# An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/608-7343
http://wwwipd.ira.uka.de/EIR/

Technical Report 2000-!!!

February 29, 2000



**Abstract**

80 implementations of the same set of requirements, created by 74 different programmers in various languages, 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 language comparisons

When it comes to the pros and cons of various programming languages, programmers and computer scientists alike are usually highly opinionated. In contrast, only relatively little high-quality objective information is available about the relative merits of different languages. The scientific and engineering literature provides many comparisons of programming languages — in different ways and with different restrictions:

Some are purely theoretical discussions of certain language constructs. The many examples range from Dijkstras famous letter "Go To statement considered harmful" [6] to comprehensive surveys of many languages [4, 16]. These are non-quantitative and usually partly speculative. Some such works are more or less pure opinion-pieces.

Some are benchmarks comparing a single implementation of a certain program in either language for expressiveness or resource consumption, etc.; an example is [10]. 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, 14], often focussing on either a single language construct, e.g. [13, p.227], or a whole notational style, e.g. [13, p.121], [18].

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.

The present work provides some objective information comparing several languages, namely C, C++, Java, Perl, Python, Rexx, and Tcl. It has the following features:

- 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.

# 2 Origin of the programs

The programs analyzed in this report come from two different sources. The Java, C, and C++ programs were produced in the course of a controlled experiment, the others were produced under less well understood conditions and were submitted by Email.

The programming task was a program (called *phonecode*) that maps telephone numbers into strings of words according to a given dictionary and a fixed digit-to-character encoding. It will be described in Section 4.

### 2.1 Non-script group: C, C++, Java

All C, C++, and Java programs were produced in 1997/1998 during a controlled experiment comparing the behavior of programmers with and without previous PSP (Personal Software Process [11]) training. All of the subjects were Computer Science master students. They chose their programming language freely. 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 attempts at passing this test.

Detailed information about the subjects, the experimental procedure, etc. can be found in [17].

### 2.2 Script group: Perl, Python, Rexx, Tcl

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 4.

For brevity (and for brevity only), I will often refer to this set of languages (Perl, Python, Rexx, Tcl) as *script languages* and to the respective programs as *scripts*. The other three languages (C, C++, Java) will correspondingly be called *non-script languages*, the programs as *non-scripts*.

## 3 Validity: Are these programs comparable?

The rather different conditions under which these programs were produced raise an important question: Is it 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? 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.

### 3.1 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 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 (see [17]). 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. Let's 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 still fairly young.

In 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.

## 3.2 Work time reporting accuracy

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 "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)."*

However, there is evidence (described in Section 5.7) 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.

## 3.3 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. 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 limit[1] not present in the submission procedure of the script group.

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 costed 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 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. However, 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.

## 3.4 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 4.3), resulting in an incorrect program. Since only few of these programmers were 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. In most cases, correcting the mistake in a faulty program was trivial.

[1] 64 MB total, 30 seconds maximum per output plus 5 minutes for loading on a 143 MHz Sparc Ultra I.

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 Section 4 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 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.

### 3.5 Other minor problems

The non-script programmers had a further slight disadvantage, because they were forced to implement on a particular computer. 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 disadvantage because the platform on which they will be evaluated (a Rexx implementation called "Regina") is not the platform on which they were originally developed. 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 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, will just as clearly be valid.

## 4 The programming problem: phonecode

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 [17]. 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 different with respect to the submission procedure etc.

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

## 4.1 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*
- *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 Phonecode 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 phonecode 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 "phonecode" 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.*

## 4.2 Task requirements description

*Consider 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* |

*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 (`"`). 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 youself 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.*

## 4.3 The hint

The “hint” referred to in the procedure description shown in Section 4.1 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!*

# 5 Results

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).

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

## 5.1 Plots and statistical methods

The plots and statistical methods used in the evaluation are described in some detail in [17]; we only give a short description here.

The main evaluation tool will be the multiple boxplot display, see for example Figure 2 on page 14. 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.

Most interesting observations can easily be made directly in these plots. For quantifying some of them, I will also sometimes provide the results of statistical 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 accidental and no difference (or a difference in the opposite direction) between the underlying populations does indeed exist.

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 |

At several 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 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.

## 5.2 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. The results for C and Rexx will be based on only 5 or 4 programs, respectively, and are thus rather coarse estimates of reality, but for all of the other languages there are 10 or more programs, which is a broad-enough base for reasonably precise results. Note that the sample covers 80 different programs but only 74 different authors.

## 5.3 Run time

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

### 5.3.1 Total: z1000 data set

The global overview of the program run times on the z1000 input file is shown in Figure 1. We see that for all languages a few very slow programs exist, but except for C++, Java and Rexx, at least three quarters of the programs run in less than one minute.

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 |

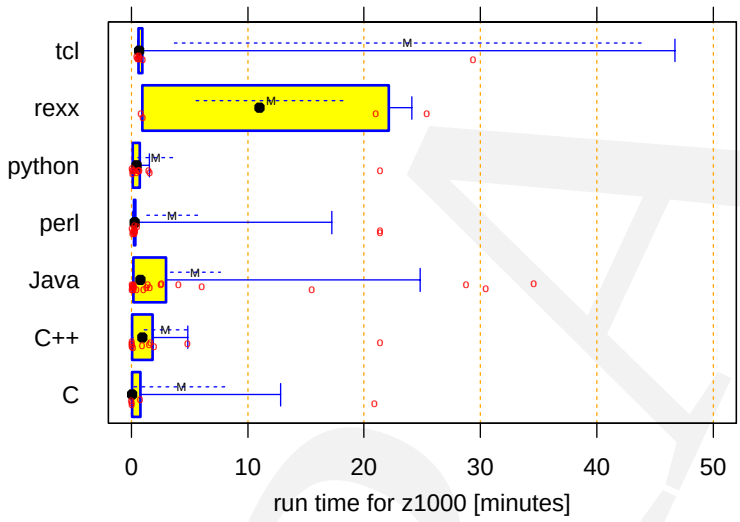


Figure 1: Program run time on the z1000 data set. Several programs were timed out with no output after about 20 minutes. One Tcl program took 202 minutes. The bad/good ratios range from 1.5 for Tcl up to 27 for C++.

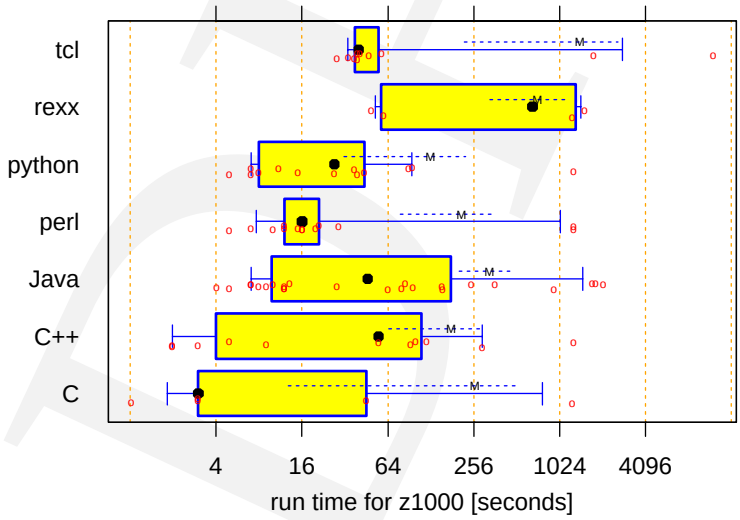


Figure 2: Program run time on the z1000 data set. Equivalent to Figure 1, except that the axis is logarithmic and indicates seconds instead of minutes.

In order to see and discriminate all of the data points at once, we can use a logarithmic plot as shown in Figure 2. 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++ ($p = 0.30$).
  (Attention: 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 ($p = 0.018$), and Tcl ($p = 0.041$). They even tend to be smaller than those of Java ($p = 0.19$).
- The median run times of Perl are also smaller than those of Rexx ($p = 0.014$), and Tcl ($p = 0.002$).
- As mentioned above, although it doesn't look like that, the median of C++ tends to be smaller than that of Java ($p = 0.18$).
- 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 test for differences in interquartile range of logarithmic run times (i.e. differences in box width in Figure 2) between Perl and Python indicates $p = 0.17$.

Remember not to over-interpret the plots for C and Rexx, because they have only few points.

If we aggregate the languages into only three groups, as shown in Figure 3, 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.

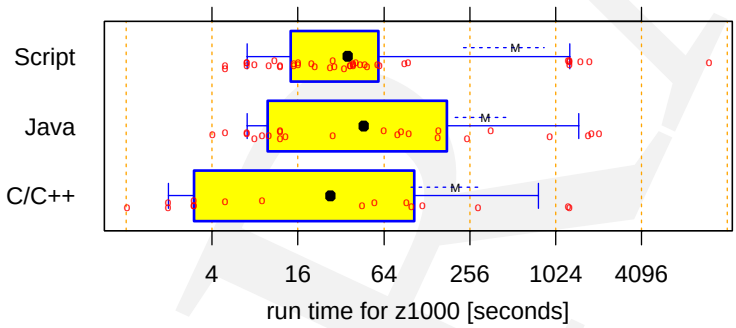


Figure 3: Program run time on the z1000 data set on logarithmic axis; just like Figure 2, except that the samples for several languages are aggregated into larger groups. The bad/good ratios are 4.1 for script, 18 for Java and 35 for C/C++.

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 1.15 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.

### 5.3.2 Initialization phase only: z0 data set

We can repeat the same analysis for the case where the program only reads and stores the dictionary — most programs also do some preprocessing in this phase to accelerate further execution. Figure 4 shows the corresponding run time.

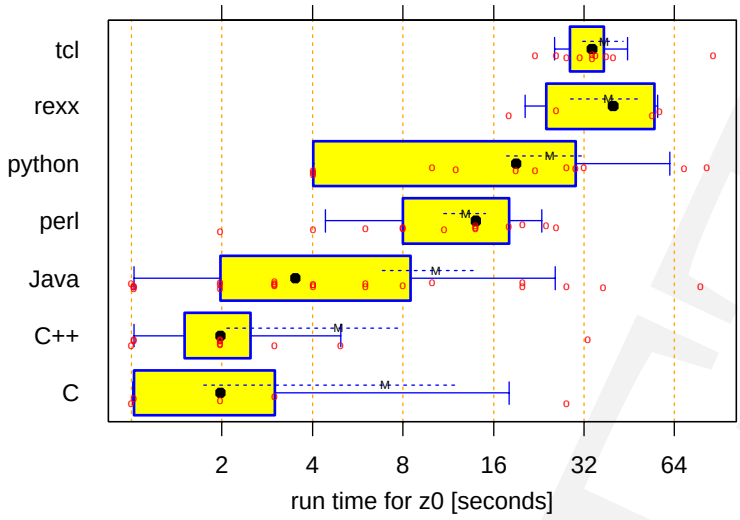


Figure 4: Program run time for loading and preprocessing the dictionary only (z0 data set). Note the logarithmic axis. The bad/good ratios range from 1.3 for Tcl up to 7.5 for Python.

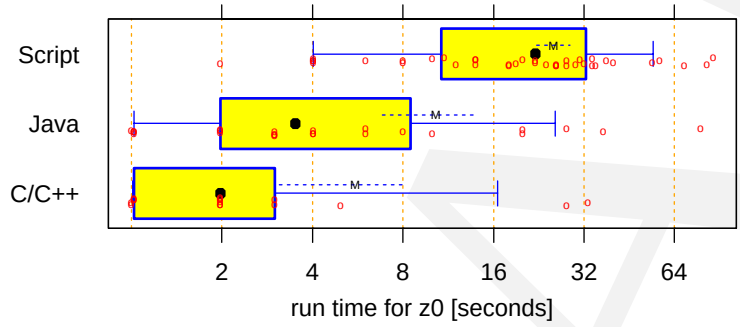


Figure 5: Program run time for loading and preprocessing the dictionary only (z0 data set); just like Figure 2, except that the samples for several languages are aggregated into larger groups. The bad/good ratio is about 3 for Script and C/C++ and 4.3 for Java.

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 and Java is faster.

For the aggregate grouping (Figure 5) 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.

### 5.3.3 Search phase only

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 6 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 ($p = 0.091$), Perl ($p = 0.009$), 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 ($p = 0.005$), and Tcl ($p = 0.091$). They even tend to be smaller than those of Java ($p = 0.11$).

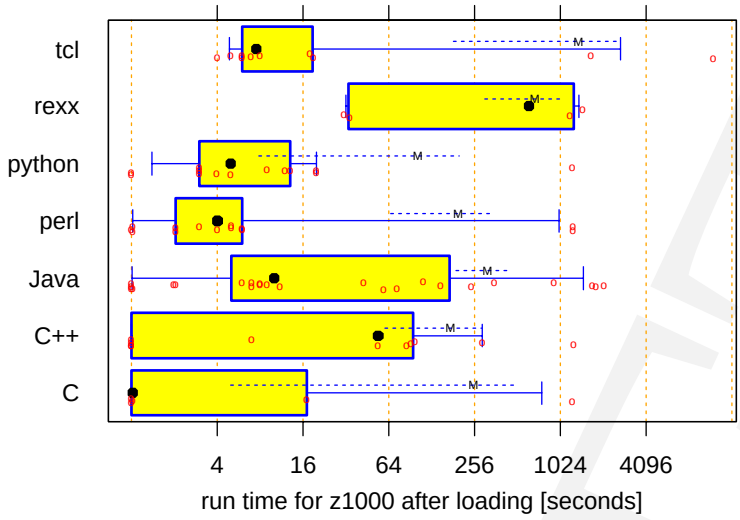


Figure 6: Program run time for the search phase only. Computed as time for z1000 data set minus time for z0 data set. Note the logarithmic axis. The bad/good ratios range from 2.9 for Perl up to over 50 for C++ (in fact 95, but unreliable due to the imprecise lower bound).

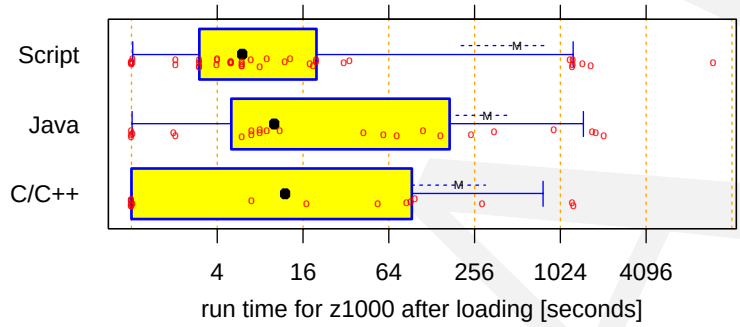


Figure 7: Program run time for the search phase only. Computed as time for z1000 data set minus time for z0 data set. This is just like Figure 6, except that the samples for several languages are aggregated into larger groups. The bad/good ratio is about 7 for Script, 34 for Java, and over 50 for C/C++ (in fact 95, but unreliable due to the estimated lower bound).

- The median run times of Perl are smaller than those of Rexx ($p = 0.013$), Tcl ($p = 0.010$), and even Java ($p = 0.035$).
- Although it doesn't look like that, the median of C++ is not significantly different from any of the others (two-sided tests yield $0.26 < p < 0.91$).

The aggregated comparison in Figure 7 indicates no significant differences between any of the groups, neither for the pairs of medians ($p \geq 0.13$) nor for the pairs of means ($p \geq 0.17$).

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 2.3 and smaller than that of C/C++ by a factor of at least 3.3.

## 5.4 Memory consumption

How much memory is required by the programs?

Figure 8 shows the total process size at the end of the program execution for the z1000 input file.

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 the clearly the Java programs.

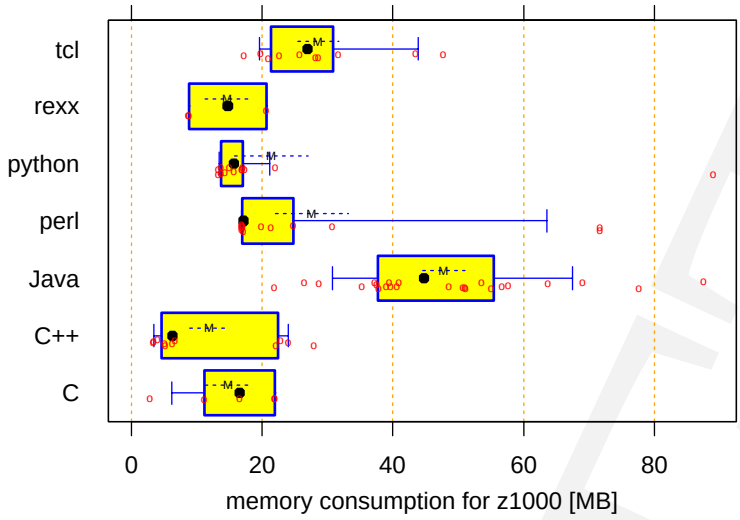


Figure 8: Amount of memory required by the program, including the interpreter or run time system, the program itself, and all static and dynamic data structures. The bad/good ratios range from 1.2 for Python up to 4.9 for C++.

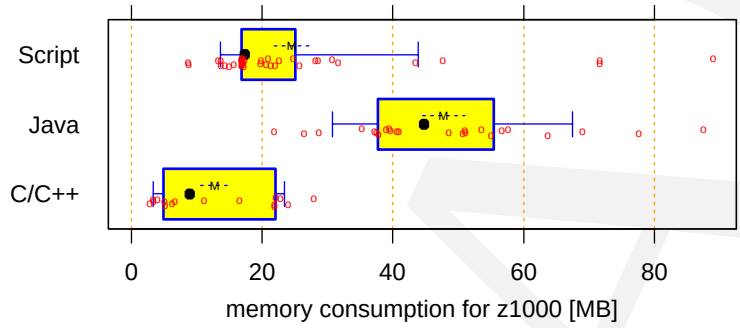


Figure 9: Like Figure 8, except that the languages are aggregated into groups. The bad/good ratios are 1.5 for Script and for Java and 4.5 for C/C++.

- Except for Tcl, only few of the scripts consume more memory than the worse half of the C and C++ programs.
- Tcl scripts require more memory than other scripts.
- For Python and Perl, 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.
- On the average (see Figure 9) 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.

I conclude that 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 cannot beat a parsimonious C or C++ program.

## 5.5 Program length and amount of commenting

How long are the programs?
How much comments do they contain?

Figure 10 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).

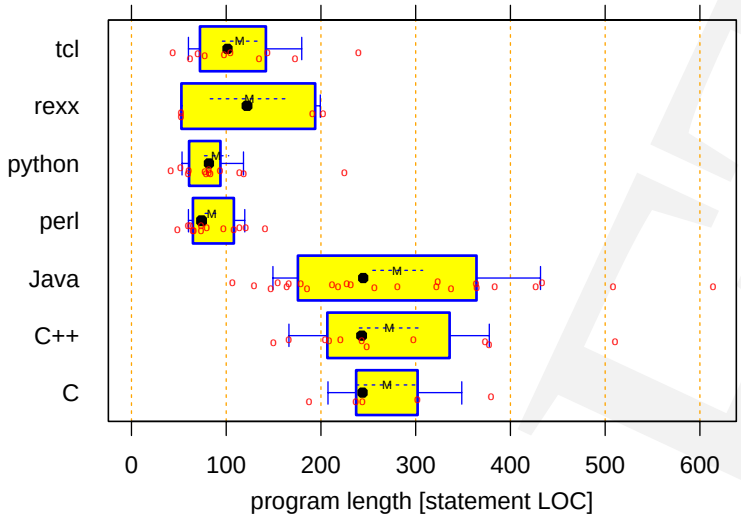


Figure 10: Program length, measured in number of non-comment source lines of code. The bad/good ratios range from 1.3 for C up to 2.1 for Java and 3.7 for Rexx.

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.

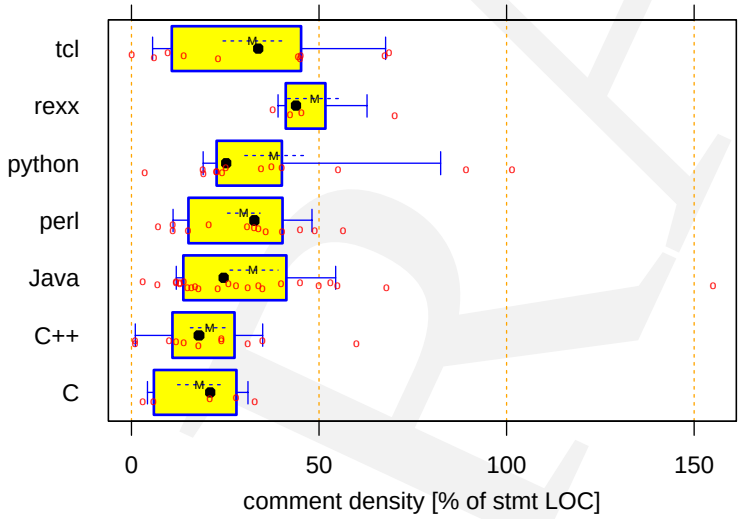


Figure 11: Percentage of comment lines plus commented statement lines, relative to the number of statement lines. The bad/good ratios range from 1.3 for Rexx up to 4.2 for Tcl.!!!

At the same time, scripts tend to contain a significantly higher density of comments (Figure 11), with the non-scripts averaging a median of 22% as many comment lines or commented lines as statement lines and the scripts averaging 34% ($p = 0.020$).

## 5.6 Program reliability

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

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 12.

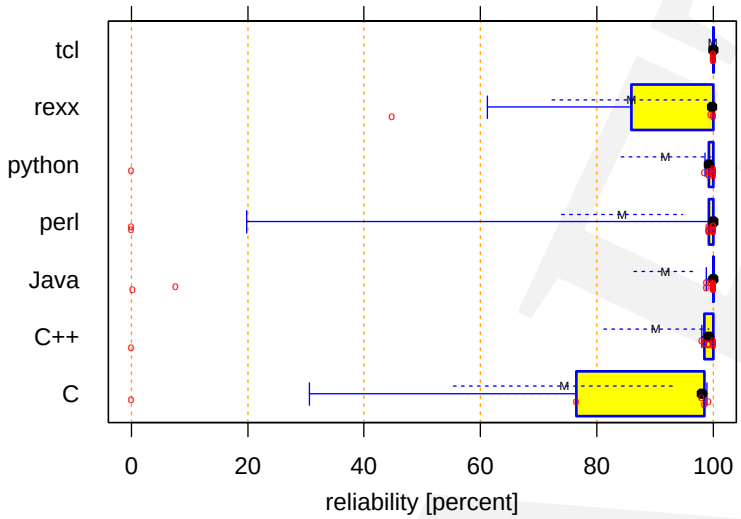


Figure 12: Program output reliability in percent for the z1000 input file.

5 programs (1 C, 1 C++, 2 Perl, 1 Python) 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.

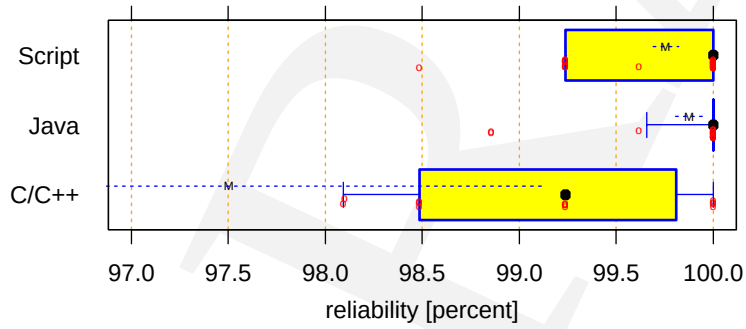


Figure 13: Program output reliability in percent (except for those programs with reliability below 50 percent), with languages aggregated into groups.

If we ignore the above-mentioned 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 10% of the script programs; Figure 13) by language group, we find that the Java programs tend to be the most reliable, and the script programs tend to be better than the C/C++ programs. These differences, however, 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.

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 Figure 14.

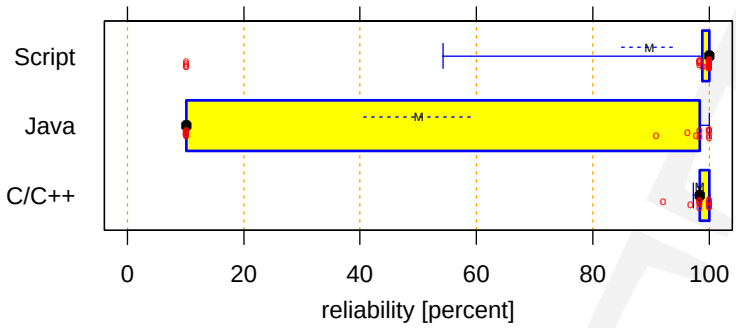


Figure 14: Program output reliability for the m1000 input file in percent (except for those programs whose z1000 reliability was below 50 percent), with languages aggregated into groups.

Most programs cope with this situation well, but half of the Java programs and four 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. Note that the huge size of the box for the Java data in Figure 14 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%.

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

## 5.7 Work time

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

### 5.7.1 Data

Figures 15 and 16 show the total work time as reported by the script programmers and measured for the non-script programmers.

As we see, scripts take less than half as long as non-scripts. Note that the meaning of the measurements is not exactly the same: First, non-script times always include the time required for reading the requirements, whereas many of the script participants apparently did not count that time. Second, 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 about the program design in the meantime (see the quotes in Section 3 above). Third, some of the script participants estimated (rather than measured) their work time. Finally, we do not know whether all non-script participants were honest in their work time reporting.

### 5.7.2 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

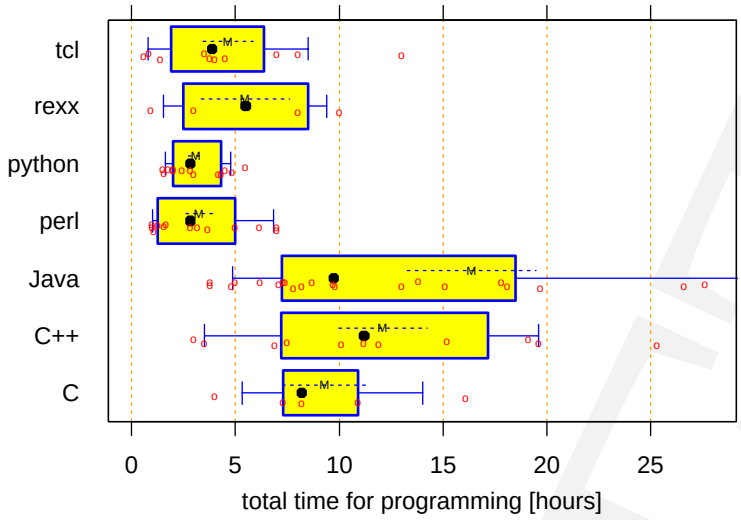


Figure 15: Total working time for realizing the program. Script group: times as measured and reported by the programmers. Non-script group: times as measured by the experimenter. The bad/good ratios range from 1.5 for C up to 4 for Perl. Three Java work times at 40, 49, and 63 hours are not shown.

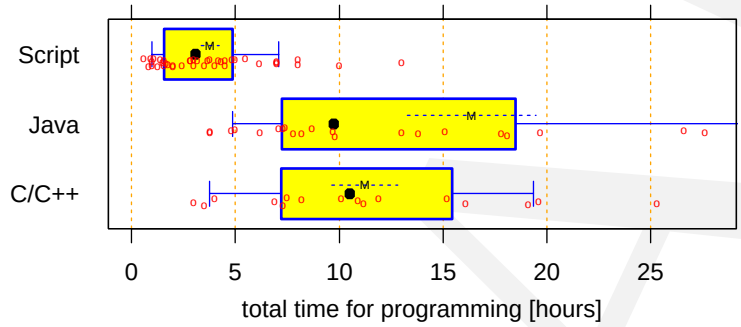


Figure 16: Like Figure 15, except that the languages are aggregated into larger groups. The bad/good ratio is 3.1 for Script, 2.6 for Java, and 2.1 for C/C++.

(rather than less capable ones) in the script group compared to the non-script group if programmer capabilities are different on average.

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 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 [15, 19].

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 [20].

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 |

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 for each language LOC/FP (the number of lines typically required to implement one function point) and the 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·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 to function points per time have also been published, e.g. [12]. Table 3 provides the relevant excerpt from the language table and Figure 17 relates this data to the actual productivity observed in the present study.

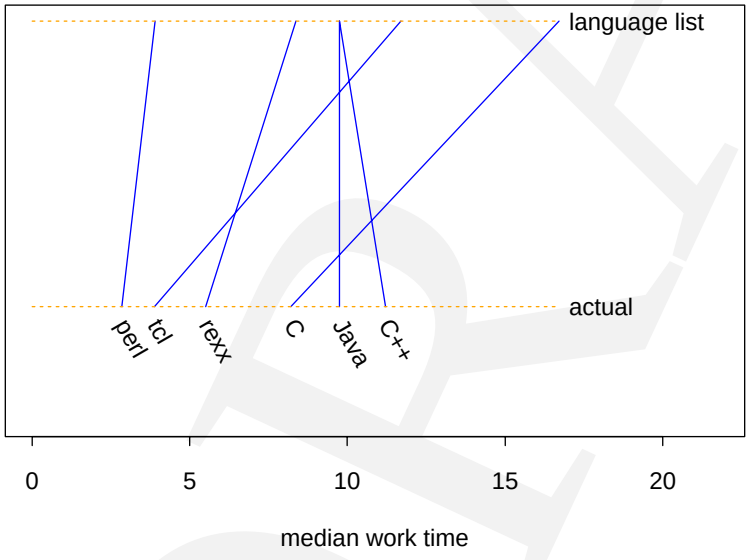


Figure 17: Actual median work times of each language compared to those we would expect from the relative productivity as given in Capers Jones' programming language list [5], normalized such that the Java work times are exactly as predicted. We find that the language list underestimates the productivity of C and Tcl for this problem. For the phonecode problem, C is almost as well-suited as C++, 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 fairly 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 based on this rule is plotted in Figure 18. Judging from the productivity range of Java, all data points except maybe for the top three of Tcl and the top two of Perl are quite believable. In particular, except for Python, all medians are in the range 22 to 29. Hence, the LOC productivity plot lends a lot of credibility to the reported times: Only five 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 $0.07 \leq p \leq 0.10$.

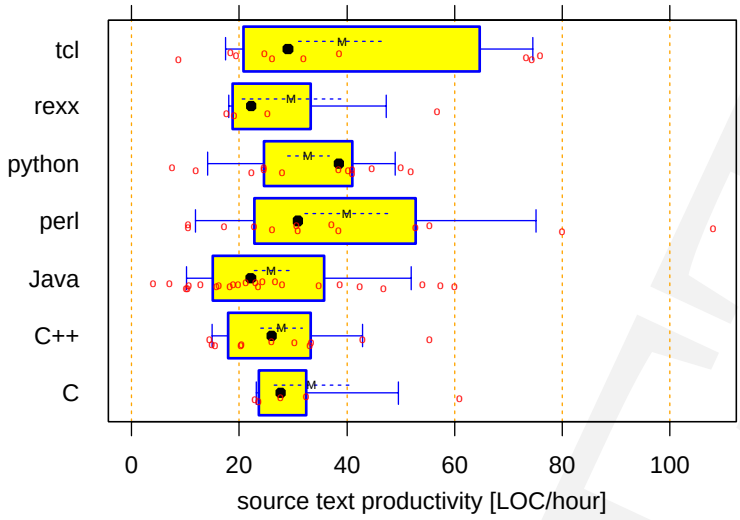


Figure 18: Source text productivity in non-comment lines of code per total work hour. The bad/good ratios range from 1.4 for C up to 3.1 for Tcl.

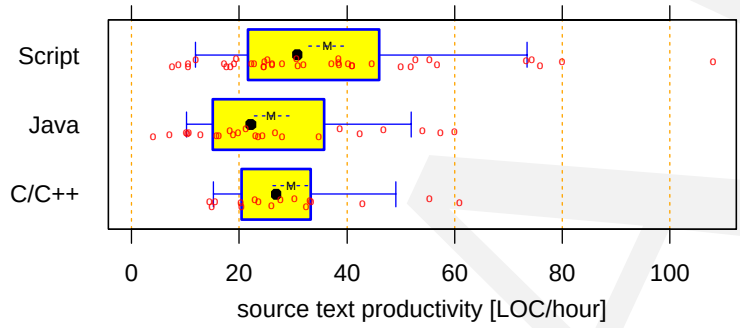


Figure 19: Like Figure 18, except that the languages are aggregated into groups. The bad/good ratio is 2.1 for Script, 2.4 for Java, and 1.6 for C/C++.

Even in the aggregated view (Figure 19) with its much larger groups, the difference between C/C++ and scripts is not significant ($p = 0.20$), only the difference between Java and scripts is ($p = 0.027$), the difference being at least 6.5 LOC/hour (with 80% confidence).

### 5.7.3 Conclusion

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.

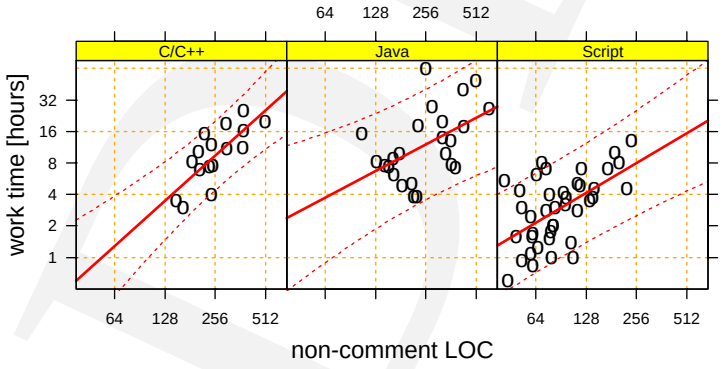


Figure 20: The same data as in Figure 19, except that the program lengths and work times are shown separately. The lines are a standard least squares regression line and its 90% prediction interval. Note the logarithmic axes.

Figure 20 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 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.

## 5.8 Program structure

If one considers the designs chosen by the authors of the programs in the various languages, there is a striking difference.

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.

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 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.

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.

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 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.

## 5.9 Programmer self-rating

The non-script programmers were asked several questions about their previous programming experience, as described in detail in [17]. Unfortunately, none of these questions had much predictive value for any aspect of programmer performance in the experiment, so I will not delve into this data at all.

The script programmers were asked but a single 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 21; the rank correlation is -0.327) and also with shorter work time for producing the program (Figure 22; the rank correlation is -0.320).

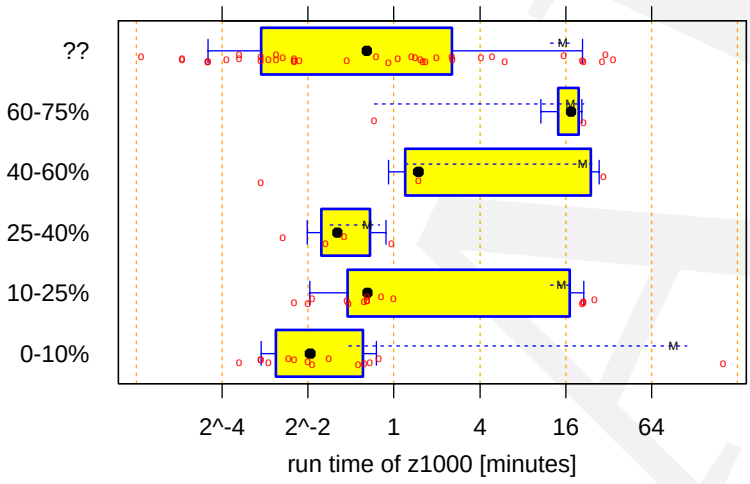


Figure 21: Relationship between self-rating and program efficiency: higher self-rating is correlated with faster programs. The uppermost boxplot represents all non-script programs. Note the logarithmic axis.

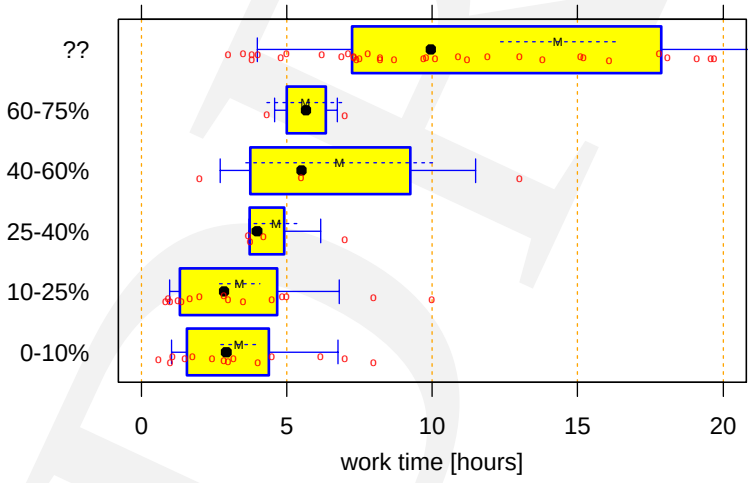


Figure 22: Relationship between self-rating and working time for writing the program: higher self-rating is correlated with shorter work time. The uppermost boxplot represents all non-script programs. Note the logarithmic axis.

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

# 6 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 significant 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.
- 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) is about as large or even larger than the variability due to different languages on average.

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. However, it must be emphasized that the results are valid for the phonecode problem only, generalizing to different application domains 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 things:

- As of JDK 1.2.1 (and on the Solaris platform), the memory overhead of Java is still huge compared to C or C++, but the run time efficiency has become quite acceptable.
- The often so-called "scripting languages" Perl, Python, Rexx, and Tcl can be reasonable 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.
- 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.

# 7 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 (stmtL), output reliability for z1000 input file in percent (z1000rel), output reliability for m1000 input file in percent (m1000rel), total subject work time in hours (whours), 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 | stmtL | z1000rel | m1000rel | whours | 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 | 0.400 | 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 | 1.00 | 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 | 21.400 | 0.300 | 20320 | 121 | 0.00 | 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 | 21.400 | 0.500 | 14632 | 119 | 0.00 | 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% |

## 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 Löhr. 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*, 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] 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.

[11] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI series in Software Engineering. Addison Wesley, Reading, MA, 1995.

[12] Robert Klepper and Douglas Bock. Third and fourth generation language productivity differences. *Communications of the ACM*, 38(9):69–79, September 1995.

[13] 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.

[14] 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.

[15] George A. Miller. The magic number seven, plus or minus two. *The Psychological Review*, 63(2):81–97, March 1956.

[16] Michael Philippsen. Imperative concurrent object-oriented languages. Technical Report TR-95/50, International Computer Science Institute, University of California, Berkeley, CA, August 1995.

[17] 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.

[18] D.A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6:28–36, September 1989.

[19] Richard M. Shiffrin and Robert M. Nosofsky. Seven plus or minus two: A commentary on capacity limiations. *Psychological Review*, 101(2):357–361, 1994.

[20] C.E. Walston and C.P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.