Programming Language Feature Agglomeration

Jeremy Singer Callum Cameron Marc Alexander
University of Glasgow
firstname.lastname@glasgow.ac.uk

Abstract

Feature-creep is a well-known phenomenon in software systems. In this paper, we argue that feature-creep also occurs in the domain of programming languages. Recent languages are more expressive than earlier languages. However recent languages generally extend rather than replace the syntax (sometimes) and semantics (almost always) of earlier languages. We demonstrate this trend of agglomeration in a sequence of languages comprising Pascal, C, Java, and Scala. These are all block-structured Algol-derived languages, with earlier languages providing explicit inspiration for later ones. We present empirical evidence from several language-specific sources, including grammar definitions and canonical manuals. The evidence suggests that there is a trend of increasing complexity in modern languages that have evolved from earlier languages.

1. Introduction

As programming language theory has developed and compiler technology has become more powerful, so programming languages have become increasingly complex. This phenomenon is recognized in the evolution of a single language like Fortran [14]: the initial specification of FORTRAN [2] is vastly different to Fortran 2008. Features from the earliest version of FORTRAN form a small subset of modern-day Fortran, which has accumulated a host of new syntactic constructs and behaviors. In this paper, we will observe the same trends, i.e.

1. increasing complexity
2. backwards compatibility

in a family of related programming languages, where later languages are, in some sense, successors of earlier languages. The four languages we study in this paper are: Pascal, C, Java and Scala.

One key constraint is that new programming languages need to resemble existing languages in order to be widely and rapidly adopted. Syntactic familiarity engenders trust on the part of the developers. This obliges language designers to retain old features of languages in an attempt to support backwards compatibility, or at least some semblance of it. For instance, Gosling [6] states that 'we tried to stick as close as possible to C++ in order to make the [Java] system more comprehensible.' Stroustrup [16] explains that C++ retains 'the lowlevel and unsafe features of C' since he strongly feels that 'a language designer has no business of trying to force programmers to use a particular style'.

In view of this constraint, new languages are likely to be relatively large and unwieldy. The new language tends to have new features, otherwise there is no compelling reason for it. However the new language generally also supports features of existing languages, otherwise no-one will use it. In this paper, we provide some empirical evidence to support our argument that modern languages (Java, Scala) are significantly larger than older languages (Pascal, C) due to their greater complexity.

1.1 Contributions

In this work, we make the following contributions:

1. We outline four metrics to quantify the relative complexity of a set of programming languages.
2. We provide an empirical study of four programming languages, showing how recent languages are inherently more complex than earlier languages.
3. We discuss how language designers might take concrete steps to avoid excessive complexity when devising new programming languages.

2. Selected Languages

In this language study, we are going to consider four mainstream programming languages. Below we set out the systematic basis for the selection of these languages. The criteria are:

- *popularity*: each language must feature in the Tiobe index[1] of top 50 programming languages.
- *tool availability*: each language must have freely available toolchains, ideally with source code available for inspection.
- *primary documentation*: each language must have a published manual, co-authored by one or more of the language designers.
- *conceptual elegance*: each language must be created by persons recognized as programming language authorities by their peers. For instance, the language designers should be Turing award winners or ACM Fellows in recognition of their contributions to programming language progress.

Table 1 lists the four languages selected for this study, which meet the criteria outlined above.

Importantly, there are clear inter-dependences between the selected languages. Pascal is the earliest language in our selection. C appears to be motivated in part by the developers' frustration with Pascal [9]. Java [6] relies strongly on the tradition of C and C++. Scala [12] is explicitly designed to 'work seamlessly with ... mainstream object-oriented languages such as ... Java.'

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, checked May 2014

language	*year*
Pascal	1970
C	1978
Java	1996
Scala	2003

Table 1. Programming languages systematically selected for study

We intend to trace these relationships between the languages, in terms of their chronological and conceptual orderings.

3. Language Syntax Study

This section focuses on comparative studies of artifacts relating to the syntax of the various programming languages.

3.1 Keywords

In this initial study, we examine the number of *keywords* in each language. This is a simplistic measure of complexity, in terms of the size of each language's primitive vocabulary. Keywords are used to express data types, data structures, control flow and modularity, *inter alia*. A larger set of keywords indicates a more complex language, with a richer set of programming constructs. We are aware that C-like languages rely on punctuation symbols more than Pascal-like languages. The effect of this reliance on symbols will be considered in the grammatical comparison in Section 3.1.

We retrieve keyword lists from freely accessible web sites, as listed below. ISO standards and similar sources would provide a more reliable basis for this study.

The set of **Pascal** keywords is specified at `http://www.gnu-pascal.de/gpc/Keywords.html`. A total of *36 keywords* are valid in any dialect of Pascal.

The set of **C** keywords is listed at `http://en.wikipedia.org/wiki/C_syntax#Reserved_keywords`. This list gives a total of *37 keywords*.

The **Java** Language Specification keywords at `http://docs.oracle.com/javase/specs/jls/se5.0/html/lexical.html#3.9` gives *50* keywords for Java version 1.5. Note that this list does not include literals e.g. `null`, `true`, or reserved words that are not current keywords e.g. `goto`, `operator`. Earlier version of Java featured fewer keywords. There are *46* keywords in Java version 1.0.

The **Scala** reference document at `http://www.scala-lang.org/docu/files/ScalaReference.pdf` lists *49* reserved words.

From this initial investigation, the languages cluster into two groups. Pascal and C have similar sized vocabularies. Both are imperative languages with support for user-defined abstract data types. Java and Scala have similar keyword vocabulary sizes to each other, since they are both object-oriented languages that are significantly more complex than the earlier imperative languages.

> **Observed Evolution:** Later languages have more keywords.

3.2 Grammars

In the next investigation, we examine the sizes of the grammars that define each language. There is a range of metrics for grammar complexity, e.g. [7]. We measure the number of rules in the BNF grammar specification for each language. Rather than considering the keywords in isolation, we study how they may be composed to generate legal programs. This is another measure of language complexity.

language	*book*	*pages*
Pascal	[8]	167
C	[10]	272
Java	[1]	891
Scala	[13]	736

Table 3. Programming Language Manual sizes

We rely on extant open source grammar definition files for our four languages under study. These files are generally inputs to parser generator systems like yacc. Table 2 shows the details of the actual grammar files we used. Although these grammars are not all canonical, they should give a general representation of the relative complexity of the language grammar. (Only the Scala grammar is written by the original language developers.)

In each eBNF grammar specification, we will count the number of non-terminal expansions, i.e. the number of actual rules. Where a single rule consists of multiple clauses, i.e. A ::= B | C, we count each alternate clause B and C as a separate rule.

Table 2 is not particularly conclusive. It seems that Pascal and Java have more complex grammars, largely due to their built-in facilities for modularization, which C does not support directly. The Scala grammar seems smaller, but the comparison may be unfair since it is specified in a different notation.

> **Observed Evolution:** The study is inconclusive based on the grammatical evidence above. There is no evolutionary trend in any direction.

4. Language Manual Study

This section focuses on the standard manuals for each programming language under study. We attempt to make a systematic selection of manuals: our aim is to select the canonical manual for each language. This requires (1) the manual to be written by the language designer(s) and (2) the first edition of the manual to be published within five years of the public release of the language.

For Pascal, we choose the *Pascal User Manual and Report* [8]. For C, we select the K&R textbook, in its most popular ANSI C edition [10]. For Java, we choose the *Java Programming Language* manual [1] rather than the more formal (but less accessible) *Java Language Specification*. For Scala, we select the *Programming in Scala* manual [13].

4.1 Manual Size

This study measures the number of pages in each programming language manual. The manual size should be an indication of the language's relative complexity. Table 3 gives the results of this analysis.

Again, the languages appear to cluster into two distinct groups. Pascal and C have more concise manuals, whereas the Java and Scala manuals are more verbose. This metric may correlate directly with the number of keywords in a language, see Section 3.1. A larger keyword vocabulary will naturally require more explanation. Kernighan and Ritchie [10] comment that 'C is not a big language, and it is not well served by a big book.'

It is also likely that as the conceptual complexity of programming languages increases, then their descriptions become longer. Some programming constructs evolve to become more complicated, as we discover in the next section. The well recognized phenomenon of evolution within a single language may be observed using the same metric of programming language manual size. Ta-

language	*source location*	*specification language*	language version	rules
Pascal	`http://ccia.ei.uvigo.es/docencia/PL/doc/bison/pascal/pascal.y`	Bison	ISO 7185 Level 0	242
C	`http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`	Bison	ANSI C89	211
Java	`http://www.cs.dartmouth.edu/~mckeeman/cs118/notation/java11.html`	Bison	v1.1	278
Scala	`http://lampsvn.epfl.ch/svn-repos/scala/scala-documentation/trunk/src/reference/SyntaxSummary.tex`	eBNF in LaTeX	v2.7	200

Table 2. Grammar Files used for Grammar Size Study

year	*Java v.*	*edition*	*pages*
1996	1.0	1st	333
1997	1.1	2nd	442
2000	1.3	3rd	595
2006	1.5	4th	891

Table 4. Growth in size of 'The Java Programming Language' through four editions (data from Google books)

ble 4 shows how the Java manual has grown over various editions, tracking the increasing complexity of the language itself.

Another cause for manual growth is that older books have more compact typesetting. Newer books are cheaper to typeset, and electronic books largely eliminate concerns of per-page cost.

Observed Evolution: Later languages have longer manuals.

4.2 Linguistic Comparison

In this final study, we examine a programming construct that is shared between all the languages under study. For this cross-sectional study, we consider the for loop construct. This iterative control flow statement is supported by all four languages, although there are differences in the complexity of the construct between languages.

We examine the main section in each programming language's canonical manual that describes the for loop construct in that language. We study the natural language descriptions and compare metrics on them. Table 5 describes the source texts used in the analysis.

As Table 5 shows, the descriptions of for loops become more verbose with more recent languages. The manual authors use more sentences in their descriptions. The sentence length increases with the modernity of the language. Note also that the average sentence length is greater for modern languages (Java and Scala) than for older languages (Pascal and C). This increase in description length reflects the corresponding increase in for loop semantic richness. Whereas Pascal for loops merely increment or decrement the index variable at each iteration, C for loops can perform an arbitrary update of any number of index variables. The original version of Java retains the C semantics, but since has been extended to include for each style loops operating over iterable data structures. The Scala for loop supports a variety of iteration styles including traditional integer index variables, iterating over data structures, and list comprehension style iteration.

In terms of source code provided in the manuals, modern language manuals provide more source code fragments, although the density of source code to natural language is much lower. The modern manuals (Scala and Java) provide much more commentary on the source code fragments. For instance, Odersky et al. [13] present a Scala grep utility function, accompanied by the following comment: 'A variable named `trimmed` is introduced halfway through the for expression. That variable is initialized to the result of `line.trim`. The rest of the for expression then uses the new variable in two places…'

This level of detail is lacking in the earlier manuals. For instance Jensen and Wirth [8] present an 18-line program that iteratively computes the sum of a series of real valued numbers using four different orderings of the numbers. There is no accompanying natural language commentary, and after the source code, the question is posed: 'Why do the four "identical" sums differ?' The reader is expected to infer that the answer is due to loss of precision in the floating-point arithmetic.

In the earlier manuals (particularly [8]) the language is concise and technical. In the later manuals, the language is more informal and richer in metaphor. For instance, Odersky et al. [13] introduce the Scala for loop as follows: 'Scala's for expression is a Swiss army knife of iteration. It lets you combine a few simple ingredients in different ways to express a wide variety of iterations. '

Observed Evolution: Later languages have more complex constructs that require extended descriptions and examples.

5. Threats to Validity

Unrepresentative Language Selection: As outlined in Section 2, we followed a principled approach in selecting the four languages used for this study. Other languages that meet the specified criteria include C++ and C# but time restrictions prevented their consideration. It would be interesting to apply the same metrics to Fortran as a form of calibration since agglomeration has clearly happened with this language (as successive standards are always backwards compatible).

Inappropriate Artifacts: We attempted to remain as general as possible, so we did not consider particular compilers for each language. Instead we restricted our attention largely to language specifications and canonical documentation. We note that the study of ISO style documentation would be more authoritative, but this is only possible for standardized languages like Pascal and C. Further, we focused on the languages rather than any corpus of programs written in those languages. Again, this is in the interests of generality. We did not examine the relative sizes of language standard libraries. In some cases, e.g. Smalltalk, the core language has relatively little syntax and many constructs like iteration constructs are actually library calls.

language	*manual*	*section (pages)*	*sentences*	*words/sentence*	*code fragments*
Pascal	[8]	4.C.3 (23–26)	12	13.2	9
C	[10]	3.5 (60–63)	47	16.7	11
Java	[1]	10.5 (208–212	57	21.5	13
Scala	[13]	7.3 (154–159)	70	18.1	13

Table 5. For loop descriptions in canonical language manuals

Inappropriate Metrics: We applied a range of metrics to the language artifacts in Sections 3 and 4. The metrics calculations are intuitively straightforward and independently reproducible.

Analysis Bias: With only four data points for each study and a natural clustering of older imperative languages versus newer object-oriented languages, some subjective analysis bias is likely. This could be mitigated by a larger study, involving more languages and a wider range of metrics.

6. Related Work

Chen et al. [4] provide a detailed study of 17 programming languages including C, Java and Pascal. They apply statistical analysis to correlate a range of quantitative factors such as generality and machine independence for each language. They derive equations to predict the relative popularity of programming languages based on multivariate regression. Their quantification approach could be perceived as an arbitrary scheme, since it is based on a subjective assignment of values to categorical features. In contrast, we have used a metrics-based approach to assess language complexity.

Steele [15] argues that languages with restricted vocabularies (whether natural languages or programming languages) are cumbersome. Although small languages may be learnt quickly, it is less efficient to communicate using small languages. He states that Java began as a small language and is growing gradually over time. Java programmers have evolved with the language. We provide empirical evidence for the growth of Java in Table 4. We suggest that Steele's argument also applies to a family of related languages.

Overbey et al. [14] study the evolution of Fortran. They apply refactoring transformations to legacy Fortran code to remove outdated constructs which are difficult to optimize for modern architectures. Thus they support all language constructs, but transparently map legacy constructs to newer language idiom. This is a transparent form of backward compatibility. Modern compilers may support similar optimizations, e.g. loop vectorization in GCC effectively turns a sequential for loop into a parallel for loop.

7. Conclusions

In this paper, we present quantitative evidence to show that Java and Scala are larger and more complex than C and Pascal. It seems inevitable that languages which retain some element of backwards compatibility (or at least, syntactic familiarity) are bound to be larger than the languages they are intended to replace. This is a natural feature of programming language evolution across languages. However the concern is that newer languages will agglomerate so many legacy language features that they can never discard. Thus newer languages become excessively large and unwieldy, hence no longer useful.

The JavaScript language is multi-paradigm: it is a modern, widely adopted language that supports many different programming styles. However some experienced developers recommend the use of an *elegant subset* of the language, which Crockford [5] calls 'the good parts' of JavaScript. On a similar note, recent implementations of JavaScript support `strict` mode, which prevents the use of certain problematic constructs such as the `with` statement [11]. Perhaps every large, modern language has an elegant subset? Should this subset be a moving target over time, as newer features are introduced? Brooks [3] suggests the existence of a 'sweet spot' in language complexity, at which point a language is sufficiently expressive to be useful but not excessively complex to become intractable.

Should languages retain backwards compatibility features in perpetuity? In the same way that Java APIs can be marked as `deprecated`, perhaps programming language constructs could be deprecated too? A compiler could issue warnings to users that certain constructs are not well-supported, perhaps because of performance issues. This would encourage the developer to rewrite code to use alternative constructs.

In summary, we advocate that such questions should be addressed in the context of a diverse, co-evolving family of languages, rather than for a single language only.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison-Wesley, 4th edition, 2005.

[2] J. Backus. Specifications for the IBM formula translating system, FORTRAN. Technical report, IBM, Nov. 1954.

[3] J. Brooks, F.P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[4] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *Software, IEEE*, 22(3):72–79, May 2005.

[5] D. Crockford. *JavaScript: the good parts*. O'Reilly, 2008.

[6] J. Gosling. Java: An overview, 1995.

[7] J. Gruska. Complexity and unambiguity of context-free grammars and languages. *Information and Control*, 18(5):502–519, 1971.

[8] K. Jensen and N. Wirth. *Pascal user manual and report*. Springer, 2nd edition, 1974.

[9] B. W. Kernighan. Why pascal is not my favorite programming language. Technical Report 100, Bell Laboratories, 1981. URL `http://www.fh-jena.de/~kleine/history/languages/Kernighan-WhyPascalIsNotMyFavoriteProgrammingLanguage.pdf`.

[10] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall, 2nd edition, 1988.

[11] Mozilla Developer Network. Strict mode. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode`. accessed 21 July 2014.

[12] M. Odersky. The Scala experiment: can we provide better language support for component systems? In *POPL*, pages 166–167, 2006.

[13] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 1st edition, 2008.

[14] J. Overbey, S. Negara, and R. Johnson. Refactoring and the evolution of fortran. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 28–34, May 2009.

[15] G. L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[16] B. Stroustrup. A history of C++: 1979–1991. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 271–297, 1993.