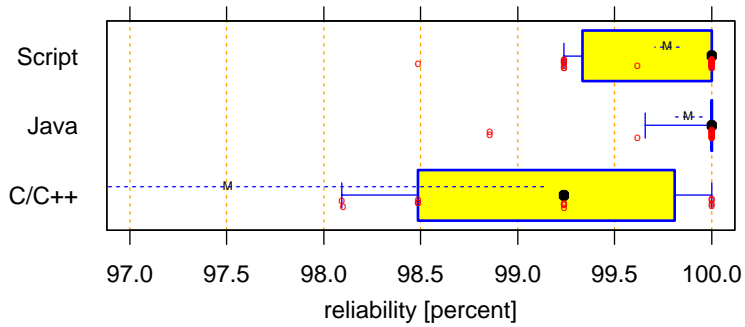


Figure 18: Program output reliability in percent (except for those programs with reliability below 50 percent), with languages aggregated into groups.

that C/C++ programs are on average less reliable than both the Java and the script programs ($p < 0.0004$ for the median, $p < 0.04$ for the mean).

7.2 Discussion

These latter differences all depend on just a few programs showing one or the other out of a small set of different behaviors and should hence not be over-generalized.

On the other hand, since these differences show exactly the same trend as the fractions of highly faulty programs mentioned above, there is some evidence that this ordering of reliability among the language groups in the present experiment may be real.

Remember, though, that the reliability advantage of the scripts may be due to the better test data available to the script programmers.

But at least it is fair to say there is no evidence in this data suggesting that script programs may be less reliable than non-scripts.

7.3 Results for “surprising” inputs

It is very instructive to compare the behavior on the more evil-minded input file m1000, again disregarding the programs already known as faulty as described above. The m1000 input set also contains phone numbers whose length and content is random, but in contrast to z1000 it even allows for phone numbers that do not contain any digits at all, only dashes and slashes. Such a phone number always has a correct encoding, namely an empty one, but one does not usually think of such inputs when reading the requirements. Hence the m1000 input file tests the robustness of the programs. The results are shown in Figures 19 and 20.

Most programs cope with this situation well, but half of the Java programs and 4 of the script programs (1 Tcl and 3 Python) crash when they encounter the first empty phone number (which happens after 10% of the outputs), usually due to an illegal string subscript or array subscript. The languages that do not perform array subscript checking may still produce an incorrect output for the empty phone numbers but at least survive the mistake and then continue to function properly. Note that the huge size of the box for the Java data in Figure 20 is quite arbitrary; it completely depends on the position of the first empty telephone number within the input file.

Except for this phenomenon, there are no large differences. 13 of the other programs (1 C, 5 C++, 4 Java, 2 Perl, 2 Python, 1 Rexx) fail exactly on the three empty phone numbers, but work allright otherwise, resulting in a reliability of 98.4%.

7.4 Discussion

Summing up, it appears warranted to say that the scripts are not less reliable than the non-scripts.

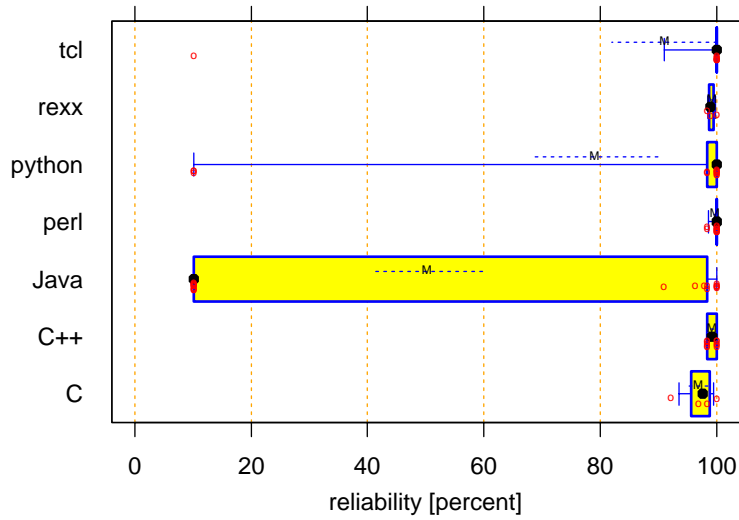


Figure 19: Program output reliability for the m1000 input file in percent (except for those programs whose z1000 reliability was below 50 percent), with languages aggregated into groups.

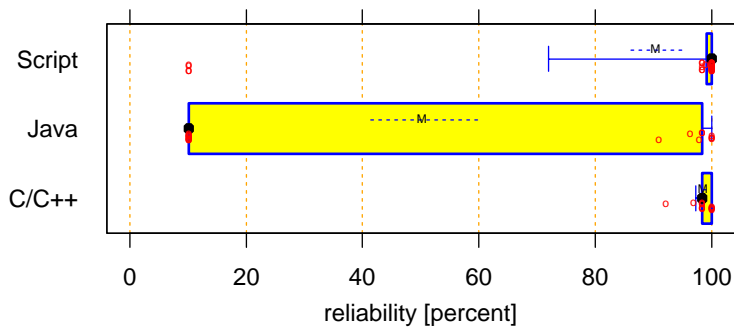


Figure 20: Program output reliability for the m1000 input file in percent (except for those programs whose z1000 reliability was below 50 percent), with languages aggregated into groups.

The only salient reliability difference in the given situation is not between scripts and non-scripts but rather between languages that perform array subscript checking and those that do not — in our particular measurement, the latter are at an advantage, but the same problems exist in these programs and are of course comparatively harder to find and fix.

8 Results for programmer work time

How long have the programmers taken to design, write, and test the program?

8.1 Results

Figures 21 and 22 show the total work time as reported by the script programmers and measured for the non-script programmers.

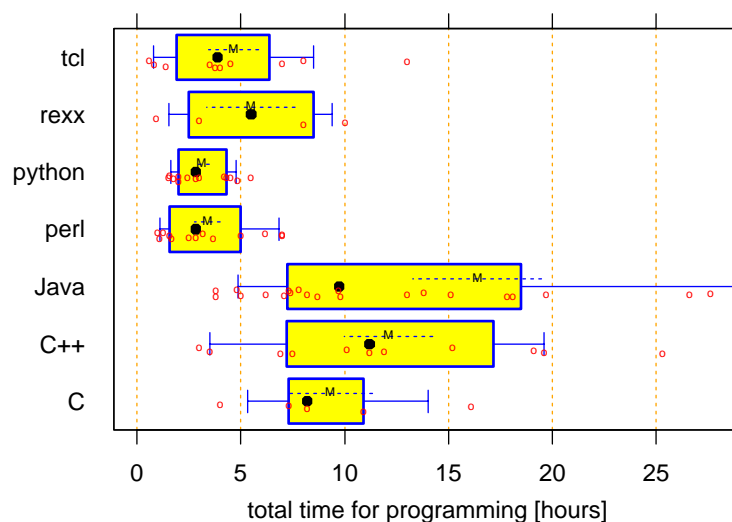


Figure 21: Total working time for realizing the program. Script group: times as measured and reported by the programmers. Non-script group: times as measured by the experimenter. The bad/good ratios range from 1.5 for C up to 3.2 for Perl. Three Java work times at 40, 49, and 63 hours are not shown.

As we see, scripts (total median 3.1 hours) take only about one third of the time required for the non-scripts (total median 10.0 hours). Note, however, that the meaning of the measurements is not exactly the same: First, non-script times always include the time required for reading the requirements (typically around 15 minutes), whereas many of the script participants apparently did not count that time. Second, some of the script participants estimated

(rather than measured) their work time. Third, we do not know whether all script participants were honest in their work time reporting. Fourth, and most importantly, all of the non-script participants started working on the solution immediately after reading the requirements, whereas some of the script participants started only days later but did not include the time in which they were thinking, perhaps subconsciously, about the program design in the meantime (see the quotes in Section 3.6 above).

8.2 Discussion

Even if we subtract a substantial discount from the times of the non-script programmers, these results are fairly spectacular: Scripts with the same functionality are written and tested in half the time of corresponding non-scripts. For the class of problems, to which these results can be generalized (which we cannot characterize based on this study, but which certainly is not small) and maintenance issues left aside (which we do not investigate empirically here; but see Sections 6.2 and 9), this implies a large improvement in programming productivity by replacing non-script languages by script languages — a trend one can already observe frequently in practice.

The scripting literature and folklore features a number of case studies where programs previously written in a non-script language were rewritten in a script language in only a fraction of the time. Even if one is willing to fully believe the numbers given, they do not mean very much: Usually, the first implementation of a program involves a lot of work for determining and refining the requirements and if the reimplementation is done by the same programmer, program design is also performed a lot quicker the second time around.

The present study provides data that is more realistic and also provides a comparison of not just individual cases, but of a statistical average for a substantial number of cases. No programmer in the sample has performed a re-implementation in a different language. But is the data credible?

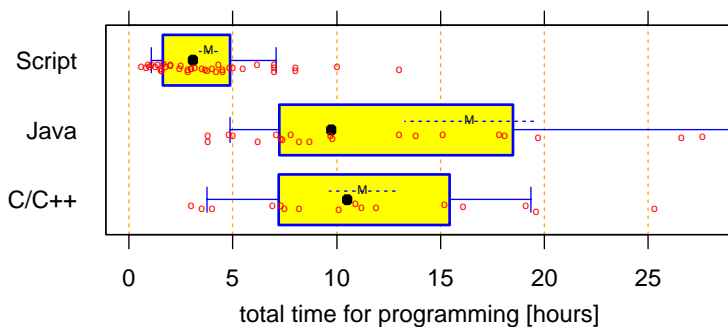


Figure 22: Like Figure 21, except that the languages are aggregated into larger groups. The bad/good ratio is 3.0 for Script, 2.6 for Java, and 2.1 for C/C++.

8.3 Validation

Fortunately, there is a way how we can check two things at once, namely the correctness of the work time reporting *and* the equivalence of the programmer capabilities in the script versus the non-script group. Note that both of these possible problems, if present, will tend to bias the script group work times downwards: we would expect cheaters to fake their time to be smaller, not larger, and we expect to see more capable programmers (rather than less capable ones) in the script group compared to the non-script group if the average programmer capabilities are any different.

This check relies on an old rule of thumb, which says that programmer productivity measured in lines of code per hour (LOC/hour) is roughly independent of the programming language: With a few extreme exceptions such as APL or Assembler, the time required for coding and testing a program will often be determined by the amount of functionality that can be expressed per line, but the time required per line will be roughly constant.

This rule is mostly an empirical one, but it can be explained by cognitive psychology: Once a programmer is reasonably fluent in a programming language, one statement or line of code is the most important unit of thinking (at least during coding and debugging phases). If that is dominant, though, the capacity limit of short term memory — 7 units plus or minus two — suggests that the effort required for constructing a program that is much longer than 7 lines may be roughly proportional to its number of lines, because the time required for correctly creating and handling one unit is constant and independent of the amount of information represented by the unit [17, 23].

Actually, two widely used effort estimation methods explicitly assume the productivity in lines of code per hour is independent of programming language:

The first is Barry Boehm's CoCoMo [3]. This popular software estimation model uses software size measured in LOC as an input and predicts both cost and schedule. Various so-called *cost drivers* allow adjusting the estimate according to, for instance, the level of domain experience, the level of programming language experience, the required program reliability etc. However, the level of programming language used is not one of these cost drivers, because, as Boehm writes, "*It was found [...] that the amount of effort per source statement was highly independent of language level.*" [3, p.477]. He also cites independent research suggesting the same conclusion, in particular a study from IBM by Walston and Felix [25].

The second is Capers Jones *language list* for the Function Point [1] method. Function Points are a software size metric that depends solely on program functionality and is hence independent of programming language [2]. Jones publishes a list [5] of programming languages, which indicates the value of LOC/FP for each language (the number of lines typically required to implement one function point) and also its so-called *language level* LL, a productivity factor indicating the number of function points that can be realized per time unit T with this language: $LL = FP/T$. T depends on the capabilities of the programmers etc. In this list, LL is exactly inversely proportional to LOC/FP; concretely $LL \cdot LOC/FP = 320$, which is just a different way of saying that the productivity of any language is a fixed 320 LOC per fixed time unit T. Independent studies confirming language productivity differences with respect

Table 3: Excerpt from Capers Jones' programming language table for the languages used in this study. LL is the language level and LOC/FP is the number of lines of code required per function point. See the main text for an explanation.

language	LL	LOC/FP
C	3.5	91
C++	6	53
Java	6	53
Perl	15	21
Python	—	—
Rexx	7	46
Tcl	5	64

to function points per time have also been published, e.g. [13]. Table 3 provides the relevant excerpt from the language table and Figure 23 relates this data to the actual productivity observed in the present study.

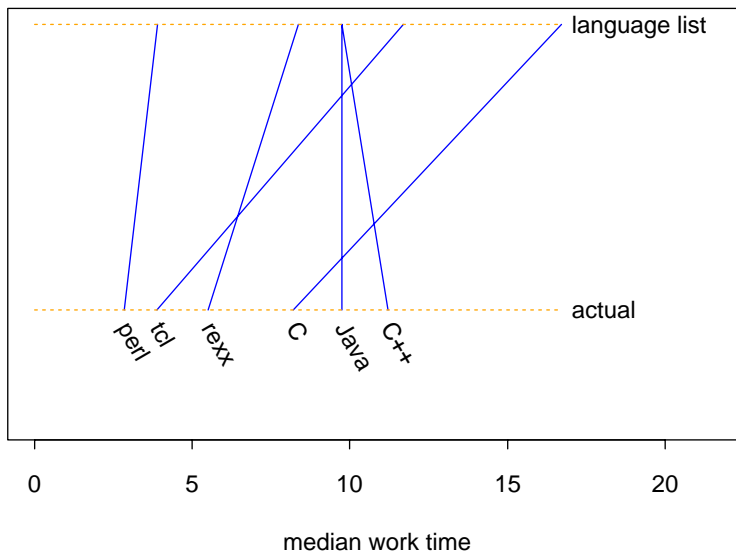


Figure 23: Actual median work times of each language compared to those we would expect from the relative productivity as given in Capers Jones' programming language list [5], normalized such that the Java work times are exactly as predicted. We find that the language list underestimates the productivity of C and Tcl for this problem. For the phonocode problem, C is almost as well-suited as C++ at least given the approaches used by most participants, in contrast to the language levels indicated by Jones. For Tcl, the given language level of 5 may be a typo which should read "15" instead. For the other languages, the prediction of the table is reasonably accurate.

So let us accept the statement "the number of lines written per hour is independent of programming language" as a rule of thumb for this study. The validation of our work time data

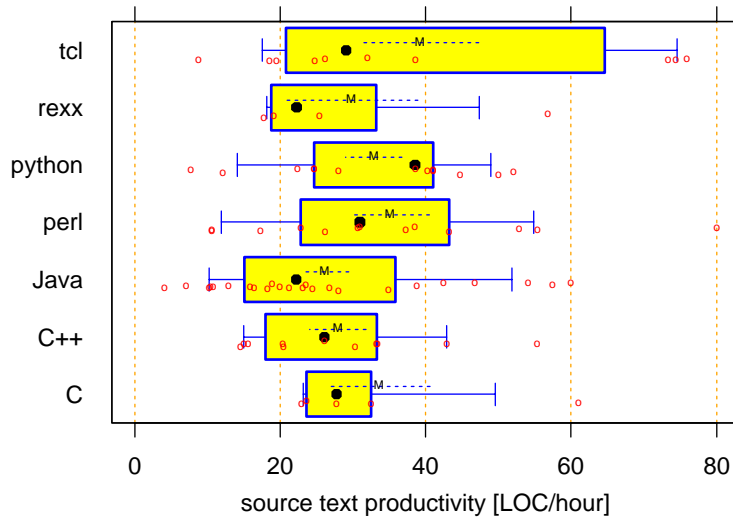


Figure 24: Source text productivity in non-comment lines of code per total work hour. The bad/good ratios range from 1.4 for C up to 3.1 for Tcl.

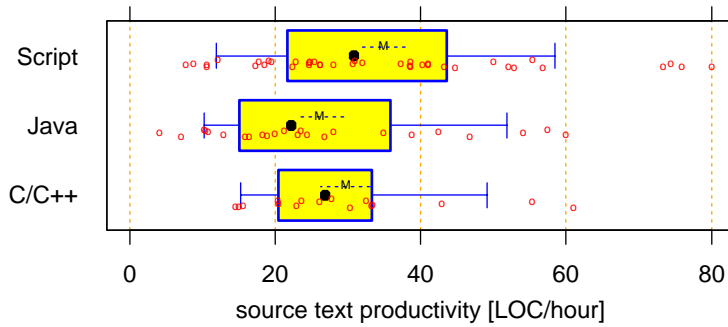


Figure 25: Like Figure 24, except that the languages are aggregated into groups. The bad/good ratio is 2.0 for Script, 2.4 for Java, and 1.6 for C/C++.

based on this rule is plotted in Figure 24. Judging from the productivity range of Java, all data points except maybe for the top three of Tcl and the top one of Perl are quite believable. In particular, except for Python, all medians are in the range 22 to 31. The variability between programmers is also similar across the languages. Hence, the LOC productivity plot lends a lot of credibility to the reported times: Only four productivity values overall are outside the (reliable) range found in the experiment for the non-script programs.

None of the median differences are clearly statistically significant, the closest being Java versus C, Perl, Python, or Tcl where $0.07 \leq p \leq 0.10$.

Even in the aggregated view (Figure 25) with its much larger groups, the difference between C/C++ and scripts is not significant ($p = 0.22$), only the difference between Java and scripts is ($p = 0.031$), the difference being at least 5.2 LOC/hour (with 80% confidence).

This comparison lends a lot of credibility to the work time comparison shown above. The times reported for script programming are probably only modestly too optimistic, if any, so that a work time advantage for the script languages of about factor two holds.

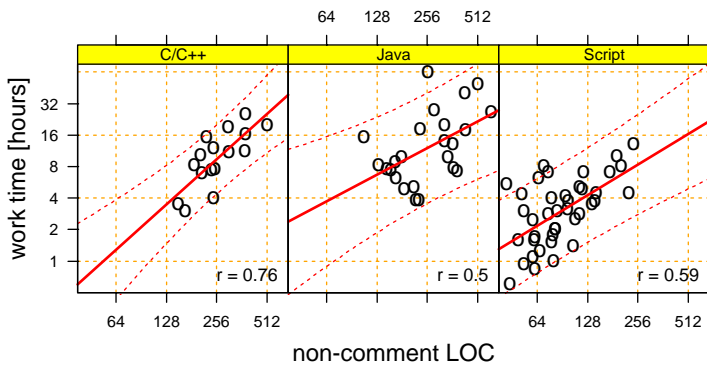


Figure 26: The same data as in Figure 25, except that the program lengths and work times are shown separately. The lines are a standard least squares regression line and its 90% prediction interval. Note the logarithmic axes. r is the correlation coefficient of the logarithmic data.

Figure 26 shows the same data as a two-dimensional plot including a regression line that could be used for (logarithmically) predicting work time from expected size. The higher productivity of the script languages shows up as a trend line lying lower in the plot. The C/C++ line is steeper than the others, which in this logarithmic plot shows non-linear increase of effort: programs that are twice as long take more than twice as much work time. This is probably due to the fact that the best C/C++ programmers not only were more productive but also wrote more compact code.

9 Discussion of program structure

If one considers the program designs (i.e., algorithms and data structures) typically chosen in the various languages, there is a striking difference.

9.1 Results for scripts

Most of the programmers in the script group used the associative arrays provided by their language and stored the dictionary words to be retrieved by their number encodings. The search algorithm simply attempts to retrieve from this array, using prefixes of increasing length of the remaining rest of the current phone number as the key. Any match found leads to a new partial solution to be completed later. See three rather different implementations of this algorithm in the Figures 27, 28, and 29.

These programs were selected from their respective group according to the following criteria: Short (so the code can be displayed on one page), efficient (in terms of execution time), and written in a short time (and hence hopefully by a rather competent programmer).

9.2 Discussion

The Tcl script has arguably the simplest and most straightforward formulation of the algorithm. However, this clarity comes at the price of a rather broad call interface (with four parameters) and printing of the results from within the search routine (which is no good modularization). Furthermore, the Tcl script takes 9 times as long for execution as the Python and Perl scripts.

The Python version avoids all three of these drawbacks and is still very easy to understand. However, it fails to handle phone numbers with no digits correctly.

The Perl program, in comparison, is fairly convoluted and hard-to-understand, but is both correct and efficient.

I have not attempted to verify whether the programming styles represented by these three somewhat arbitrarily selected programs are in any way typical among the scripts of each language group (though many people may expect they are). However, please note that the styles are by no means preferred or triggered — let alone enforced — by particular features of the respective language. Quite on the contrary: Each of the variants can straightforwardly be translated into each of the other two languages without changing its structure.

9.3 Results for non-scripts

In contrast, essentially all of the non-script programmers chose either of the following solutions. In the simple case, they simply store the whole dictionary in an array, usually in

```

sub encode {
    my $phonenum = shift; # retrieve argument
    my @results = &encodeByWords($phonenum);
    return if @results && !defined $results[0]; #antibacktrack
    return @results;
    return $phonenum if length $phonenum == 1;
    my $first = substr $phonenum,0,1;
    @results = &encodeByWords(substr $phonenum,1);
    return unless defined $results[0];
    return map {"$first $_"} @results; #prepend $first everywhere
}

sub encodeByWords {
    my $phonenum = shift; # retrieve argument
    my @results, $hasword, $head; # make variables local
    for my $len (1..length($phonenum)-1) { # for increasing prefixes
        next unless exists $words{$head = substr $phonenum,0,$len};
        $hasword++; #word found for prefix substring
        my @found=&encode(substr $phonenum,$len); # encode rest
        for my $tail (@found) { # concatenate each head with each tail
            push @results, map {"$_ $tail"} @{$words{$head}};
        }
    }
    # end recursion: encode by just one word, if possible:
    push @results, @{$words{$phonenum}} if exists $words{$phonenum};
    return undef if $hasword && !@results; # no complete encoding found?
    return @results; # else: return all encodings thus found
}

```

Figure 27: Example Perl search routine from program s149103, the third-fastest-written, fastest-running, and shortest Perl script. Identifiers and comments have been changed to maximize readability. `words` is the associative array that is indexed by digit strings (partial phone numbers) and contains as values lists of words matching these partial numbers.

both the original character form and the corresponding phone number representation. They then select and test one tenth of the whole dictionary for each digit of the phone number to be encoded, using only the first digit as a key to constrain the search space. This leads to a simple, but inefficient solution.

The more elaborate case uses a 10-ary tree in which each node represents a certain digit, nodes at height n representing the n -th character of a word. A word is stored at a node if the path from the root to this node represents the number encoding of the word. This is the most efficient solution, but it requires a comparatively large number of statements to implement the tree construction. In Java, the large resulting number of objects also leads to a high memory consumption due to the severe per-object memory overhead incurred by current implementations of the language.

```

def encode(phonenumber, digitAllowed=1):
    results = []
    for i in range(1, len(phonenumber)+1): # for increasing substrings
        s1 = phonenumber[:i]             # split number into two parts
        s2 = phonenumber[i:]
        if words.has_key(s1):             # first part has an encoding
            digitAllowed = 0 # no single digit needed or allowed
            if s2: # second part is not empty, try to encode it
                for m in encode(s2):
                    for w in words[s1]:
                        # combine encodings for first and second part
                        results.append("%s %s" % (w, m))
            else: # remainder is empty, first part is result
                results.extend(words[s1])
    if digitAllowed:
        # no results found starting here AND
        # we also did not insert a digit just before
        if phonenumber[1:]: # match the rest
            for m in encode(phonenumber[1:], 0):
                # combine first digit with further encodings found
                results.append("%s %s" % (phonenumber[0], m))
        else:
            # last character matches itself.
            results.append(phonenumber[0])
    return results

```

Figure 28: Example Python search routine from program s149205, the fastest-written and fastest-running, but only fifth shortest Python script. This script fails and crashes on empty phone numbers. Identifiers and comments have been changed to maximize readability. “words” is the associative array that is indexed by digit strings (partial phone numbers) and contains as values lists of words matching these partial numbers.

9.4 Global discussion

In light of these results, the shorter program length of the script programs can be explained by the fact that most of the actual search is done simply by the hashing algorithm used internally by the associative arrays. In contrast, the non-script programs with their array or tree implementations require most of these mundane elementary steps of the search process to be coded explicitly by the programmer. This is further pronounced by the effort (or lack of it) for data structure declarations.

The larger variation in execution time and in memory consumption that was found for the non-scripts can be explained by the two very different dictionary lookup mechanisms employed within each of these groups (one being memory-conservative but having effort linear in the number of words, the other requiring larger administrative memory overhead but providing logarithmic access time).

It is an interesting observation that despite the existence of hash table implementations in both the Java and the C++ class libraries none of the non-script programmers used them (but

```

proc encode {phonestring phonenumber resultwords wasdigit} {
  if {![string length $phonenumber]} { # nothing more to encode
    puts "$phonestring: [join $resultwords { }]" # print $resultwords found
    return
  }
  set found 0
  for {set i 0} {$i<[string length $phonenumber]} {incr i} {
    set subnumber [string range $phonenumber 0 $i] # pick number prefix
    set subwords {}; catch {set subwords $::words($subnumber)} # find words
    foreach subword $subwords { # for each word found for prefix
      encode $phonestring [string range $phonenumber [expr $i+1] end] \
        [concat $resultwords [list $subword]] 0 # encode remainder
      set found 1
    }
  }
  if {!$found && !$wasdigit} { # insert one digit, encode the rest
    encode $phonestring [string range $phonenumber 1 end] \
      [concat $resultwords [string range $phonenumber 0 0]] 1
  }
}

```

Figure 29: Example Tcl search routine from program s149407, the fastest-written, seventh-fastest-running, and shortest Tcl script. Identifiers and comments have been changed to maximize readability. “words” is the associative array that is indexed by digit strings (partial phone numbers) and contains as values lists of words matching these partial numbers.

rather implemented a tree solution by hand), whereas for almost all of the script programmers the hash tables built into the language were the obvious choice.

10 Testing two common rules of thumb

Having so many different implementations of the same requirements allows for a nice test of two common rules of thumb in programming:

- The time/memory tradeoff: To make a program run faster, one will often need to use more memory.
- The elegance-is-efficient rule: A shorter (in terms of lines of code) solution to the same problem will often also run faster than a longer one.

10.1 The run time versus memory consumption tradeoff

The time/memory tradeoff is a frequent effect in algorithm design: An algorithm can often be made asymptotically faster if it is allowed to use an asymptotically larger data structure.

Similar effects may also occur on a more tactical level in programming, e.g. by precomputing and storing certain results rather than recomputing them each time they are required, or by storing data in a more uniform manner rather than in the most compact way.

Can a time/memory tradeoff be found in the programs of our study for one or several languages?

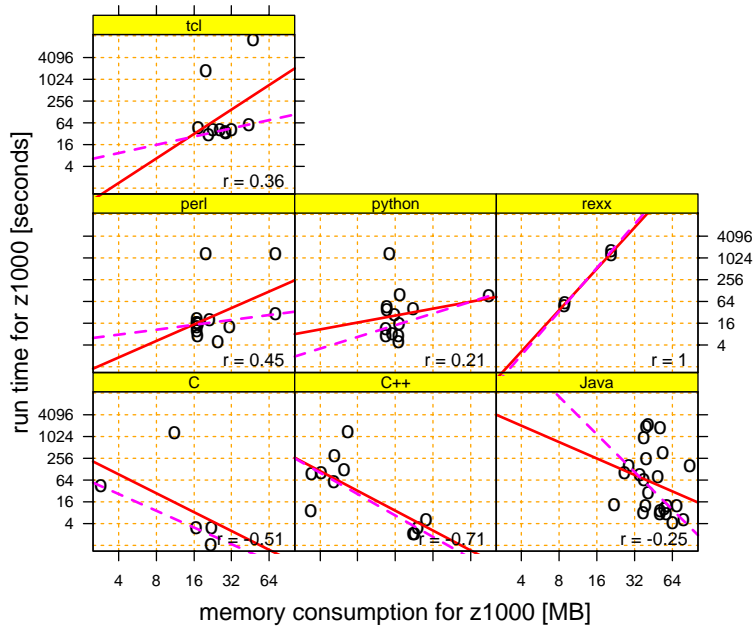


Figure 30: Memory consumption versus program run time. The thick line is a least squares regression trend line; the dashed line is a least absolute distance trend line. r denotes the correlation (computed on the logarithms of the values). Note the logarithmic axes.

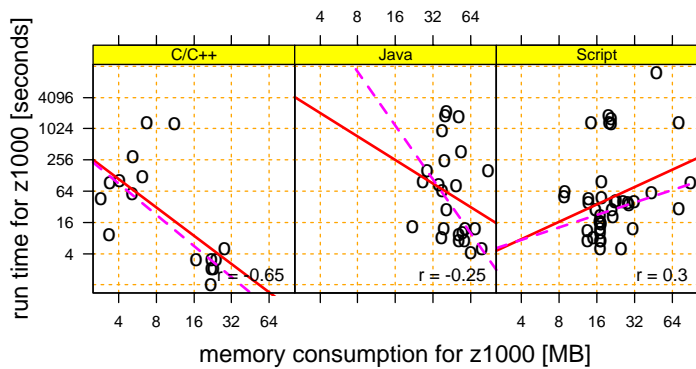


Figure 31: The same data as in Figure 30, by language group. Note the logarithmic axes.

The time/memory tradeoff is shown for the individual languages in Figure 30 and for the language groups in Figure 31.

Apparently, the rule is quite correct for all three non-script languages: The faster running C and C++ programs require quite clearly more memory than the slower ones. A similar effect appears to be present for Java, though much more blurred because Java gives more opportunities for memory inefficiency at a tactical level.

The reason in all three cases is the design dichotomy described above between programs using the simpler group-by-first-character-only approach on the one hand and programs implementing the full 10-ary search tree on the other hand.

For the script languages, the opposite rule tends to be true: Those programs that use more memory actually tend to be slower (rather than faster) than the others, because given the associative array design of the scripts, additional memory does not contribute to algorithmic efficiency, but rather represents tactical inefficiency only.

10.2 Are shorter programs also more efficient?

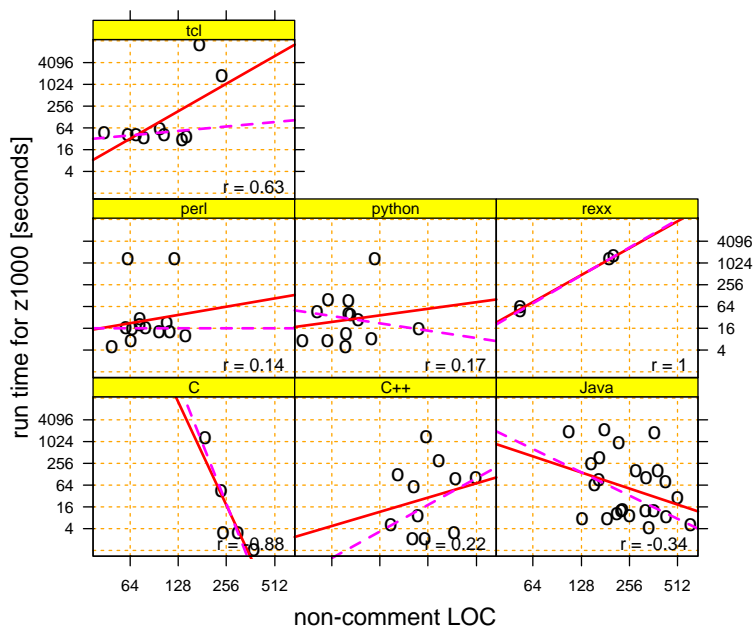


Figure 32: Program length versus program run time. The thick line is a least squares regression trend line; the dashed line is a least absolute distance trend line. Note the logarithmic axes.

The data for the elegance-is-efficient rule is shown for the individual languages in Figure 32 and for the language groups in Figure 33. The evidence for this rule is rather mixed: For the

phonecode problem, a somewhat longer program appears to be required for full efficiency in both C and Java — in opposition to the rule; see also Section 9. In contrast, for some strange reason the rule appears to hold for C++, at least for the data we have here. For the individual script languages, program length appears to have rather little influence on run time, if we discount the Rexx programs (where there are only 4 programs from 3 different programmers) and the two extremely slow Tcl programs. However, if we consider all script programs together as in Figure 33, there is quite some evidence that the rule may be correct: with 90% confidence, the correlation is larger than 0.25.

11 Metrics for predicting programmer performance

In practical software engineering situations it would often be very useful if one could predict an unknown programmer's performance in advance, in particular the time required to complete a task and various quality attributes of the resulting program such as those investigated in this study.

The large number of directly comparable data points in this study allows for testing several possible approaches to such predictions.

11.1 Results for objective metrics

The non-script programmers were asked several questions about their previous programming experience, as described in detail in [21]. Examples are number of years of programming experience, total amount of program code written, size of largest program ever written, etc.

None of these questions had any substantial predictive value for any aspect of programmer performance in the experiment, so I will not delve into this data at all.

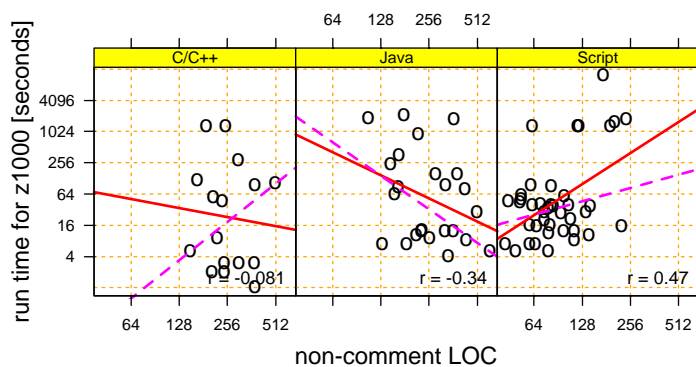


Figure 33: The same data as in Figure 32, by language group. r denotes the correlation (computed on the logarithms of the values). Note the logarithmic axes.

11.2 Results for programmer self-rating

In contrast, the script programmers (but unfortunately not the non-script programmers) were asked the following question:

```
# Overall I tend to rate myself as follows compared to all other programmers
# (replace one dot by an X)
# among the upper 10 percent      .
# upper 11 to 25 percent         .
# upper 25 to 40 percent         .
# upper 40 to 60 percent         .
# lower 25 to 40 percent         .
# lower 11 to 25 percent         .
# lower 10 percent               .
```

On this scale, the programmers of as many as 14 of the scripts (35%) rated themselves among the upper 10 percent and those of another 15 (37.5%) among the top 10 to 25. The programmers of only 9 scripts (22.5%) rated themselves lower than that and 2 (5%) gave no answer. Across languages, there are no large self-rating differences: If we compare the sets of self-ratings per language to one another, using a Wilcoxon Rank Sum Test with normal approximation for ties, no significant difference is found for any of the language pairs ($0.58 < p < 0.94$).

As for correlations of self-rating and actual performance, I found that higher self-ratings tend to be somewhat associated with lower run time (as illustrated in Figure 34; the rank correlation is -0.33) and also with shorter work time for producing the program (Figure 35; the rank correlation is -0.30).

No clear association was found for memory consumption, program length, comment length, comment density, or program reliability.

Summing up, to predict how good a programmer will perform compared to others, it may be the best to just ask him or herself for an opinion of his or her general capabilities. This is not to be confused with direct personal estimates of, say, expected effort for one particular task.

12 Conclusions

The following statements summarize the findings of the comparative analysis of 80 implementations of the phonecode program in 7 different languages:

- Designing and writing the program in Perl, Python, Rexx, or Tcl takes only about half as much time as writing it in C, C++, or Java and the resulting program is only half as long.
- No clear differences in program reliability between the language groups were observed.
- The typical memory consumption of a script program is about twice that of a C or C++ program. For Java it is another factor of two higher.

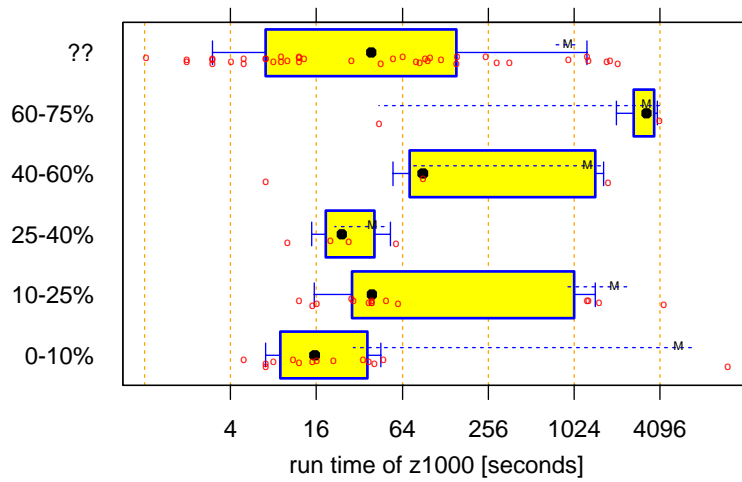


Figure 34: Relationship between self-rating and program efficiency: higher self-rating is correlated with faster programs. The uppermost boxplot represents all non-script programs. Note the logarithmic axis.

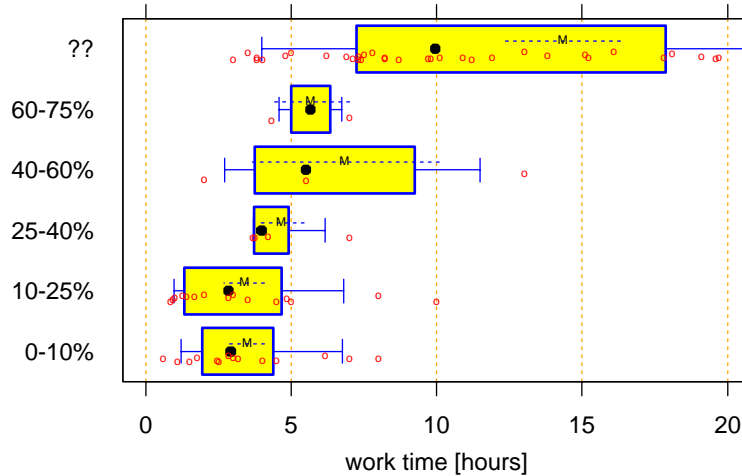


Figure 35: Relationship between self-rating and working time for writing the program: higher self-rating is correlated with shorter work time. The uppermost boxplot represents all non-script programs.

- For the initialization phase of the phonecode program (reading the 1 MB dictionary file and creating the 70k-entry internal data structure), the C and C++ programs have a strong run time advantage of about factor 3 to 4 compared to Java and about 5 to 10 compared to the script languages.
- For the main phase of the phonecode program (search through the internal data structure), the advantage in run time of C or C++ versus Java is only about factor 2 and the script programs even tend to be faster than the Java programs.
- Within the script languages, Python and in particular Perl are faster than Rexx and Tcl for both phases.
- For all program aspects investigated, the performance variability due to different programmers (as described by the bad/good ratios) tends to be larger than the variability due to different languages.

Due to the large number of implementations and broad range of programmers investigated, these results, when taken with a grain of salt, are probably reliable despite the validity threats discussed in Section 3.6. We can expect to find similar results for most members of the frequent class of programs that performs sequential transcoding of input records into output records based on some kind of translation vocabulary. Generalizing the results to very different application domains, however, would be haphazard.

It is likely that for many other problems the results for the script group of languages would not be quite as good as they are. However, I would like to emphasize that the phonecode problem was not chosen so as to make the script group of languages look good — it was originally developed as a non-trivial, yet well-defined benchmark for programmers' ability of writing reliable programs.

I conclude the following::

- The so-called “scripting languages” Perl, Python, Rexx, and Tcl can be preferable alternatives to “conventional” languages such as C or C++ even for tasks that need to handle fair amounts of computation and data. Their relative run time and memory consumption overhead will often be acceptable and they may offer significant advantages with respect to programmer productivity — at least for small programs like the phonecode problem.
- Within the group of scripting languages, Tcl has the caveat of substantially longer minimum execution time compared to Perl and Python. Perl may have the caveat of requiring more disciplined programmers for obtaining readable programs compared to Tcl and Perl.
- Interpersonal variability, that is, the capability and behavior differences between programmers using the same language, tends to account for more differences between programs than a change of the programming language.

12.1 Further reading

If you are interested in further information about how scripting languages compare to one another or to conventional languages, you can find some statements from the point of view of the Python people on <http://www.python.org/doc/essays/comparisons.html>; from the point of view of the Perl people on <http://www.perl.com/language/versus/>; and from the point of view of the Tcl people on <http://www.scriptics.com/advocacy/>. A wealth of comparisons written by others is collected on the Python website on <http://www.python.org/doc/Comparisons.html>.

If you want to learn how to write efficient programs in scripting languages, a very useful short article is <http://www.python.org/doc/essays/list2str.html>. It was written for Python, but most of its lessons apply to other script languages as well. Some of the language books such as [24] for Perl and [16] for Python also contain sections on writing efficient programs.

A Appendix: Specification of the phonecode programming problem

The problem solved by the participants of this study (i.e. the authors of the programs investigated here) was called *phonecode*.

The exact problem description given to the subjects in the non-script group is printed in the appendix of [21]. The following subsections reproduce the description given on the web page for the participants of the script group. It is equivalent with respect to the functional requirements of the program, but had to be different with respect to the submission procedure etc.

Underlined parts of the text were hyperlinks in the original web page.

A.1 The procedure description

(First few paragraphs left out)

The purpose of this website is collecting many implementations of this same program in scripting languages for comparing these languages with each other and with the ones mentioned above. The languages in question are

- *Perl*
- *Python*
- *Rexx*
- *Tcl*

The properties of interest for the comparison are

- *programming effort*

54A APPENDIX: SPECIFICATION OF THE PHONECODE PROGRAMMING PROBLEM

- program length
- program readability/modularization/maintainability
- elegance of the solution
- memory consumption
- run time consumption
- correctness/robustness

Interested?

If you are interested in participating in this study, please create your own implementation of the Phonocode program (as described below) and send it to me by email.

I will collect programs **until December 18, 1999**. After that date, I will evaluate all programs and send you the results.

The **effort** involved in implementing phonocode depends on how many mistakes you make underways. In the previous experiment, very good programmers typically finished in about 3 to 4 hours, average ones typically take about 6 to 12 hours. If anything went badly wrong, it took much longer, of course; the original experiment saw times over 20 hours for about 10 percent of the participants. On the other hand, the problem should be much easier to do in a scripting language compared to Java/C/C++, so you can expect much less effort than indicated above.

Still interested?

Great! The procedure is as follows:

1. Read the task description for the “phonocode” benchmark. This describes what the program should do.
2. Download
 - the small test dictionary test.w,
 - the small test input file test.t,
 - the corresponding correct results test.out,
 - the real dictionary woerter2,
 - a 1000-input file z1000.t,
 - the corresponding correct results z1000.out,
 - or **all of the above** together in a single zip file.
3. Fetch this program header, fill it in, convert it to the appropriate comment syntax for your language, and use it as the basis of your program file.
4. Implement the program, using only a single file.
(Make sure you measure the time you take separately for design, coding and testing/debugging.)
Once running, test it using test.w, test.t, test.out only, until it works for this data. Then and only then start testing it using woerter2, z1000.t, z1000.out.
This restriction is necessary because a similar ordering was imposed on the subjects of the original experiment as well – however, it is not helpful to use the large data earlier, anyway.
5. A note on testing:

- Make sure your program works correctly. When fed with `woerter2` and `z1000.t` it must produce the contents of `z1000.out` (except for the ordering of the outputs). To compare your actual output to `z1000.out`, sort both and compare line by line (using `diff`, for example).
 - If you find any differences, but are convinced that your program is correct and `z1000.out` is wrong with respect to the task description, then re-read the task description very carefully. Many people misunderstand one particular point.
(I absolutely guarantee that `z1000.out` is appropriate for the given requirements.)
If (and only if!) you still don't find your problem after re-reading the requirements very carefully, then read this [hint](#).
6. Submit your program by email to prechelt@ira.uka.de, using **Subject: phonecode submission** and preferably inserting your program as plain text (but watch out so that your email software does not insert additional line breaks!)
 7. Thank you!

Constraints

- Please make sure your program runs on Perl 5.003, Python 1.5.2, Tcl 8.0.2, or Rexx as of Regina 0.08g, respectively. It will be executed on a Solaris platform (SunOS 5.7), running on a Sun Ultra-II, but should be platform-independent.
- Please use only a single source program file, not several files, and give that file the name `phonecode.xx` (where `xx` is whatever suffix is common for your programming language).
- Please do not over-optimize your program. Deliver your first reasonable solution.
- Please be honest with the work time that you report; there is no point in cheating.
- Please design and implement the solution alone. If you cooperate with somebody else, the comparison will be distorted.

A.2 Task requirements description

Consider the following mapping from letters to digits:

<i>E</i>	<i>J N Q</i>	<i>R W X</i>	<i>D S Y</i>	<i>F T</i>	<i>A M</i>	<i>C I V</i>	<i>B K U</i>	<i>L O P</i>	<i>G H Z</i>
<i>e</i>	<i>j n q</i>	<i>r w x</i>	<i>d s y</i>	<i>f t</i>	<i>a m</i>	<i>c i v</i>	<i>b k u</i>	<i>l o p</i>	<i>g h z</i>
<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>

We want to use this mapping for encoding telephone numbers by words, so that it becomes easier to remember the numbers.

Functional requirements

Your task is writing a program that finds, for a given phone number, all possible encodings by words, and prints them. A phone number is an arbitrary(!) string of dashes (`-`), slashes (`/`) and digits. The dashes and slashes will not be encoded. The words are taken from a dictionary which is given as an alphabetically sorted ASCII file (one word per line).

Only exactly each encoding that is possible from this dictionary and that matches the phone number exactly shall be printed. Thus, possibly nothing is printed at all. The words in the dictionary contain letters (capital or small, but the difference is ignored in the sorting), dashes (`-`) and double quotes (`"`).

56A APPENDIX: SPECIFICATION OF THE PHONECODE PROGRAMMING PROBLEM

For the encoding only the letters are used, but the words must be printed in exactly the form given in the dictionary. Leading non-letters do not occur in the dictionary.

Encodings of phone numbers can consist of a single word or of multiple words separated by spaces. The encodings are built word by word from left to right. If and only if at a particular point no word at all from the dictionary can be inserted, a single digit from the phone number can be copied to the encoding instead. Two subsequent digits are never allowed, though. To put it differently: In a partial encoding that currently covers k digits, digit $k + 1$ is encoded by itself if and only if, first, digit k was not encoded by a digit and, second, there is no word in the dictionary that can be used in the encoding starting at digit $k + 1$.

Your program must work on a series of phone numbers; for each encoding that it finds, it must print the phone number followed by a colon, a single(!) space, and the encoding on one line; trailing spaces are not allowed.

All remaining ambiguities in this specification will be resolved by the following **example**. (Still remaining ambiguities are intended degrees of freedom.)

Dictionary (in file test.w):

an
blau
Bo"
Boot
bo"s
da
Fee
fern
Fest
fort
je
jemand
mir
Mix
Mixer
Name
neu
o"d
Ort
so
Tor
Torf
Wasser

Phone number list (in file test.t):

112
5624-82
4824
0721/608-4067
10/783--5
1078-913-5

381482
04824

Program start command:

```
phonecode test.w test.t
```

Corresponding correct program output (on screen):

```
5624-82: mir Tor  
5624-82: Mix Tor  
4824: Torf  
4824: fort  
4824: Tor 4  
10/783--5: neu o"d 5  
10/783--5: je bo"s 5  
10/783--5: je Bo" da  
381482: so 1 Tor  
04824: 0 Torf  
04824: 0 fort  
04824: 0 Tor 4
```

Any other output would be wrong (except for different ordering of the lines).

Wrong outputs for the above example would be e.g.

```
562482: Mix Tor, because the formatting of the phone number is incorrect,  
10/783--5: je bos 5, because the formatting of the second word is incorrect,  
4824: 4 Ort, because in place of the first digit the words Torf, fort, Tor could be used,  
1078-913-5: je Bo" 9 1 da , since there are two subsequent digits in the encoding,  
04824: 0 Tor , because the encoding does not cover the whole phone number, and  
5624-82: mir Torf , because the encoding is longer than the phone number.
```

The above data are available to you in the files `test.w` (dictionary), `test.t` (telephone numbers) and `test.out` (program output).

Quantitative requirements

Length of the individual words in the dictionary: 50 characters maximum.

Number of words in the dictionary: 75000 maximum

Length of the phone numbers: 50 characters maximum.

Number of entries in the phone number file: unlimited.

Quality requirements

Work as carefully as you would as a professional software engineer and deliver a correspondingly high grade program. Specifically, thoroughly comment your source code (design ideas etc.).

The focus during program construction shall be on correctness. Generate exactly the right output format right from the start. Do not generate additional output. I will automatically test your program with hundreds of thousands of phone numbers and it should not make a single mistake, if possible — in particular it must not crash. Take yourself as much time as is required to ensure correctness.

Your program must be run time efficient in so far that it analyzes only a very small fraction of all dictionary entries in each word appending step. It should also be memory efficient in that it

does not use 75000 times 50 bytes for storing the dictionary if that contains many much shorter words. The dictionary must be read into main memory entirely, but you must not do the same with the phone number file, as that may be arbitrarily large.

Your program need *not* be robust against incorrect formats of the dictionary file or the phone number file.

A.3 The hint

The “hint” referred to in the procedure description shown in Section A.1 actually refers to a file containing only the following:

Hint

Please do not read this hint during preparation.

*Read it only if you **really** cannot find out what is wrong with your program and why its output does not conform to z1000.out although you think the program must be correct.*

If, and only if, you are in that situation now, read the actual hint.

The link refers to the following file:

Hint

If your program finds a superset of the encodings shown in z1000.out, you have probably met the following pitfall.

Many people first misunderstand the requirements with respect to the insertion of digits as follows. They insert a digit even if they have inserted a word at some point, but could then not complete the encoding up to the end of the phone number. That is, they use backtracking.

This is incorrect. Encodings must be built step-by-step strictly from left to right; the decision whether to insert a digit or not is made at some point and, once made, must never be changed.

Sorry for the confusion. The original test had this ambiguity and to be able to compare the new work times with the old ones, the spec must remain as is. If you ran into this problem, please report the time you spent finding and repairing; put the number of minutes in the 'special events' section of the program header comment. Thanks a lot!

B Appendix: Raw data

Below you find the most important variables from the raw data set analyzed in this report. The meaning of the variables is (left to right): subject ID (person), programming language (lang), run time for z1000 input file in minutes (z1000t), run time for z0 input file in minutes (z0t), memory consumption at end of run for z1000 input file in kilobytes (z1000mem), program length in statement lines of code (LOC), output reliability for z1000 input file in percent (z1000rel), output reliability for m1000 input file in percent (m1000rel), total subject work time in hours (work), subject's answer to the capability question “I consider myself to be among the top X percent of all programmers” (caps).

person	lang	z1000t	z0t	z1000mem	LOC	z1000rel	m1000rel	work	caps
s018	C	0.017	0.017	22432	380	98.10	96.8	16.10	??
s030	C	0.050	0.033	16968	244	76.47	92.1	4.00	??
s036	C	20.900	0.000	11440	188	0.00	89.5	8.20	??
s066	C	0.750	0.467	2952	237	98.48	100.0	7.30	??
s078	C	0.050	0.050	22496	302	99.24	98.4	10.90	??
s015	C++	0.050	0.050	24616	374	99.24	100.0	11.20	??
s020	C++	1.983	0.550	6384	166	98.48	98.4	3.00	??
s021	C++	4.867	0.017	5312	298	100.00	98.4	19.10	??
s025	C++	0.083	0.083	28568	150	99.24	98.4	3.50	??
s027	C++	1.533	0.000	3472	378	98.09	100.0	25.30	??
s033	C++	0.033	0.033	23336	205	99.24	98.4	10.10	??
s034	C++	21.400	0.033	6864	249	0.00	1.1	7.50	??
s042	C++	0.033	0.033	22680	243	100.00	100.0	11.90	??
s051	C++	0.150	0.033	3448	221	100.00	98.4	15.20	??
s090	C++	1.667	0.033	4152	511	98.48	100.0	19.60	??
s096	C++	0.917	0.017	5240	209	100.00	100.0	6.90	??
s017	Java	0.633	0.433	41952	509	100.00	10.2	48.90	??
s023	Java	2.633	0.650	89664	384	7.60	98.4	7.10	??
s037	Java	0.283	0.100	59088	364	100.00	10.2	13.00	??
s040	Java	0.317	0.283	56376	212	100.00	98.4	5.00	??
s043	Java	2.200	2.017	36136	164	98.85	90.9	8.70	??
s047	Java	6.467	0.117	54872	166	100.00	10.1	6.20	??
s050	Java	0.200	0.167	58024	186	100.00	10.2	4.80	??
s053	Java	0.267	0.100	52376	257	99.62	10.2	63.20	??
s054	Java	1.700	0.717	27088	324	100.00	10.2	13.80	??
s056	Java	0.350	0.067	22328	232	100.00	100.0	18.10	??
s057	Java	0.467	0.000	38104	434	100.00	10.2	17.80	??
s059	Java	4.150	0.050	40384	147	100.00	10.2	7.40	??
s060	Java	3.783	0.100	29432	281	98.85	96.3	27.60	??
s062	Java	16.800	0.067	38368	218	100.00	10.2	3.80	??
s063	Java	1.333	0.450	38672	155	100.00	100.0	7.30	??
s065	Java	1.467	0.117	49704	427	100.00	97.9	39.70	??
s068	Java	31.200	0.050	40584	107	100.00	10.2	15.10	??
s072	Java	30.100	0.067	52272	365	100.00	100.0	7.80	??
s081	Java	0.200	0.150	79544	614	100.00	10.2	26.60	??
s084	Java	0.150	0.133	65240	338	100.00	100.0	9.70	??
s087	Java	0.267	0.083	39896	322	100.00	100.0	19.70	??
s093	Java	37.100	0.050	41632	179	100.00	10.2	9.80	??
s099	Java	0.267	0.217	70696	228	100.00	98.4	3.80	??
s102	Java	0.167	0.150	51968	130	0.18	6.6	8.20	??
s149101	perl	0.267	0.183	17344	60	99.24	100.0	1.08	0-10%
s149102	perl	21.400	0.417	73440	62	0.00	0.0	1.67	10-25%
s149103	perl	0.083	0.067	25408	49	100.00	100.0	1.58	NA
s149105	perl	0.200	0.100	31536	97	100.00	100.0	3.17	0-10%
s149106	perl	0.117	0.033	17480	65	99.24	100.0	6.17	0-10%
s149107	perl	0.350	0.333	17232	108	100.00	100.0	2.50	0-10%
s149108	perl	0.483	0.433	73448	74	100.00	100.0	2.83	10-25%
s149109	perl	0.167	0.133	17312	141	100.00	100.0	3.67	25-40%

s149110	perl	0.200	0.133	17232	114	99.24	100.0	5.00	10-25%
s149111	perl	0.267	0.233	17224	80	100.00	100.0	1.00	10-25%
s149112	perl	0.250	0.233	17576	66	100.00	98.4	1.25	10-25%
s149113	perl	67.500	0.300	20320	121	100.00	0.0	7.00	60-75%
s149114	perl	0.333	0.233	21896	74	99.24	98.4	7.00	25-40%
s149201	python	0.650	0.317	22608	82	99.24	100.0	2.00	10-25%
s149202	python	1.583	1.367	17784	61	99.24	98.4	1.58	NA
s149203	python	0.183	0.167	13664	79	100.00	100.0	1.77	0-10%
s149204	python	0.117	0.067	13632	60	100.00	100.0	2.43	0-10%
s149205	python	0.083	0.067	17336	78	100.00	10.2	1.50	0-10%
s149206	python	0.117	0.067	17320	42	100.00	100.0	5.50	40-60%
s149207	python	0.133	0.067	15312	114	100.00	100.0	2.83	0-10%
s149208	python	0.450	0.367	16024	94	99.24	98.4	4.20	25-40%
s149209	python	72.300	0.500	14632	119	100.00	0.0	4.83	10-25%
s149210	python	0.250	0.200	17480	225	100.00	10.2	4.50	0-10%
s149211	python	1.483	1.150	91120	82	98.48	10.2	2.00	40-60%
s149212	python	0.617	0.467	14048	84	99.24	100.0	3.00	0-10%
s149213	python	0.733	0.533	14000	52	99.24	100.0	4.32	60-75%
s149301	rexx	0.817	0.300	8968	53	100.00	98.4	0.93	10-25%
s149302	rexx	25.400	0.900	21152	203	44.75	46.6	8.00	10-25%
s149303	rexx	21.000	0.950	21144	191	99.62	98.9	10.00	10-25%
s149304	rexx	1.000	0.433	9048	53	100.00	100.0	3.00	10-25%
s149401	tcl	0.650	0.567	32400	62	100.00	100.0	0.83	10-25%
s149402	tcl	0.567	0.433	29272	78	100.00	100.0	4.00	0-10%
s149403	tcl	0.617	0.517	28880	144	100.00	100.0	4.50	10-25%
s149405	tcl	0.967	0.667	44536	98	100.00	100.0	3.75	25-40%
s149406	tcl	0.650	0.583	26352	105	100.00	100.0	1.38	10-25%
s149407	tcl	0.783	0.467	17672	44	100.00	100.0	0.60	0-10%
s149408	tcl	202.800	0.633	48840	173	100.00	100.0	7.00	0-10%
s149409	tcl	0.683	0.567	23192	70	100.00	100.0	8.00	0-10%
s149410	tcl	29.400	1.433	20296	240	100.00	10.2	13.00	40-60%
s149411	tcl	0.467	0.367	21448	135	100.00	100.0	3.50	10-25%

The reliability values of 98% and higher for z1000rel occur due to some very subtle I/O issues when the programs are executed in the test harness (where output is to a pipe rather than to a terminal). These programs should be considered correct.

The reliability values of 98% and higher for m1000rel occur due to two very minor ambiguities in the task specification (see the Appendix of [20]). These programs should be considered correct.

Acknowledgements

I thank all participants who submitted a program for this study. Without you, the unusually broad coverage could not have been obtained.

References

- [1] Allan J. Albrecht and Jr. Gaffney, John E. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, November 1983.
- [2] Charles A. Behrens. Measuring the productivity of computer systems development activities with function points. *IEEE Transactions on Software Engineering*, SE-9(6):648–652, November 1983.
- [3] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.
- [4] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lühr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.
- [5] Software Productivity Research Capers Jones. Programming languages table, version 7. <http://www.spr.com/library/0langtbl.htm>, 1996 (as of Feb. 2000).
- [6] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [7] Alireza Ebrahimi. Novice programmer errors: Language constructs and plan composition. *Intl. J. of Human-Computer Studies*, 41:457–480, 1994.
- [8] Bradley Efron and Robert Tibshirani. *An introduction to the Bootstrap*. Monographs on statistics and applied probability 57. Chapman and Hall, New York, London, 1993.
- [9] Les Hatton. Does OO sync with how we think? *IEEE Software*, 15(3):46–54, March 1998.
- [10] Jarkko Hietaniemi. perlhist: The Perl history records. Unix Manual Page, March 1999.
- [11] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. awk vs. ... an experiment in software prototyping productivity. Technical report, Yale University, Dept. of CS, New Haven, CT, July 1994.
- [12] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI series in Software Engineering. Addison-Wesley, Reading, MA, 1995.
- [13] Robert Klepper and Douglas Bock. Third and fourth generation language productivity differences. *Communications of the ACM*, 38(9):69–79, September 1995.
- [14] Jürgen Koenemann-Belliveau, Thomas G. Mohrer, and Scott P. Robertson, editors. *Empirical Studies of Programmers: Fourth Workshop*, New Brunswick, NJ, December 1991. Ablex Publishing Corp.

- [15] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. *J. of Object-Oriented Programming*, 1992.
- [16] Mark Lutz. *Programming Python*. O'Reilly and Associates, March 2001.
- [17] George A. Miller. The magic number seven, plus or minus two. *The Psychological Review*, 63(2):81–97, March 1956.
- [18] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
- [19] Michael Philippsen. Imperative concurrent object-oriented languages. Technical Report TR-95/50, International Computer Science Institute, University of California, Berkeley, CA, August 1995.
- [20] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany, March 2000. ftp.ira.uka.de.
- [21] Lutz Prechelt and Barbara Unger. A controlled experiment on the effects of PSP training: Detailed description and evaluation. Technical Report 1/1999, Fakultät für Informatik, Universität Karlsruhe, Germany, March 1999. ftp.ira.uka.de.
- [22] D.A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6:28–36, September 1989.
- [23] Richard M. Shiffrin and Robert M. Nosofsky. Seven plus or minus two: A commentary on capacity limitations. *Psychological Review*, 101(2):357–361, 1994.
- [24] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly and Associates, 2000.
- [25] C.E. Walston and C.P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.