Appendix A

Programming language metrics

Timur Fayzrakhmanov, Innopolis University
v0.4 (updated 22 Dec, 2022)

This document is a collection of over 70 unique *programming language metrics*. The purpose of this document is to provide *dimensions* (features, properties, aspects) by which any two computer languages can be qualitatively and quantitatively compared. These metrics can be used to analyse languages, define requirements, create rankings, provide tips for language designers, or simply give a bird's-eye view on existing language features. The list is based on metrics commonly used in programming language research, development and use, as well as the years of author/contributors personal experience. This appendix is a part of an article "Introducing Programming Language Metrics" written by Timur Fayzrakhmanov for *Proceedings of the Institute for System Programming of the Russian Academy of Sciences* (Proceedings of ISP RAS).

Contribution

This document is open-source. To contribute new metrics, typo fixes, or suggest any other improvements, please send an email to `tim.fayzrakhmanov@gmail.com` or make a pull request/open issue at `https://github.com/timfayz/language-metrics`. Please, specify your full name, public email, and affiliation if necessary. Full list of contributors can be found at the end of this document (Section Acknowledgement).

Legend

Metrics are grouped into nine basic categories:

1. *User experience* – a user background affecting the language use
2. *Language recognizability* – how popular the language is
3. *Language infrastructure* – surrounding documentation and libraries
4. *Language development and support* – maintenance, user support, and tooling
5. *Language special features* – coding experience and special-purpose features
6. *Language implementation and programs* – compiler and its generated executable files
7. *Language specialization and design* – focus and syntactic/semantic design decisions
8. *Language definition* – specification, formalization, and standardization
9. *Language origin* – by whom, when, and why the language was originally conceived

Each metric has an ordinal number, name, indicators for measuring score, and examples of a user feedback. The order of categories and metrics within is by potential 'impact factor' for an *ordinary end-user* rather than by the impact factor for a potential language designer.

`First column` contains:

1. **Metric name** with a polarity sign: ↑ 'higher or support is better', ↓ 'lower or absence is better', and ○ 'neutral or depends'
2. *Feature names* found in the advertising descriptions of languages (should be read as "*Language is / has / supports ...*")
3. Typical examples of languages with a good demonstration score (based on public information, author/contributors experience, with no supporting references)

`Second column` contains a set of indicators for measuring metric 'score'. If many, indicators can be added together or used individually to adjust the desired accuracy.

`Third column` describes typical positive '`+`' or negative '`-`' end-user perception (usually emotional ones) that have been found "in the wild" (forums, comments, contributors/author experience). Sometimes we put content of the third column in the second (after a long dash '—') to save some vertical space.

	Metric name$^{(\text{polarity})}$ *Feature name* Typical representative	**How to measure** Common ▸ indicators for measuring metric score	**Typical end-user perception** Positive + or negative − comments found in the wild, when the score is high/low in the metric's polarity
	User Experience		
1.	**Familiarity**$^{\uparrow}$	▸ *N* of years coding in the language	+ "It is easier to code in because I already know the language"
2.	**Similarity**$^{\uparrow}$ C, C++, C# Pascal, Modula, Oberon	▸ Language is similar to other languages known by the user	+ "The language is really easy to grasp because it looks similar to others"
	Language Recognizability		
3.	**Popularity**$^{\uparrow}$ *Popular* *Mainstream* *Rich set of libraries* *Rich community support* Python, JavaScript	▸ Rank of the language in popularity ranks/surveys: TIOBE[1], PYPL[2], IEEE Sepectrum[3], StackOverflow Survey[4], GitHub's State of Octoverse[5] If manually (the order reflects an ease of checking): ▸ Wiki page is available ▸ Is in 'Popular' category at GitHub's search[6] ▸ Reddit community is available + *N* of members ▸ *N* of questions at StackOverflow[7] ▸ *N* of packages at GitHub[6] ▸ *N* of references/tutorials in web searches ▸ *N* of books written ▸ YouTube videos are available ▸ Job openings are available	+ "Language must be safe to learn because a lot of people already use it and there must be a reason for it" + "Language must be actively developed* and its development won't be abandoned soon" + "There are plenty of tutorials, examples, snippets, answers to get started" + "It's probably easier to find a job" * language development might be stagnating even if it is still actively used or considered popular. That is why we included 'Development' as a separate metric
4.	**Trendiness**$^{\uparrow}$ *Trendy* *'Rising star'* Haskell, Python, Rust	▸ *N* of stars in public repository compared to the date of the project inception ▸ Language has a surge of interest in newsgroups, conference talks and media channels	+ "The language seems promising. If I start using it now, it may payoff in the future (new jobs, niches, technological advantage)"
	Language Infrastructure		
5.	**Documentation**$^{\uparrow}$ *Easy to read* *Comprehensive* *Full of examples* PHP, C#, Go	▸ Language has 'official documentation', 'reference manual', or 'programmer's guide' that: ▸ Clearly describes how to get started ▸ Written in a clear/informal manner ▸ Has a wide coverage ▸ Contains illustrative code samples (examples) ▸ Well linked with other parts of documentation ▸ Loads quickly	+ "With good examples in documentation I can easily start prototyping my own project" + "I can easily find an answer to any of my question concerning the language" − "It is almost impossible to use and learn language without a well-written documentation"
6.	**3rd-party Resources**$^{\uparrow}$ *Rich community support*	▸ *N* of textbooks available (for various kind of users; from novices to experienced developers) ▸ Online resources: tutorials, articles, posts ▸ Q&A websites ▸ Videos	+ "It is great when language has a lot of additional resources, tutorials, etc. that explain the same language from different angles, and for different users"
7.	**Standard Library**$^{\uparrow}$ *Rich/Clean stdlib* *'Batteries included'* Go, Python, Java, C++	▸ *N* of packages available in standard library ▸ Language is following exhaustive vs minimalistic standard library approach	+ "Rich standard library means I can build a lot without switching to unreliable 3rd-party libraries that might be buggy or become unmaintainable"
8.	**3rd-party Libraries**$^{\uparrow}$ *Rich ecosystem* JavaScript, Python, C++	▸ *N* of packages available on GitHub or language's own repository network	+ "The more packages available in the wild, the faster I can create my own solution, just by using someone else's work"
	Language Development and Support		
9.	**Development**$^{\uparrow}$ *Actively Developed* Python, C++	(the overall language development dynamics) ▸ How recently was the stable release ▸ *N* of releases per month/year ▸ *N* of commits per month/year	+ "If the language is actively developed, then it's not going to *'die'* soon, and so we can rely on it" + "Bugs reported in the previous version(s) are to be fixed in the next one"

10.	**IDE support**↑ *Supported by many IDEs* Java	▸ *N* of 3rd-party IDEs supporting the language IDE support = syntax highlight, syntax checker, code formatter, auto-completion, refactoring, code search, debugging, linter, etc. (each feature gives 'point')	+ "I can use language in my favorite IDE" - "Without IDE support (like syntax, error highlighting, autocompletion, and such), the modern use of language is almost impossible"
11.	**Milestone**↑ *Stable*	▸ Language has reached version 1.0 (ie. its library API, syntax, and language constructs became fixed)	+ "Language API isn't in complete flux, so we can rely on it without worrying of breaking changes in the next update"
12.	**Backward-compatibility**° *Backwards-compatible* C++, JavaScript	▸ Every new release keeps language API, syntax, and language constructs backward compatible with previous release(s)	+ "My codebase can rely on the API it was originally written in and yet keep updating compiler for possible performance improvements"
13.	**Technical support**↑ *24/7 Technical support*	▸ Language provides a service with direct human-based technical support	—
		Language Special Features	
14.	**Garbage collection**° *Automatic memory management* Go, Java, Python, C#	▸ Language provides a garbage collector (GC) —— + "Language takes care of my resources so I don't need to think about manual memory allocation and deallocation"	- "Programs in the language with automatic memory control are memory hungry and probably cannot be used for embedded systems" - "Language does not give me manual memory control to do my own (unsafe) stuff"
15.	**Type safety**↑ *Strong typing* *Static type-checking* Rust, Go, Haskell	▸ Language provides any form of runtime or/and compile-time type checking (ie. prevents a program to perform illegal operations on values that do not have appropriate data type)	+ "Programs written in this language are reliable, less error-prone, and always behave the way I defined them to behave" - "I am so annoyed with constant type checking errors that I simply cannot write programs in it"
16.	**Memory safety**↑ *Safe/Memory-safe* Rust, Go, Kotlin	▸ Language provides any form of mechanisms to prevent illegal memory access in a program (runtime/compile-time checks for buffer/stack overflows, dangling pointers, double freeing, etc.)	+ "Programs written in this language are more safe and less prone to memory leaks"
17.	**Type richness**↑ *Rich types* Haskell, Scala, Typescript	▸ Language has high descriptive power in its type system (eg. support for interfaces, generics, algebraic, high-order, dependent types, etc.)	+ "Language allows me to define complex types, data and program behaviour as well as verify them prior execution"
18.	**Exception handling**↑ *Exception handling* C++, Python, Java	▸ Language provides mechanisms for handing unexpected runtime errors without immediate crash / resuming execution	+ "Language allows me to handle runtime errors such that I am able to recover execution flow or exit properly"
19.	**Concurrency**↑ *Parallel computing* *Multithreaded* *Coroutines* C/C++, Go, Erlang	▸ Language supports any form of parallel execution and multithreaded computing: ▸ Heavyweight threads (also native, OS threads) ▸ Lightweight coroutines (also fibers, generators, 'green threads')	+ "Language allows me to do parallel computing (ie. utilize as much computing power as possible) in a manageable way"
20.	**Instruction-level parallelism**↑ *Parallel computing* *SIMD programming* C/C++	▸ Language supports any form of vectorized operations or 'SIMD programming' (eg. explicit directives for vectorized/'streaming' data structures, operations, loop unrolling, etc.)	+ "Language allows me to do professional optimization of my code to get the maximum performance and efficiency of my programs"
21.	**GPU computing**↑ *Parallel computing* *Scientific computing* C/C++	▸ Language provides well-supported libraries or primitives to dispatch execution onto GPU(s)	+ "Language allows me to accelerate my programs with the power of GPU"
22.	**Distributed computing**↑ *Distributed computing* C/C++, Julia, Erlang	▸ Language provides mechanisms to distribute a single program or execution flow upon several physically separated machines (incl. separated by network)	+ "Language allows me to do highly scalable computation across multiple machines"
23.	**Message passing**↑ *Distributed computing* Erlang, Smalltalk, Java	▸ Language supports sending messages between abstract objects which can be objects, parallel processes, subroutines, functions or threads	+ "Language gives me a single model of objects that simply communicate with each other, no matter whether they are functions or parallel processes"
24.	**Reflection**↑ *Reflective* Go, Julia, JavaScript	▸ Language provides constructs to 'see' and modify its own code (normally, at runtime; eg. accessing variable names, function signatures, etc.)	+ "I can access meta-data of classes, interfaces, fields or methods at runtime without knowing their names at compile time, which allows me to write much more generic code and do all kinds of static/dynamic code analysis"

25.	**Lazy evaluation**$^{\circ}$ Haskell, Io, Clojure, Scala	▸ Language supports holding up the evaluation of an expression until its value is needed ▸ Language allows switching back to or explicitly forcing (normal) 'eager evaluation' when needed ——— + "In lazy language it is possible to define infinite lists and elegantly handle streams of data"	+ "My code can be more efficient in terms of memory and performance because values don't need to be computed if they aren't going to be used" − "Lazy evaluation brings a certain amount of memory bloat, and requires too much knowledge of the program and algorithms to get the benefits" − "It is not clear when exactly side effects are going to happen and so it is hard to debug"
26.	**Lambda expressions**$^{\uparrow}$ Haskell, Scheme, many...	▸ Language supports anonymous functions	+ "I can construct higher-order functions or use them as values to return from other functions"
27.	**Package manager**$^{\uparrow}$ *Package manager* C# NuGet, Python pip	▸ Language allows to download and install packages and dependencies using one of its (built-in) CLI commands	+ "Language comes with its own package manager so I don't need to install some 3-rd party packages to get things up and running"
28.	**Doc generator**$^{\uparrow}$ *Doc comments* Java, C#	▸ Language supports 'documentation comments' (formatting tags) and is able to generate (HTML) pages based on these annotations	+ "I can embed parts of program documentation directly into my source code and get nice-looking pages for free"
29.	**Build system**$^{\uparrow}$ *Native build system* Zig	▸ Language allows to write build scripts in itself without using external tools or other languages (such as Bash, make, CMake, Maven, etc.)	+ "It is great that I don't need to learn other building tools and their cryptic languages in order to automate my project building routines"
30.	**Error hints**$^{\uparrow}$ *Smart compiler* *Helpful debug messages* Elm	▸ Language compiler or run-time environment provides error messages that are instructive enough to understand and to fix them	+ "Language is really good in helping to fix my code. I get not only an error message but also a hint how to fix it"
31.	**Code formatting**$^{\uparrow}$ *No more formatting wars* Go fmt, C clang-format	▸ Language compiler can automatically reformat code to follow default/user-defined coding standards	+ "I don't need to spend time following numerous and over-complicated coding styles to format my code. Let the language to do it instead"
32.	**Macros**$^{\uparrow}$ *Metaprogramming* C/C++, Zig, Nim	▸ Language supports any form of metaprogramming or defining 'macros' to execute logic during compile time	+ "I can do a lot of prepossessing during compile time so that runtime is not occupied by unnecessary computations"
33.	**Native IDE**$^{\uparrow}$ *Built-in IDE* Eiffel → EiffelStudio	▸ Language offers its own integrated development environment ——— + "Native IDE may give much better integration than 3rd-party alternatives"	− "I don't want to change my environment just because of the language" (if only native IDE available) − "It's unlikely that build-in IDE is better than my current"
34.	**REPL**$^{\uparrow}$ *Interactive* Python, Scala	▸ Language has interactive Read–Eval–Print-Loop mode	+ "It is easy to play with the language and test code snippets"
35.	**Embedding**$^{\uparrow}$ *Embeddable* Lua, Tcl, Red, Lisp	▸ Language (as 'guest') can run in N of ('host') languages or applications	+ "Language can be used as a *scripting* language to automate repetitive tasks in my favorite application or other host language (eg. Bash in shell, Python in Blender, Lua in World of Warcraft)"
36.	**Bindings**$^{\uparrow}$ *FFI support** Python/Go ← C/C++ Kotlin ↔ Java	▸ Language supports direct function calls (bindings) of a specific or N other languages without wrappers or special API *FFI (Foreign Function Interface) – a language is capable to call functions written in another language providing so called 'bindings' (primarily to C)	+ "My code can easily use the libraries of other language(s)"
37.	**Transpilation**$^{\uparrow}$ *Transpiled* Haxe, TypeScript, Elm	▸ Language is able to compile code into source code of other N (high-level) languages	+ "I can keep writing my code in the language to the benefits of which I already get used to but also benefit from other language(s) infrastructure, libraries, performance, etc."
38.	**IR access**$^{\uparrow}$ *Open interface* *Deep language integration* C# or VB (using Roslyn)	▸ Language, for each compilation step, provides internal intermediate representation (IR) export (eg. pre-processed source code, parse tree, syntax tree, intermediate code, etc.)	+ "Probably language provides a good amount of data for implementing advanced IDE features (debuggers, static analyzers, code formatters, dependency checkers, visualizers, etc.)"
39.	**Unicode support**$^{\uparrow}$ *UTF-8 support* Java, C#, Go, Swift	▸ Language supports Unicode Standard for representing characters in strings or identifiers	+ "I can work with special characters such as emoji in my strings or use foreign language identifiers"

40.	**GOTO support**$^{\circ}$ C/C++, Go, Fortran	▸ Language supports '`goto`' statements for unconditional jumps to specific program locations (usually by means of labels)	+ "I can create custom control structures where the built-in ones does not satisfy my (professional/low-level) needs" – "GOTO statements can be easily abused by unskillful programmer and lead to notorious Spaghetti code"
	Language Implementation and Programs		
41.	**Compilation speed**$^{\uparrow}$ *Fast compilation* C, Go, Zig	▸ How fast compiler compiles programs in s/ms/ns	+ "Recompilation time in this language is really short, which allows me to make the feedback loop between code changes and results short"
42.	**Runtime speed**$^{\uparrow}$ *Fast* C/C++, Rust, Zig	▸ How fast programs run in s/ms/ns	+ "Language is blazingly fast, programs written it run really quickly"
43.	**Compile-time memory footprint**$^{\downarrow}$ *Low memory usage* C, Pascal, Forth	▸ The amount of memory in bytes needed to compile a program (or *while* compiling the program)	+ "I can compile big projects without thinking that I will run out of memory on my machine"
44.	**Runtime memory footprint**$^{\downarrow}$ *Low memory footprint* C/C++, Fortran, Rust	▸ The amount of memory in bytes that a program uses *while* running	+ "Programs written in this language hardly use any RAM (compared to others), which means the compiler performs good optimizations, emits efficient code and probably suitable for embedded systems"
45.	**Compiler/VM size**$^{\downarrow}$ *Lightweight* Lua	▸ Size of language compiler or VM in LOC/bytes	+ "Language is lightweight, minimalistic and (possibly) embeddable"
46.	**Executable size**$^{\downarrow}$ *Compact programs* *Slim binaries* C, Oberon, Zig	▸ Size of executables, including the ones for VM, in bytes (eg. with default compiler options)	+ "Programs are small, possibly fast, and may fit into embedded systems"
47.	**Compiler/VM portability**$^{\uparrow}$ *Portable* C/C++, Java	▸ *N* of platforms the language compiler/VM can run on	+ "I can compile my code on many platforms" or "I can run compiler/VM on many platforms"
48.	**Executable portability**$^{\uparrow}$ *Cross-compiled* *Portable, Transpiled* C/C++, Java	▸ *N* of 'target platforms' the language programs can be run on	+ "I can write code once and run it anywhere (WORA)"
49.	**CLI complexity**$^{\downarrow}$ *Simple to use* Go	▸ *N* of `$ language command`s ▸ *N* of command line `--options`	+ "Command Line Interface of the language is easy and simple to use and remember"
50.	**Self-hosting**$^{\uparrow}$ *Self-hosted* Zig, Go, Rust	▸ Language implementation is written in itself ——— + "If language is self-hosted, it can be considered 'serious', 'production ready' and independent from others"	+ "Language can get more contributions to its compiler by people who before would only work on the standard library"
51.	**Open-source**$^{\uparrow}$ *Open Source* *OSI-approved* Python, Go	▸ Language (compiler) source code is open-source and available for download, modification, recompilation, distribution, static linking and commercialization	+ "Open-source is good because anyone can contribute to language development: do code reviews, fix bugs, write modules, documentation, etc." – "If open-source, it is not clear who is responsible for the project and fixing bugs. It can be abandoned at any time"
52.	**License**$^{\circ}$ *MIT license*	▸ Language license type (MIT, GPL, BSD etc.)	+ "Nonrestrictive license types give a language freedom to be not confined to any single ownership, and prevent attempts to be company or technology-specific"
	Language Specialization and Design		
53.	**Paradigm**$^{\circ}$	▸ Language presents itself as following a particular or multiple paradigms (eg. procedural, object-oriented, functional)	+ "I like when the language mix different paradigms because I can approach problems using a paradigm that is the most effective for the solution"

54.	**Visual language**$^{\circ}$ *Visual Programming* DRAKON, Scratch	▸ Language has a graphical representation and can be used as a visual modeling or programming language	+ "I like the visual expression of my code to better understand and manipulate my program"
55.	**Esoteric language**$^{\circ}$ Brainfuck	▸ Language is considered as 'esoteric' (esolang)	+ "I can use the language as a form of software art to show off my skills"
56.	**Educational language**$^{\circ}$ Logo	▸ Language is specifically designed or can be used to introduce pure computer science ideas (also known as 'tiny', 'small', or 'first')	+ "I can use the language to concentrate on pure ideas without being distracted with unnecessary infrastructural details"
57.	**Domain-specialization**$^{\circ}$ *Used by professionals* *Hardened by industry* R in statistics Matlab/Python in scientific computations	▸ Language became one of the standard tool used in a certain domain —— - "Language is not safe to invest time because if I use it, I'll stuck in its domain"	+ "Language is safe for time investment because other people in my domain already use it" + "Typical problems have been solved already" + "It will be easier to find a job (or simply, you don't find any without having skills in it)" + "I'll be able to do what other professionals do"
58.	**Platform-orientation**$^{\circ}$ *Deep integration* Apple → Swift Microsoft → C#	▸ Language is primarily driven by or developed for a certain platform and its infrastructure	+ "Language provides the best integration experience for this platform" - "If I use this language I will probably *stuck* in its infrastructure"
59.	**Expressiveness**$^{\uparrow}$ *Expressive, Powerful* Python	▸ Length of program in LOC to express a typical problem comparing to the same task written in another language [8]	+ "Language is easy to write, it is concise, short and elegant; code do not repeat itself" (if ↑, otherwise) - "Language is difficult to write, read, and maintain; code grows fast"
60.	**Syntactic complexity**$^{\downarrow}$ *Laconic, Concise* *Elegant, Simple* Lisp ↓, C++ ↑	▸ N of production rules language grammar has ▸ N of keywords	+ "Language is simple, elegant, concise and has a small learning curve" (if ↓, otherwise) - "Language is bulky, complex, bloated and has a steep learning curve"
61.	**Syntactic coherence**$^{\uparrow}$ *Clean syntax* APL, Brainfuck ↓ Elm ↑	▸ Ratio between word- vs ASCII-based operators, keywords, and constructs ▸ Keywords are in/distinguishable ▸ Use of ASCII in identifiers is not/allowed ▸ Lack/use of underscores in reserved identifiers	- "Code is cryptic, noisy, ripples in eyes and difficult to follow" (if ↑, otherwise) + "Code is clean, consistent and easy to follow"
62.	**Semantic complexity**$^{\downarrow}$ *Simple* Go	▸ N of language constructs ▸ N of built-in operators	+ "The less construct language has, the less I need to remember"
63.	**Semantic coherence**$^{\uparrow}$ *Consistent design* *Easy to learn* *Coherent* Lisp	▸ Language constructs are composable with each other ▸ Language follows a paradigm 'everything is an expression'	+ "Language feels well-designed, coherent, and easy to learn. It has a small amount of constructs, everything is composable with each other, and there are little/no special rules or exceptions"
64.	(Syntactic/Semantic) **Homoiconicity**$^{\circ}$ *Code as data* Lisp, Scheme	▸ Code can be directly interpreted as data (ie. as language built-in structures), and inversely, data can be executed as code	+ "Language feels magical and self-referential" + "I can easily generate programs or do program analysis written in that language"
65.	**Design independence**$^{\circ}$ *Inspired by X* *Designed from scratch* where X is a well-known language	▸ Language design is 'inspired' by other languages, or it is a continuation of 'language family'	+ "If the language is inspired by X, and X wasn't bad, then the new one is going to be at least as good as its predecessor(s)" + "If a language designed from scratch, it is probably fresh and ambitious enough to give a good 'punch' to others"
		Language Definition	
66.	**Specification**$^{\uparrow}$ C/C++, Java	▸ Language has a normative Specification with a complete in-/semi-/formal definition of its form (syntax) and behaviour (semantics) ▸ Specification includes the specification of standard library	- "If specification is too big, the language is probably over-complicated to hold in one's programmer head and so, difficult to learn" + "If specification is simple/short, the language can be probably easily re-implemented or ported to new architectures"

67.	**Standardization**$^{\uparrow}$ *Standardized* C/C++	▸ Specification is based on the consensus of different parties that may include firms, interest groups, standards organizations or governments	+ "It is good that I can have independent compilers for the same code base and switch them if there is performance or development stagnating issues" − "It became huge, bulky and slow-moving because its design is now dispatched to a standardization committee rather than individual(s)"
68.	**Formal syntax**$^{\uparrow}$ SQL, C#, Go, Python	▸ Specification includes the formal grammar of language syntax (normally in EBNF)	+ "I can use it to write a parser for language analysis or as a basis for its reference implementation"
69.	**Formal semantics**$^{\uparrow}$ *Formalized* Standard ML, PL/I	▸ Specification includes the definition of language semantics in some theory or formal system (eg. Set theory+First-order logic, Category theory, etc.)	+ "Behaviour of my programs can be verified with mathematical rigour" + "Language can be used for mission- and safety-critical software systems",
		Language Origin	
70.	**Origin**$^{\circ}$ *Came from X* X is a well-known company or eminent university	▸ Language was born as an academic, industry, or a hobby project ——— + "If the language was born in industry, it is probably battle-tested, pragmatic, and understandable by a normal human being"	− "If the language was born in academia, probably it is *not* well suited for the real industrial software development" + "If it was born in academia, it is well-designed, has a mathematical rigour, formally defined behaviour, and potentially verifiable programs"
71.	**Author**$^{\circ}$ *Designed by X* X is a prominent person	▸ Author name(s) who designed, implemented or gave rebirth to the language	+ "If the author is well-known developer/researcher, then the language should be well-designed too"
72.	**Initial purpose**$^{\circ}$ *Designed for X*	▸ The problem domain the language was originally (historically) designed for	+ "If the language was created for X, then it should probably do it well"
73.	**Age**$^{\circ}$ *Developed since X* X suggests maturity	▸ Date or N of years from the first release or exposition	+ "If the language is developed over many years, then it must be mature, has comprehensive documentation, and vast infrastructure" − "If the language is too old, then it is slow developed, its design overloaded with special cases and exceptions, and it is overall conservative towards new advancements"

Acknowledgement

1. Eugene Zouev, *Innopolis University*, e.zuev@innopolis.ru

References

[1] *TIOBE Programming Community index*. URL: https://web.archive.org/web/20221102095019/https://www.tiobe.com/tiobe-index/ (visited on 04/11/2022).

[2] Pierre Carbonnelle. *PYPL PopularitY of Programming Language index*. URL: https://web.archive.org/web/20221102011203/http://pypl.github.io/PYPL.html (visited on 04/11/2022).

[3] Stephen Cass. *Top Programming Languages 2022 - IEEE Spectrum*. URL: https://spectrum.ieee.org/top-programming-languages-2022 (visited on 04/11/2022).

[4] *Stack Overflow Developer Survey 2022*. Stack Overflow. URL: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies (visited on 04/11/2022).

[5] *The State of the Octoverse — Top Languages*. The State of the Octoverse. URL: https://octoverse.github.com/2019/#top-languages (visited on 04/11/2022).

[6] *GitHub — Advanced Search*. GitHub. URL: https://github.com/search/advanced (visited on 04/11/2022).

[7] *Stack Overflow — Search questions by Tags*. Stack Overflow. URL: https://stackoverflow.com/tags (visited on 04/11/2022).

[8] Mike Mol. *Rosetta Code*. URL: http://www.rosettacode.org/wiki/Rosetta_Code (visited on 04/11/2022).