МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2009
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2009
Proceedings of the Thirty Eighth Spring Conference of
the Union of Bulgarian Mathematicians
Borovetz, April 1–5, 2009

SAYING THE SAME IN MANY LANGUAGES

Boyko Bantchev

This article is a report of an experiment and a method proposal for assessment and comparison of programming languages. The method is empirical and is based on solving a problem many times, using many languages. Some observations, made in the course of conducting the experiment, are presented.

Exploring programming languages. Together with algorithms, programming languages are at the heart of computing. The interplay between the two generates the whole richness of the subject. An algorithm abstracts out an action, usually one that solves a particular problem. A language provides media for expressing algorithms, and, in turn, abstracts over them to create a view of computing.

Computing is a rich and diverse subject, and this is reflected by creating so many programming languages representing a range of paradigms, major and minor. That the differences between languages are so significant is revealing for how complex and elusive the very meaning of computing is.

To a searching mind, this great diversity brings some questions. How do we assess a language's fitness for solving a problem, or a class of problems? How do we compare languages in this respect? When are two programming languages really, and not only superficially, different? What are the possible (as opposed to currently known) major programming paradigms? Is there a set of basic notions, starting from which it would be possible to embrace and systematise all programming languages and their respective computational models?

These are all difficult questions, answering which requires collecting and putting into usable form information about many programming languages. One way of doing this is formalising some of the programming languages' aspects. Another, less formal one, is that of comparative descriptions where a set of representative topics is discussed in the context of a number of languages.

In this article, an even more ad hoc and less formal approach is reported: an empirical one. Its essence is taking a programming problem, solving it in many languages, in different ways, and making observations and comparisons along the way. I strongly believe that this kind of activity should be applied complementarily to formal study of programming languages, and that it is as important.

A method of language exploration. What sort of a problem do we need for this kind of language exploration? A deliberate choice was to take a sufficiently simple but non-trivial one. By 'sufficiently simple' I mean one that can be solved in a program small

enough to be followed by a person with no experience in the given programming language, provided that the program is accompanied by a reasonable amount of explanation (note that the latter is considered, in a sense, inseparable from the program as a solution to the problem). 'Non-trivial' means that solving it requires more than, say, a sequence of library-function calls or a single iterative/recursion process. Ideally, a solution should be forced to expose a significant portion of the expression style and repertoire that characterizes the language.

Put in other words, too simple programs tend to be uninformative of the language, but too large ones are not necessary, as merely increasing the volume adds no (useful) information.

A problem to test a wide range of languages against should be general enough, in the sense that solving it does not rely on, or exclusively favour a specific programming language or style.

Another requirement of the method of language assessment is to write complete programs and not just program fragments. This bounds one to address properly input/output and other issues of real programs. Doing otherwise conceals important facets of the language use and therefore cannot provide a ground for realistic conclusions.

Just as important for a realistic assessment is to have a precisely defined problem and solve it exactly as specified in every language under exploration. Without this, certain language advantages or disadvantages remain unexposed, and comparing solutions (and thus languages) on a common base is not possible.

Fair assessment and comparison also implies that in each solution the language is used only as defined by its standard. Not doing so may, in the case of a language with differing implementations or dialects, open possibilities for writing 'better' programs, but is not faithful to the language and therefore not acceptable.

The particular problem that has been chosen for this empirical study is writing a calculator for reverse-Polish (a.k.a. RPN) expressions. A program should read, parse and evaluate text lines. For each line that correctly represents an expression its value is printed. A more detailed statement of the problem is given in [1], together with all the program solutions explained. Currently, there are implementations in fifty programming languages. Several more are planned.

In fact, I have set before myself a more ambitious goal than just solving the problem in as many languages as possible. Part of the project is to provide terse and highly informative overviews of the implementation languages, as well as some others. This turns out to be a very labour-, time- and other resource-consuming task. It is partially fulfilled, with present coverage of about twenty languages, but there is much more to do. As the whole work is web-based, I also considered it important to provide references to relevant web resources for each language. Additionally, there is a general 'Links' section with an extensive collection of various references to web resources related to programming languages. All the text is available in [1].

It should be noted that remotely related efforts have been made by others, but the one presented here is more complete, and unique in several important respects. Unlike other collections of programs solving one problem, this one is a work of a single person, thus ensuring, on the one hand, a clear problem statement and implementation requirements, and consistency, completeness, and rigour across implementations, on the other. It also differs by providing explanations to the programs, language overviews and links to other

resources. The various parts of the project serve the general purpose of learning about, exploring and assessing programming languages. For the first time an attempt is made to turn this kind of activity into a method of language exploration, rather than simply a set of illustration programs.

Observations. The programming problem being considered is simple yet not trivial in several respects. For example, solving it implies two repetitive processes – for reading line by line and for processing an individual line – one nested within the other, and both of unlimited length. For properly ending the outer process, the language should provide means for programmatically detecting the end of input, and it turns out that not all languages do it. Since each input line is itself of unlimited length, either text of arbitrary size must be read in the program, or processing a line should be properly interleaved with reading it in pieces. Reading in pieces, in turn, implies detecting the end of an input line, and this too is challenging or impossible in some languages.

Part of solving the problem requires some form of text processing, in particular – parsing a string into meaningful items, possibly building a suitable structure along the way. Text is ubiquitous for representing data, thus the problem domain is typical enough and a good one to testing languages against.

According to the algorithmic method chosen for solving the problem, there might be the need for a simple data structure, for example a stack. Since an input line is not limited in length, that structure must be expandable to an arbitrary size.

It turns out that computing an RPN expression, however simply stated as a problem, gives plenty of room for building abstractions and making implementation choices. This is very useful, as it makes possible within each language to design a solution that is best fitted to the language. In fact, in every implementation I have tried my best to present the corresponding language in the most favourable way possible. Had I chosen a problem definition that forced a particular style of implementation, I would have limited myself to testing specific language features rather than the language's ability to solve problems.

A section in [1] is dedicated to the various algorithmic and implementation options. The reader is referred to that text for details, but in order to give an impression of the range of possibilities, here is a brief account of the ways in which an RPN expression can be (parsed and) computed:

- working backwards recursively;
- working forwards using a stack;
- working forwards recursively;
- reduction (replacing already evaluated sub-expressions with their respective results until a single number remains);
- concurrent evaluation of sub-expressions;
- reflection (letting the expression be interpreted in the language itself);
- employing a parser generator to automate parsing (and computing the sub-expressions as they get known in the course of parsing).

Whether the said options are really applicable may depend on other decisions and on inherent limitations of the language. For example, only some of the above are possible if input is being read in pieces rather than in whole lines.

A simple yet meaningful way to judge implementations, and through them the corresponding languages, is their size. Amazingly, the size differs wildly across languages. The

shortest program (Perl) appears to be only 8 lines long, while the largest one (Pascal) is almost 120, which makes a factor of 15!

The Perl implementation is short due to making use of regular expressions for parsing and replacement, which this language is particularly good in. As a rule, the implementations that fall in the 'reduction' category are rather compact, but they are not the only ones with this property. Languages that allow for automating, in one way or another, significant portion of parsing, lead to short programs. Conversely, if parsing is programmed in details, or a data container, such as a stack, with dynamic storage allocation, is explicitly programmed (which is the case in Pascal), the implementation is unavoidably long.

A particular reason for quite a large number of implementations to be of greater size than what can be expected is having to ensure that numbers in all formats required by the problem specification, and only such numbers are accepted. Surprisingly many languages provide means for, say, telling if a string represents a number, but place limitations on the form of the numeric data they accept.

Program length is important but not the sole criterion for language assessment and comparison. Programs have to be understandable. A short program is generally easier to understand than a longer one, but there are also other contributing factors. It may be that programs of a bit larger size than close to the minimal one tend to be much more easily readable for an inexperienced person, while still rather compact. The Ruby, Rexx, Python, D, Logo, Scala implementations are examples of this.

The amount of explanations needed for a program to be understood is also informative of program's comprehensibility. Implementations in languages with an unusual (or not widely popular) expression style certainly need more accompanying text to aid their understanding. In the described experiment, such are e. g. Lisp, Pop-11, and Erlang, each of which is represented by a relatively short program with (again, relatively) long explaining text.

It deserves mentioning that terse (and simple!) implementations are sometimes possible in rather old languages. A striking example of this is Snobol, now a 40 years old language with still unsurpassed expressive power for text processing. Other 'old' but rather effective languages are AWK, Logo, Rexx, Icon, and Refal (although a rather significant part of the Refal implementation has to deal with parsing and computing the value of a real number; strangely, real numbers can be used in expressions but not read in that language).

Although the problem statement certainly does not imply any extraordinary language features for solving it, some languages still make it impossible, or at least extremely hard, to write a complying implementation in them. In some cases, considering the respective language too important to miss, I have nevertheless written a solution, and made it clear how it is functionally different from what is required. JavaScript, for example, has no its own means for input and output – it borrows them from its host environment. Smalltalk has no standard channels for I/O, so one has to adopt a non-standard convention and (or) stick to a particular language dialect in this respect. A number of languages (ABC, Algol 60, BASIC, Euphoria, Vim script) turn to not provide means for detecting end-of-input, so the closest possible that can be done to ensure that an arbitrary amount of lines can be read by the RPN calculator is reading again and again, indefinitely.

(Syntax) error detection and handling being part of the requirements for implementing

a calculator in any language, it can be observed how useful exception-handling is in this respect. As it turns out, a great many of the implementations make use of such a mechanism where one is available, or of its equivalent in other cases. This is not because the implementations were artificially 'bent' to use exception handling, but simply due to exception-based control structuring really often winning against alternative solutions. CLU's authors were perhaps the first to realise that exception handling is not only for error handling but also a control construct of general utility. Although seemingly little attention, at least explicitly, is being paid to that more general use nowadays, it is notable that almost all contemporary programming languages feature exception handling: it has become one of the characteristics of modern languages and their use.

Some so-called stack-oriented languages themselves make use of the reverse-Polish notation, and therefore are semantically closest to the data format in the sample problem, viz. postfix arithmetic expressions. Moreover, most such languages are interpretive and reflective, so, in principle, RPN expressions should be readily evaluated in them. Despite this, for a number of reasons it turns out that the problem is not necessarily solvable with particular easiness in these languages. In Forth, for example, integer and floating-point computations are rather difficult to mix – in fact so much so that writing an RPN calculator in that language was eventually considered too great an effort and would lead to such a convoluted program that it was not worth doing it. This is paradoxical in view of the fact that Forth is the 'canonical' and still most widely used stack-oriented language. Joy was also left out: this quite interesting experimental language unfortunately lacks useful I/O and text processing capabilities to be practical. PostScript and Pop-11, two other representatives of this family of languages, happen to produce reasonably short solutions, but perhaps not as small and not as straightforward as may be hoped for. Of all stack-oriented languages, Factor was the only one in which a direct, truly small implementation of an RPN calculator was possible, in a reflective style.

Several languages, e. g. Lisp, Prolog, Haskell, and Erlang, include a built-in or library-provided lexical analyser, and it is tempting to employ it for the lexical phase of parsing an RPN expression. However, a closer look reveals that such a lexical analyser is always oriented towards the particular language to such an extent that it is hardly possible or practical to rely on it for parsing text that is unrelated to that language. For example, the Lisp lexer would regard as numbers more strings than the RPN calculator problem specification permits, and would conceal the presence of characters that are treated as comments in Lisp. In effect, the said lexers are not universal tools, but are mainly useful in programs that parse the same language as the one in which the program itself is written.

Some languages provide no particularly suitable tools for solving the specific problem under consideration – and in fact no tools for any other specific problem – but are so well designed as general-purpose programming instruments that it is almost always straightforward to write a program for whatever problem, even when all details must be coded from the ground up. Such well-known programming 'workhorses' are C and Scheme, each one in its own distinctive manner. Both are small languages, with only a small number of built-in or library functions. Still, as the RPN calculator problem shows once more, implementations in these languages are reasonably short and direct.

In order to appreciate this, one can compare, for example, C and Java. Although the latter language, unlike C, has enough library facilities to automate both reading and

tokenising the input to the calculator, the C implementation is only a little larger than the one in Java. Where C lacks certain specific functionality, it compensates with easily combinable general tools and laconism. Java, on the other hand, tends to exhibit a wordy and clumsy style of expression.

Concluding remarks. Assessing and comparing programming languages based on solving a sample problem in different languages (and possibly in more than one way in a language) is a practical, highly informative method. With a properly chosen problem and abiding by the rules of strict implementation, adhering to the language standard etc., it is possible to expose many of the relevant strengths and weaknesses of a language within the volume of a small program. In addition, such complete, true implementations prove better examples to be used in a general presentation of a language than the biased and incomplete ones usually seen in textbooks, on web sites, etc.

The experiment has shown some expected, or at least foreseeable, as well as some unanticipated results. The sheer diversity of algorithmic approaches and language constructs being exposed in the course of designing and implementing programs in many languages is a wealth of information in itself. It also provides insight with regard to how even a seemingly simple problem definition may lead to many substantially different computational patterns.

Obviously, other sample problems can be used to the same effect, including more than one together. An open problem in this respect is finding a set of programming tasks whose implementations would serve as a sufficient basis for language assessment and comparison in all aspects of language use.

REFERENCES

[1] B. Bantchev. *Programming language awareness centre.*
`http://www.math.bas.bg/bantchev/place`.
(This web site is continually under development and contains all implementations of the RPN calculator and a lot of other information on many programming languages. There are several sections of it. In particular, see the *Links* section referring to other multi-language efforts accessible over the Internet.)

Boyko Bantchev
Institute of Mathematics and Informatics
Acad. G. Bontchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: `bantchev@math.bas.bg`

ДА КАЖЕШ ЕДНО И СЪЩО НА МНОГО ЕЗИЦИ

Бойко Банчев

Тази статия описва експеримент и предлага метод за преценяване и сравняване на езици за програмиране. Методът е емпиричен и се основава на многократно решаване на задача чрез множество езици. Представени са някои наблюдения върху проведения експеримент.