Programming Languages shouldn't be "too Natural"

Dino Mandrioli
Politecnico di Milano
Piazza Leonardo Da Vinci 32
20133, Milano, Italy
+390223993522
dino.mandrioli@polimi.it

Matteo Pradella
Politecnico di Milano
Piazza Leonardo Da Vinci 32
20133, Milano, Italy
+390223993495
matteo.pradella@polimi.it

ABSTRACT

Despite much research on programming language principles, most often the design of modern languages ignores such principles which results in cumbersome, hard to understand, and error-prone code. We substantiate our claim through a short sampling of the features of some widely used languages and by referring to other criticisms widely publicized in the literature. We argue that a major reason of such an unpleasant state of the art is that programming languages evolve in a way that too much resembles that of natural languages. We advocate a different attitude in programming language design, going back to essentiality and rigorous application of few basic, well-chosen principles.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features

General Terms

Languages, Theory, Human Factors.

Keywords

Programming language principles, best practices, programming language defects.

1. INTRODUCTION

High level programming languages (PLs) have been motivated by the wish of abstracting from machine peculiarities and making the programmer's job more comfortable; as a consequence, from the very beginning they were inspired by natural languages (NLs): the ancestors FORTRAN and COBOL and, even more, ALGOL 60 not only included keywords borrowed from English but, mainly, they resumed the typical nested structure of natural language sentences.

NLs, however, exhibit many features which do not prove effective in programming, where, typically, we want precise, highly reliable products. They are flexible and tolerant; this allows, for instance, to understand even syntactically incorrect sentences; they are redundant, so that in some cases we may appreciate using several paragraphs to express a concept that could be explained as well with few lines, possibly just for the pleasure of an elegant reading. These "virtues" of NLs, however, have also some unavoidable drawbacks: they are deeply ambiguous, which often causes lack of clarity and misunderstandings with more or less serious consequences; they evolve in an uncontrolled and unpredictable way, generating many dialects and fragmentation of the various communities, as taught by the biblical history of the Babel Tower.

In this paper we argue that the recent history of PLs, probably pushed by the technology advances which made defining and implementing new languages relatively easy, went too far in the path of importing NL features, with the result of importing also their undesirable properties; to support our thesis, next we briefly and critically examine the historical evolution of PLs, then we propose a few simple principles and guidelines to avoid the risks and defects of many modern PLs.

.

2. A RETROSPECTIVE OF PROGRAMMING LANGUAGE EVOLUTION

Among the earliest high level languages FORTRAN is the one that most ignited research on the formal aspects of language syntax. Not by chance the BNF (Backus-Naur Form) and the context free grammar are essentially the same formalism born almost independently and contemporarily within the computer science and the mathematical linguistic communities. This formalism has been the stage on which the formal language and automata theory developed fundamental results in parsing and compiling during the 60s and 70s.

In those years, while the theory of syntax directed parsing and translation led to powerful and general algorithms to (almost) automatically generate a compiler from a language definition, plenty of new languages were defined with the goal of enhancing their quality and usability, though not always it was clearly understood that often their quality goals were conflicting, e.g., general purpose vs. efficiency, expressive power vs. simplicity and ease of use, machine control vs. abstraction, easy compilation vs. run-time efficiency, … .

Despite the fact that industrial applications almost ignored so much research -FORTRAN and COBOL, respectively, remained the almost unique widely used languages within engineering and business applications- several fundamental milestones were set in those years: ALGOL 60 is still now considered the ancestor of all block-structured languages, which lead to the full consequence the idea of defining complex constructs by nesting simple ones; ALGOL 68 is remembered as associated with the principle of *orthogonality*, i.e. deriving any feature as the combination of few basic ones; Simula 67 and Pascal ignited type theory which further evolved towards the notion of class and object-oriented programming; LISP and its followers pursued a *functional* style of programming.

Some of them were the result of a joint and coordinated effort of several communities whereas others were the invention of single researchers or small groups, but in all cases they were the result of elaborating a *few basic principles* and deriving language details as the consequence of such principles. Many fundamental *programming language concepts* were developed during those years, such as dynamic vs. static typing, strong typing, dynamic vs. static scoping, …: new PLs were designed by choosing some of them as the driving principle.

Noticeably in the same time IBM developed PL/I with the purpose of making it *the* universal language: it was simply a rather unstructured collection of all then known language features plus a few new ones with no rational design behind that; despite the great industrial and commercial power of the industry supporting it, PL/I is now remembered as a major failure in the history of PLs.

Adding new features to a language may increase its "power" and flexibility but often at some price in terms of clarity and ambiguity. Some apparent "enrichments" brought back in the PLs ambiguity problems typical of NLs: the same syntactic construct could be reasonably interpreted with different meanings with no precise indication of a preferred one; as a natural consequence this led to typical and serious problems of compatibility, portability and, in general, correctness.

This led various researchers to advocate *formal semantics*, i.e., the use of formalisms suitable to give a mathematically defined meaning to each syntactic construct. Many formalisms of this type have been proposed in the literature and occasionally some of them have been applied to real-life PLs; none of them, however, has been widely accepted by practitioners let alone any kind of official standardization. This reluctance is probably due to the heaviness of mathematical formalisms adopted so far to obtain complete definition of PL semantics. Indeed, whereas the concept of context-free grammar is inspired by, and very similar to, the way we normally explain the rationale of NL syntax, semantic formalisms have little similarity with the way we assign meaning to NL sentences.

The history of PLs exhibits an incredible number of languages and dialects derived therefrom, as it typically happens with NLs. However, whereas in the past new languages had little chance to reach a wide audience, were seldom adopted outside their "birth place", and even important ones, such as the ALGOL's family, remained in the realm of academic investigation, more recently we saw the phenomenon of many "suddenly and unexpectedly successful" languages; some of them died as soon as they were born, but others enjoyed a lasting success and fame. Two possible reasons for this new phenomenon are the explosion in terms of technological power, which now allows to "deliver" a working language in a few weeks, or even days, and the parallel growth of the user community, which, even restricting it to the programmers' one, is now orders of magnitude larger than the few specialists of the 50s and 60s.

Let us consider in particular the case of C. With all the due respect to its historical and technical merits, we believe that it suffers from an "original sin," which has then been inherited by most of its descendants: it has been invented and rapidly realized to help its originators in a specific, and, in that case, very important job, i.e., replacing the traditional use of assembly languages during the development of UNIX; thus, it was designed with exactly *that* purpose in mind, in particular supporting the habits of very expert and specialized programmers. A showy consequence of this feature is the plenty of abbreviations it exhibits; in many cases such abbreviations are indeed useful and became common practice with no risk of misuse, such as, e.g., '->' for '(...*).' (but how many seconds are saved by its use?), others are semantically "safe" after some non obvious clarification, such as the difference between `++i` and `i++`; but when we go into deeper semantic issues critical problems arise.

C led to the extreme consequences the "hybridation" between *expressions* as "something that delivers a value" and *statements* as "something that changes the state" with the goal of obtaining two results in one shot; but another way of expressing the same concept is the term *side effect*, which is often and correctly taught as a major risk in programming. Despite the long life of C and its systematic and continuous standardization, still critical ambiguities derive from this feature such as the "folk example" below.

```
int main(void) {
       int x = 0;
       return (x = 1) + (x = 2);
}
```

The official semantics of the above code is 'undefined', which leaves the implementer free to deliver arbitrary values; in fact some return 3 and some others 4. What if a piece of code such as this becomes part of a mission critical system?

Most modern PLs owe much to C, including its "original sin." Since its birth in fact, many new languages borrowed various syntactic constructs therefrom and plugged new elements, even major new concepts such as object orientation, into its syntactic skeleton. This process however occurred in a very unordered way, despite in many cases more or less official managing and standardization committees were set up. In our view it resembles too much the way NLs evolve where often some "users" -e.g. gangs of boys, TV commercials- introduce new slang expressions, import terms from other languages, … some of which then gain larger and larger use, with little or no control on the final effect on the "purity" of the mother tongue. In the case of PLs this often results in an accumulation of redundant or even conflicting features in a way that was already and uselessly blamed a long time ago by T. Hoare in his Turing lecture [4]. In other words, short term "user satisfaction" overwhelms rigorous design and evolution planning.

A typical example of such a "wild" language generation and managing is blatantly offered by PHP, which is almost universally criticized (see, e.g., [6]) but nevertheless has been widely adopted in important applications.

In summary, our viewpoint is that the way PLs are invented, developed, and adopted within user communities now-a-days borrows too much from NLs; not by chance many new languages are -or at least are born as- scripting languages, which typically are oriented towards translating any idea into an immediate action as opposed to organizing a well-thought design into a structured implementation and documentation.

To better exemplify, Table 1 overviews a few widely used modern languages from the point of view of the process that manages their evolution and its consequences. We are bewildered by the fact that a long history of research on PLs produced sound principles and techniques for their development but is mostly ignored by modern practices; the defects that result from this attitude are soon apparent and properly pointed out in the literature but nevertheless … the bad practices seem to keep going. On the one hand, the fact that PL evolution resembles NL one is "natural" since both of them are human-generated, but on the other hand in many cases the natural human attitude must be disciplined and "educated": an old Latin motto states "errare humanum est, sed perseverare diabolicum."[1]

Table 1.

Language	**Some example known issues (not a survey)**
Java	Many non-trivial features (reflection, generics, lambda, ...) have been added after its birth in an originally unplanned way.
C++	Some authors present it as a "federation of languages". It exhibits many ways of doing the same thing, often for compatibility with C or older versions of the language.
JavaScript	Exhibits a number of inconsistent choices. It tends to carry over a computation "no matter what": see Example 1 below. There are many ways of doing the same thing: see, e.g., something basic like inheritance! Having nullable types is a bad idea [5], and JavaScript has <u>two</u> kinds of null: *null* and *undefined*.
Python	Some basic aspects of the language were changed: e.g. method resolution order; nested static scope; unification between classes and types; parameters in generators. Decorators have been added for flexibility. There are two partially incompatible versions of the language, the 2.x family and the 3.x family. The 2.x family will be discontinued in 2020.
Ruby	No formal grammar exists: it must be inferred by the interpreter source code. Block, Proc, and Lambda are basically the same thing, and exhibit puzzling semantic variations: see Example 2 below.
Perl	It follows an almost opposite approach w.r.t. the principles recommended here: his designer L. Wall often compares Perl to a natural language and explains his decisions in Perl's design with linguistic rationale. For instance, the first Perl slogan is "There's more than one way to do it".

[1] To err is human, but to persist in error is diabolical.

Table 2 (Continued)

Example 1 - JavaScript	**Example 2 - Ruby**
> function f(x) { return x*x*x; } > f(5) 125 > f(5,3,112) 125 > f([1,2]) NaN > f(3,"hi!") 27	> f = Proc.new {|x| x*x*x} > f.call(7) 343 > f.call(7,6) 343 > g = lambda {|x| x*x*x } > g.call(5) 125 > y.call(5,6) Argument Error: wrong number of arguments (2 for 1)

3. BACK TO THE PRINCIPLES

On the contrary, we advocate going back to a more rigorous style, as it should be best practice in all non-trivial engineering activities. Here we propose a sample of positive recommendations in a deliberately rather provocative style.

If for any reason you are beginning to think about building your own, new language, the number one question to ask yourself is: "Do we really need a *new* language?" Are you sure that within the enormous panorama of the existing languages and tools you cannot find anything that can be adopted for, or possibly adapted to, your goals? Only if the answer to this question is a well-thought "Yes" proceed with your endeavor and keep in mind the following guidelines:

I. Learn from the history of PLs; carefully choose a few basic principles and derive detail decisions therefrom rather than "piling up" new features with a trial-and-error attitude. Of course the choice of principles must be taken in a coherent way on the basis of the goals and application field of the language; e.g. strong, possibly static typing, could be chosen for PLs devoted to programming critical and efficient systems, the flexibility of dynamic or loose typing could be preferred for "exploratory languages" such as scripting languages.

II. Exploit the precision and rigor provided by formal notations, possibly not only for syntactic definitions but also for critical and intricate semantic aspects. A "corollary" of the exploitation of formal notations is the possibility of applying tools such as automatic parsers and code generators; even if this recommendation is sometimes contradicted [1] by the argument that now-a-days building a compiler for a new language is not such a burden, we believe that using an automatic tool such as an LR parser generator compels to define and use syntactic features in a disciplined way.

III. Most often a language must evolve; thus, plan carefully its extensibility from the beginning. If a very flexible language is planned, that must allow for the definition of internal domain specific languages (DSL) or new constructs, avoid "clever syntactics hacks" and plan full-fledged facilities (e.g. Lisp macros) from its inception. For instance, even a recent successful language, such as Scala, despite its academic pedigree, presents a (probably too) flexible and *ad hoc* syntax, with facilities and special cases for dropping dots and parentheses in method calls, or swapping the order of object and method names. Such features are in general a double-edged sword: they can be convenient for introducing rapidly a DSL, but make the compiler more complex and are often a source of nasty bugs. Since version 2.10, Scala contains a Lisp-like macro facility (albeit still experimental), which will possibly make the previous approach obsolete but a still present source of complexity.

IV. Avoid or at least minimize redundancy, overloading, short notations, etc. If you deem a particular feature definitely useful, rigorously verify that it does not generate ambiguities or conflicts: e.g., are we sure that allowing for the omission of ';' between statements in JavaScript is a really useful feature? It makes the code probably "look nicer", like in the elegant Python syntax, but it should be done right: in Python newlines work as separators - if you don't want a newline, you *must* resort to the "despised" semicolon. On the contrary, the semicolon inference algorithm in JavaScript is a notorious source of insidious bugs. More generally: the recent history of important languages show an incredible amount of freedom left to the programmer; subsequently we see plenty of criticisms and recommendations for a disciplined and limited use of most language features (see, e.g. [2]): wasn't it better to examine *a priori* their pros and cons?
Also the freedom left to the implementer should be carefully evaluated. Besides the above example concerning the C language, another example that we criticize is the semantics of the **in-out** parameter in Ada, where the implementer is left free to adopt either a *by-copy* parameter passing technique or a *by-reference* one; the two techniques, however, are not semantically equivalent and this can cause even dramatic effects in some critical cases.

V. Keep the language development process constantly under control. Do not be afraid of experimenting and re-iterating, but do it in a rigorous way. Do not rush towards "release 1.0". An "advisory board" could be a useful means to achieve this goal. When the language begins to have a sizable community, it should monitor user feedbacks and drive language evolution in such a way that the original principles on which the language is rooted are not violated. Provide complete, precise, and consistent documentation throughout the language life; appendices based on formal specification are useful to disambiguate critical interpretations. Of course a good process is no guarantee of a good result but, in most non-trivial cases, is almost a necessary precondition.

To conclude with a bit of optimism, we noticed some encouraging tendencies. For instance, it is now broadly accepted that requiring breaks after cases in the classical `switch` construct, as introduced in C and later adopted without modifications in C++, Objective-C, Java, and JavaScript, was a bad idea and a source of a good number of errors (see for instance the recent Apple's SSL bug). Many of the newly proposed languages that have a syntax based on C, e.g., Scala, Go, TypeScript, Dart, Rust, Swift, fix this issue by using variants or totally different constructs.

An interesting case is the one of Rust, currently in development at Mozilla. Rust's design is based on clean and well-stated principles; moreover, while going toward version 1.0, some of the features of the language were indeed removed. So, Rust can be seen as a recent example of a new language, born and developed within a purely industrial and practical setting, that is in agreement with our recommendations.

After decades when many languages rooted in sound principles were proposed by the academia and ignored by industry and more recent decades when other "extemporary" languages were generated and occasionally gained wide acceptance in the practitioners community, maybe these are first signs of people leaving the "dark side" … .

4. REFERENCES

[1] Bright, W. 2014. So You Want To Write Your Own Language?. Dr Dobb's, January 21, 2014.

[2] Crockford, D. 2008. JavaScript: The Good Parts O'Reilly.

[3] Dijkstra, E. W. 1976 *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ.

[4] Hoare, C. A. R.. 1980 The Emperor's Old Clothes, Turing Award Lecture. *Communications of the ACM* 24 (2), (February 1981): pp. 75-83

[5] Hoare, C. A. R. 2009 Null References: The Billion Dollar Mistake, QCon 2009

[6] Munroe, A. 2012 PHP: a fractal of bad design, April 9, 2012 http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/