On the Evolution of Unnatural Language

Les Hatton
CISM, Kingston University
http://www.leshatton.org/

April 2, 2011

Abstract

A search of the web reveals that natural language is considered to be language which a human would consider natural and unnatural language is language a human would consider artificial. This clearly raises interpretation questions and plenty of leeway so I am going to take the latter view here and bend it somewhat to consider programming languages as particularly good examples of unnatural language.

Programming languages are something like 60 years old. Their evolution tells us much about the triumph of creativity over parsimony; the need for diversity and the frequent triumph of politics over common sense or logic. Yet their minimum set of features is defined by the beauty and simplicity of the Böhm-Jacopini theorem [1] and their underlying similarity in information theoretic terms is shown in the theorem proved by myself in [3], [4].

This essay is a reflection on programming languages both old and new, engineering, systems evolution and the role of education.

1 A Little Essential Background

In 1966, two Italian computer scientists Corrado Böhm and Guiseppe Jacopini proved a very important and relatively unsung theorem which essentially demonstrated that every computable function could be implemented in a programming language which contained three control structures which we recognise today as sequence, selection and iteration [1]. This seminal paper also initiated the *structured programming* debate, however from our point of view it sets the minimum a programming language requires to implement every computable function.

Such minimum implementations would be a bit bleak to use in practice so it is interesting therefore to consider how much else computer scientists feel they must add to enhance "expressive ability".

1.1 Standardisation: the Good and the Bad

1966 was a busy year. As far as I can see, the first ANSI (American National Standard) programming language appeared in that year and became known as Fortran 66[1]. It was followed shortly after by ANSI standard COBOL in 1968 and a regular series of new languages over the next decades although these became subsumed within the ISO (International Standards Organisation) from around 1990, (the C standard was probably the first to jump ship when ANSI C89 became ISO C90 with a simple section renumbering). At the time, standards were incredibly important. Without them, chaos would have ensued and although the promise of complete machine independence was never achieved, if you knew what you were doing, you could retain very high levels of portability with a generally negligible effort to move to another environment. Most of the time, you even got the same answers give or take a couple of significant figures, [5].

Unfortunately, things have struggled a bit since then due to rampant featurism. I've sat on a few national standards bodies in my time and one thing was very clear even in the early stages. No language committee was going to sit still and let other languages steal a march on them with some wicked new features however useless they were or turned out to be. It was a regular topic of conversation at Fortran meetings I attended that "we needed to have pointers because C had them". In fact pointers and notably dynamic memory spread across all languages very quickly, (although with odd and non empirically supported restrictions in some, as was the case in Ada 83 for example). As it turns out, almost nothing to do with programming languages is empirically supported. We are not a critical discipline and therefore we are not a scientific one either in the Popperian sense but more of this later.

This really took off with the widespread adoption of object orientation from around 1990 in spite of a complete lack of any supporting evidence. Language committees literally fell over themselves in the rush to add OO features, however inappropriate it might have been. Languages blew up like food addicts at a free hamburger stall. Just have a look at the following if you don't believe me.

Language	Size 1	Size 2	Increase factor
Ada	270Kb zipped HTML (1983)	1093Kb (1995)	4
C	191 pages (1990)	401 pages (1999)	2
C++	808 pages (1999)	1370 pages (2010 draft)	1.7
Fortran	134 pages (1978)	354 pages (1990)	2.5

This whole process whereby languages have evolved into each others' spaces has always seemed odd to me and must surely be driven by the simple need to

[1]It was also the last year in which the lamentable and now massively overpaid England football team won anything of significance but let's not go there.

attract more market share. With hindsight, it would have been much simpler to provide languages fit for a particular purpose and more emphatically encourage heterogeneous systems where the strengths of different languages combine together to provide the best application - Ada for real-time and high-integrity, C for systems work, Fortran for mathematical computation, Perl for pattern recognition and so on. This of course has happened quite naturally to a certain extent. The GNAT translator is a hybrid of Ada95 and gcc which is itself written in C; GNAT will be run on an operating system written in C (Linux) or Windows (C++); Java compilers are written in Java and Java run-time environments in C; Perl translators are written in C; there are Python translators written in (at least) Java, C and Python, and so on. So heterogeneity has occurred naturally but individual languages have become leviathans in the process. Perhaps this is itself unstoppable.

In the period 1990 - present day, the landscape has changed dramatically. I have a copy of the ACM SIGPLAN notices of March 1993 on the History of Programming Languages, (Volume 28(3)). It gives short introductions to: Ada, Algol 68, C, C++, CLU, Concurrent Pascal, Formac, Forth, Icon, Lisp, Pascal, Prolog and Smalltalk.

In 2010, students at my university offered projects in the following languages, C, C#, C++, Java, Perl, PHP, MySQL, XML, HTML, XHTML, VB.Net on XP, of which Java, PHP and MySQL dominated. Just look how little this has in common with the prior SIGPLAN list.

1.2 Validation and Compilers

One very important area in which Ada has done particularly well is in the provision of compiler testing. Until April 2000 or so, it was common for ISO languages to be validated against the validation suites which were available. In the 1990s it was possible to check standardisation of C or Fortran compilers against information provided by national bodies like NIST in the USA. Since then compiler validation has become a distant dream for many languages and the Ada community should be congratulated on keeping them publicly available.

A propos of nothing in particular, I downloaded and measured the size of the Ada validation suite as shown below. Comparing these against two other validation suites which I happen to have licences for, (C90 and Fortran 77), reveals

Language	Validation suite (LOC)	Validation lines / Standard pages
Ada 95	355,498 (.ADA) + 253,748 (.A)	1,070
C 90	298,867 (.c) + 40,176 (.h)	1,800
Fortran 77	108,842 (.f)	820

These numbers were calculated in the case of Ada by

```
% find . -name '*.ADA' -exec wc -l {} \; \
     | awk '{sum += $1;}END{print sum;}'
% find . -name '*.A' -exec wc -l {} \; \
     | awk '{sum += $1;}END{print sum;}'
```

It is interesting to reflect on the above densities of validation lines to standard pages. Although rather approximate, I will note in passing that Ada validation might perhaps be made more extensive given the size of the Ada 95 standard.

To summarise here, standardisation initially served a very valuable function providing a level of portability which had not hitherto existed. Unfortunately, the need to avoid breaking old code, (backwards compatibility), makes the international standards process highly asymmetric whereby it is generally much easier to introduce new features of unknown value in a standard revision than it is to remove existing features of known problematic behaviour. Consequently the standards get bigger and bigger and considerable effort is needed to trim them back to a level of parsimony and precision appropriate to their use in high-integrity systems. This is not a process which can be continued indefinitely and some languages, for example, C, will struggle unless the thicket of unconstrained creativity can be cut back somewhat. Historically of course, the only alternative has been to start again with yet another new language, the evidence of which surrounds us.

2 Education and Programming Skills

In tandem with the growth and complexity of traditional programming language standards, CS education has changed dramatically. We now live in a world where classic programming such as is found in the study of compilers and data structures has been largely supplanted with concepts such as "mash-ups" in non-standardised languages and exotic development environments for systems such as those found in mobile phones. Such environments mix graphical methods with more traditional coding skills in a frequently very complex and, at least to my ageing eyes, arbitrary manner. As a result there has been a dramatic shift in the centre of gravity of skill sets which can make it very difficult to find competent developers in languages such as Ada and Fortran. Worse, they tend not to be taught widely in universities which almost entirely focus on a single language - the flavour of the last few years being Java. Whether this will be the case in the near future is impossible to tell as new paradigms come and go amongst the hardy-annuals of Ada, Fortran, C and C++.

Marrying a decline in traditional programming skills with increasing complexity in those languages requiring such skills will become a significant gap to fill. There is every sign that Cloud Computing demands, rapid development

environments such as Ruby on Rails and the general evolution of web applications will increase rather than decrease the size of this gap. It almost seems to me that two completely different programming paradigms have evolved side by side.

1. The traditional life-cycle languages which involve system design, specification, implementation and validation. It is hard to conceive of a high-integrity system being built in any other way.

2. The "throw sub-systems at each other and see what works" life-cycle, so prevalent in web development.

It remains to be seen how easily people will move between these two very different paradigms. It may be that I am simply getting old but I find it relatively difficult to build web technologies because I tend to approach them too analytically and my experience is not such a benefit. I suspect the converse is also true - the rigour and discipline of specifying and building a system on which peoples lives may depend would not I feel come easily to a Ruby on Rails developer. If we were to explicitly separate them and prepare people in different ways, then that would be perfectly satisfactory according to the demands of each. However, if we continue to prepare the vast majority of CS students at university for the latter of the two, the former will be starved.

3 Underlying Linguistic Similarity

Amidst this blizzard of languages and technologies, it is very reasonable to ask if there are any common denominators. Clearly some languages have great staying power even though they regularly morph into other forms. Fortran is a wonderful example with something like 50 years of sustained use, although 1960s Fortran has very little in common with Fortran 2008, formally approved in late 2010.

Surprisingly however, considering the enormous differences which exist between different applications and different implementations, there appears to be a beautiful form of clockwork underlying this massive externally forced complexity.

In 2009, I proved a theorem [3], [4], which shows that for any system of N components each containing t_i tokens such that the total number of tokens is given by

$$T = \sum_{i=1}^{N} t_i \tag{1}$$

then the probability p_i of a component of t_i tokens appearing is overwhelmingly likely to obey a power-law in the alphabet a_i of *unique* tokens used

$$p_i \sim a_i^{-\beta} \tag{2}$$

where β is a constant, under the constraints that the total size of the system and the total amount of Shannon information in the system is conserved. The theorem combines arguments from statistical physics with standard properties of Shannon's information theory, as can be found for example in [2].

Note that this is completely language independent - it makes no assumptions about the implementation language. In this sense, a token of a programming language is either a *fixed* token a_f, (for example keywords like **if else begin end procedure** or operators like $+$ $++$ $-$) or a *variable* token a_v, (for example identifier names or constants).

Given that in smaller components, the number of fixed tokens tends to predominate (there is a fixed token startup overhead of implementing anything in a programming language) and in larger components, variable tokens predominate, the theorem further predicts that

$$p_i \sim constant \tag{3}$$

for smaller components merging asymptotically into the power-law described in (2).

Investigating this experimentally has taken some time because of the need to write a universal token parser able to distinguish between fixed and variable tokens for a number of languages with enough intelligence to distinguish the start and end of components. This has been done so far for Ada, C, C++, Fortran, Java, Perl and Tcl-Tk. This lexer takes advantage of the standard lexical analysis generator *lex* and the rest is written in C. The results of analysing the distribution of sizes of components across highly disparate systems in several languages is shown in Figure 1. This shape is precisely what is predicted by the development which leads to (2) and (3).

This provides good experimental support for the theorem and suggests that underneath all the complexity of notation, there is a very strong similarity in how we express the information inherent in systems implemented in programming languages.

4 Conclusion

Perhaps then we have come full circle. We started off in 1966 with an underlying simplicity and here almost 50 years later, we appear once again to have an underlying simplicity. However, this does not disguise the fact that the languages themselves have either disappeared or become very complex systems in their

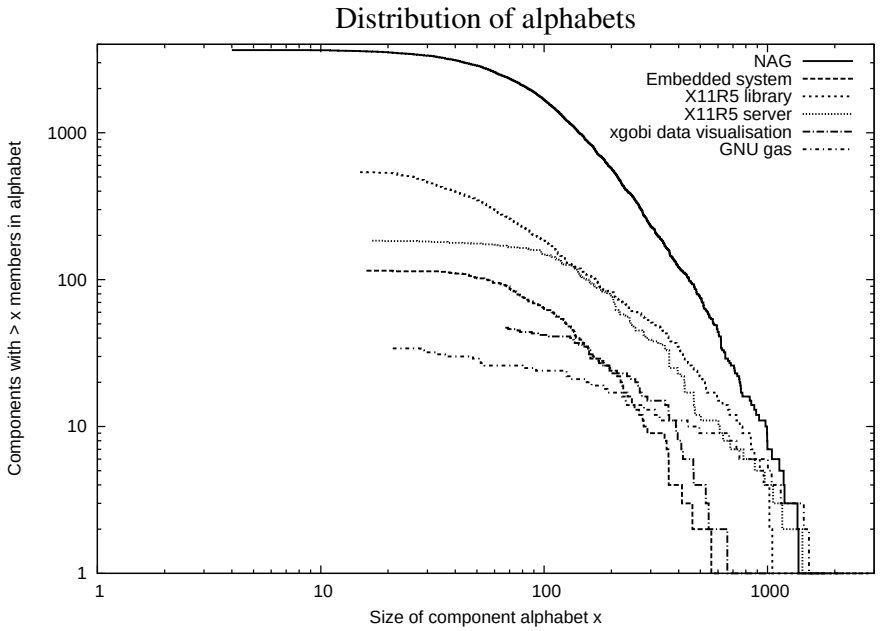

Figure 1: Examples of disparate systems in different languages which appear to follow the behaviour predicted by (2) and (3). To date, no systems have been found which depart from this.

own right, presenting a very steep learning challenge to new and even partly experienced engineers.

In parallel with this, changing patterns of education and an apparent and growing dichotomy of development paradigms places further stresses on the ability to produce high quality systems and engineers with the requisite experience. In particular, the modern "mash-up" methods seem strongly in the ascendant.

Will programming languages continue to accrete complexity ? Will the landscape in 30 years time be Ada 2032, C 2038, C++ 2036, Fortran 2040 or will interpreted languages dominate ? Given that these contenders have already had 30 years or more, it would be a brave person to bet against them in spite of the above concerns.

References

[1] Boehm, C., Jacopini, G.: Flow Diagrams, Turing machines, and Languages with only Two Formation Rules". Communications of the ACM 9(5), p.366–371 (1966)

[2] Cherry, C.: On Human Communication. John Wiley Science Editions (1963), library of Congress 56-9820

[3] Hatton, L.: Power-law distributions of component sizes in general software systems. IEEE Transactions on Software Engineering (July/August 2009)

[4] Hatton, L.: Power-laws, persistence and the distribution of information in software systems. preprint available at http://www.leshatton.org/variations_2010.html (January 2010)

[5] Hatton, L., Wright, A., Smith, S., Parkes, G., Bennett, P., Laws, R.: Sks: A large scale exercise in fortran 77 portability. Software, Practice and Experience 18(4), 301–329 (1988)