

**Quantitative Linguistic
Computing with Perl**

by

Fan Fengxiang

Deng Yaochen

2010

RAM-Verlag

Studies in quantitative linguistics

Editors

Fengxiang Fan (fanfengxiang@yahoo.com)
Emmerich Kelih (emmerich.kelih@uni-graz.at)
Reinhard Köhler (koehler@uni-trier.de)
Ján Mačutek (jmacutek@yahoo.com)
Eric S. Wheeler (wheeler@ericwheeler.ca)

1. U. Strauss, F. Fan, G. Altmann, *Problems in quantitative linguistics 1*. 2008, VIII + 134 pp.
2. V. Altmann, G. Altmann, *Anleitung zu quantitativen Textanalysen. Methoden und Anwendungen*. 2008, IV+193 pp.
3. I.-I. Popescu, J. Mačutek, G. Altmann, *Aspects of word frequencies*. 2009, IV +198 pp.
4. R. Köhler, G. Altmann, *Problems in quantitative linguistics 2*. 2009, VII + 142 pp.
5. R. Köhler (ed.), *Issues in Quantitative Linguistics*. 2009, VI + 205 pp.
6. A. Tuzzi, I.-I. Popescu, G. Altmann, *Quantitative aspects of Italian texts*. 2010, IV+161 pp.
7. F. Fan, Y. Deng, *Quantitative Linguistic Computing with Perl*. 2010, VIII+201 pp

ISBN: 978-3-942303-01-9

© Copyright 2010 by RAM-Verlag, D-58515 Lüdenscheid

RAM-Verlag
Stüttinghauser Ringstr. 44
D-58515 Lüdenscheid
RAM-Verlag@t-online.de
<http://ram-verlag.de>

Preface

Empirical research in linguistics, in particular in quantitative linguistics, relies to a high degree on the acquisition of large amounts of appropriate data and, as a matter of course, on sometimes intricate computation. The last decades with the advent of faster and faster electronic machinery and at the same time growing storage capacities at falling prices contributed, together with advances in linguistic theory and analytic methods, to the availability of suitable linguistic material for all kinds of investigations on all levels of analysis.

Ideally, a researcher in quantitative linguistics has enough programming knowledge to acquire the data needed for his/her specific study. This is, however, not always the case. If professional programmers can be asked for help, most problems may be overcome; however, also this way is not always possible. And sometimes, it may be more awkward to explain a task than to perform it.

The selection of the appropriate programming language should not be conducted by taste or familiarity (as it is, unfortunately, very often even among programmers); instead, at least the following criteria should be taken into account:

1. Quality of the language. There is a number of quality-related properties of a programming language such as ability of preventing programming mistakes, readability of the code, changeability, testability, learnability. Unfortunately, these and other properties are not independent of each other; some of them compete (e.g. efficiency is always a competitor of most other properties) whereas others co-operate (e.g. readability advances most of the others). A programmer has to decide on priorities of the quality properties with respect to the individual task and application.
2. Nature of the problem. Every programming language has advantages and disadvantages also with respect to the task to be performed. One of the criteria often cited is the old distinction between low-level (close to the basic processor instructions and to the memory organisation of the computer) and high-level languages (with concepts close to the problems or algorithms). However, matters of efficiency etc. do not play a role any more (at least in the overwhelming majority of applications) since the compilers and their optimisers produce better code than most human programmers would be able to do. But there are still concepts and tasks that can be expressed in one language better than in another one, e.g. only few programmers would prefer a scripting language for the programming a data base.
3. Size and complexity of the problem. The larger a problem and the higher its complexity the more relevant become the quality properties of the language. If, e.g. several persons work on the same project, readability of the code is of fundamental importance but even if a single programmer

II

does all the coding of a complex problem he/she will encounter problems with his own code after some time if the programming language allows code in a less readable form. In any case, corrections, changes, and maintenance of a program depend crucially on some of the quality properties.

4. Security aspects. This is a simple matter: If the application you write is supposed to run in an environment that is accessible to potential attacks (e.g., the Internet or a computer network or if other users have access to the computer) and if the data your program works with should be protected, then a special focus should be put on security properties of the programming language. Scripting languages, e.g., are known to be frail, as a rule. You should make sure that the language you use has at least protection mechanisms you can switch on. Surprisingly, this aspect is very often neglected – even by institutions such as banks (Internet banking).
5. Reliability of the compiler/interpreter/libraries. Many popular languages are not *defined*. Instead, ‘reference implementations’ are offered. However, to be sure how a language element works you would have to try it out in every possible form of use and combination with other elements – an unrealistic idea. Another aspect is even more significant in practice: Some languages, among them some of the currently most popular ones, are subject to substantial changes every now and then. The user of such a language is witness and victim of a ripening process (or mere experimentation): If your program will work with the next version of the language, is more or less a matter of chance. You should consider how much harm such a situation would do to your project if you decide to use a language that is not defined.
6. Frequency of application. It matters whether your program is an ad-hoc solution and will be used just once or a few times to evaluate some data and then will be discarded or whether it is meant to be useful for a longer time and may be changed and adapted for varying conditions. In the first case, not so much value is to be set on quality properties of the language; immediate availability of a practical solution may then come into foreground. In the latter case, however, readability, changeability and other properties play a bigger role.
7. Intended users of the software product. Similarly, if you alone will use a program, some disadvantages such as missing robustness or a bad user interface would not constitute a serious problem as you will exactly know which behaviour you have to expect and how to circumvent inelegance or even mistakes. If, on the other hand, an unknown number of unknown persons will use it you should base your product on reliable tools, among them the language you formulate your solution in.

The main problem, however, quantitative linguists will have to face – independent of how they are inclined to weight the criteria discussed above – is probably

that they fail to have an overview about programming languages and their pro's and con's. Whenever a programming layman is asked for advice the probability is high that the answer will depend on personal taste and familiarity with a language and possibly on its current popularity. You should, at least, know what criteria to base your decision on; with an idea of your priorities at hand and after discussing them with a programming expert, you can increase the chance to obtain a good hint.

Perl belongs to the so-called scripting languages. To run a program written in Perl you need the Perl interpreter; it has to be installed on your computer before the program can be executed. The reason is that such a program is interpreted, line by line, each time it is started as opposed to compiled programs which can run without any interpreter. There are, again, pro's and con's of either solution. Scripting languages have, e.g. the advantage that a program can change its own logic and easily adapt its data structures while it runs, to an extent which is impossible with compiled programs – a comfortable but also potentially dangerous facility.

Linguists fancy, in particular, the powerful language elements of Perl, which enable a programmer to write powerful programs in very short time. This property is especially useful for string and text manipulation and analysis because many ready-made tools for string handling are 'innate' to the language. Advanced programmers will find it even more useful for Internet programming. A clear disadvantage is the not so readable program text which makes finding and correcting of mistakes sometimes awkward in long and complex programs. Therefore, careful formulations and exhaustive comments within the program code are strongly recommended. If these caveats are taken into account Perl can be used with much success with little effort – and make a lot of fun.

Reinhard Köhler

Table of Contents

Preface	I
1 Introduction	1
1.1 Quantitative linguistics and Perl.....	1
1.2 Characteristics of this book and its intended readers	2
1.3 Downloading and installation.....	3
1.4 Program editor	7
1.5 Conventions used in this book.....	8
2 Perl variables and operators	11
2.1 Perl variables	11
2.2 Value assignment to variables	11
2.3 Perl numeric operators and functions	13
2.3.1 Math operators.....	13
2.3.2 Math functions.....	16
2.3.3 Numeric comparison operators	19
2.4 String operators and string comparison operators	21
2.4.1 String operators	21
2.4.2 String comparison operators.....	22
2.5 The logical operator.....	23
Exercises	24
3 Input and output	26
3.1 Input at the command line	26
3.1.1 The use of @ARGV	26
3.1.2 The use of STDIN	27
3.1.3 Command line file input.....	28
3.2 Inputting files inside a program.....	29
3.3 Some string manipulation functions	31
3.4 Applications.....	34
Exercises	40
4 Regular expressions: basic structure	41
4.1 Operators for regular expressions.....	41
4.1.1 =~ and m//	41
4.1.2 s///	42
4.1.3 tr///	45
4.2 Regular expression quantifiers and other operators	49
4.2.1 The general quantifiers and wild card.....	49
4.2.2 The greediness of the quantifiers * and +	52
4.2.3 The alternative operator, anchors and the escape operator	53
4.3 Applications.....	55
4.3.1 Text tokenizer	55
4.3.2 Computing syllabic word length	56
4.3.3 Removal of HTML codes in texts.....	57
Exercises	60

5 Regular expressions: advanced topics	61
5.1 Metacharacters for regular expressions	61
5.2 Special variables	63
5.3 Back referencing	65
5.4 Quantifying expressions	66
5.5 String manipulation functions and the <i>for</i> program control structure	68
5.6 Applications	71
5.6.1 Extraction of POS tags	71
5.6.2 Making concordance for a text.....	72
5.6.3 Extraction of lexical bundles from texts.	73
5.6.4 A Chinese tokenizer.....	75
Exercises	77
6 Arrays	79
6.1 Array creation	79
6.1.1 One dimensional arrays.....	79
6.1.2 Multi-dimensional arrays	82
6.1.3 Converting texts into arrays	84
6.2 Functions for array operations.....	85
6.2.1 Functions for array input and output.....	85
6.2.2 Array insertion, truncation and deletion.....	88
6.2.3 Sorting an array	89
6.2.4 The anonymous variable and the <i>join</i> , <i>map</i> and <i>grep</i> functions... 91	
6.3 Combining identical array elements and random sampling from an array	94
6.4 Applications	97
6.4.1 Selecting words from a wordlist	97
6.4.2 Turning a text into bigrams	98
6.4.3 Turning a text into a list of word types with frequencies.....	100
6.4.4 Computing sentence length distribution.....	101
Exercises	102
7 Hash tables	103
7.1 Hash input and output.....	103
7.1.1 Manual input and output	103
7.1.2 Hash input and output using arrays and functions	106
7.1.3 The use of <i>values()</i> , <i>each()</i> , <i>exist()</i> and <i>delete()</i>	110
7.2 Hash operations	112
7.2.1 Converting hash elements into an array	112
7.2.2 Combining two or more hashes together	113
7.2.3 Hash comparisons	115
7.2.4 Computing value frequencies.....	117
7.3 Applications	118
7.3.1 Computing per word entropy of English.....	118
7.3.2 Making a word frequency spectrum.....	119

7.3.3	Lemmatization.....	120
7.3.4	Lexical comparison between two texts	122
	Exercises	124
8	Subroutines and modules	126
8.1	Subroutines	126
8.1.1	The basic structure	126
8.1.2	Parameters of subroutines	127
8.1.3	The use of <i>return()</i> in subroutines.....	129
8.1.4	Localization of variables in subroutines	130
8.2	Modules	131
8.3	References.....	135
8.3.1	Making references	135
8.3.2	Dereferencing for scalar variables and references	136
8.3.3	Dereferencing for arrays	137
8.3.4	Dereferencing for hashes.....	139
8.4	Use of references in subroutines and modules	140
8.5	Applications	142
8.5.1	Computing arc length.....	143
8.5.2	A module for removing non-alphanumeric characters.....	144
8.5.3	A lexical comparison program	145
	Exercises	149
9	Directory and file management	150
9.1	Directory management	150
9.2	File management.....	151
9.3	Formatting output files	154
9.3.1	Outputting data in the original format.....	154
9.3.2	Arranging data in left-justified columns	155
9.3.3	Arranging data in right-justified columns	156
9.3.4	Arranging data in centre-justified columns	157
9.3.5	Formatting data that has line breaks.....	159
9.3.6	Producing page heading and paginating output files	161
9.4	Applications	164
9.4.1	A page-formatting program	164
9.4.2	Computing vocabulary growth and number of hapax legomena.....	166
9.4.3	A program for computing word range.....	171
	Exercises	175
Appendix	Model answers to the exercises	177
	Exercises of Chapter 2.....	177
	Exercises of Chapter 3.....	177
	Exercises of Chapter 4.....	181
	Exercises of Chapter 5.....	183
	Exercises of Chapter 6.....	186
	Exercises of Chapter 7.....	188

VIII

Exercises of Chapter 8.....	192
Exercises of Chapter 9.....	195
Index	202