DOI: 10.15514/ISPRAS-2022-34(6)-5

Introducing Programming Language Metrics

T.R. Fayzrakhmanov, ORCID: 0000-0001-5013-4523 <tim.fayzrakhmanov@gmail.com>
Innopolis University,
1, Universitetskaya Str., Innopolis, 420500, Russia

Abstract. We introduce possibly the first approximation of *programming language metrics* that represent a spectrum over 70 unique and carefully gathered *dimensions* by which any two programming languages can be compared. Based on those metrics, one can evaluate her own "best" language, and to demonstrate how complex feelings such as "simplicity" and "easy to use", often found as arguments in language debates and advertisements, can be decomposed into clear and measurable pieces. We put the collection as a completely separate open-source file (here as an appendix) so that everyone can participate in eliciting new and interesting dimensions used in programming languages research, development, and use. Metrics can find their use to compare languages, define requirements, create rankings, give tips for language designers, and simply provide a bird's-eye view on existing languages features found in the wild.

Keywords: programming languages; metrics; comparison; analysis

For citation: Fayzrakhmanov T.R. Introducing Programming Language Metrics. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 6, 2022. pp. 67-84. DOI: 10.15514/ISPRAS-2022-34(6)-5

Acknowledgements. We thank Eugene Zouev (Innopolis University), Nikolay Shilov (Innopolis University), Nikolay Kudasov (Innopolis University), and an anonymous reviewer from Ivannikov ISP RAS for providing useful suggestions, additional resources and corrections.

Система метрик для языков программирования

Т.Р. Файзрахманов, ORCID: 0000-0001-5013-4523 <tim.fayzrakhmanov@gmail.com>
Университет Иннополис,
420500, Россия, г. Иннополис, ул. Университетская, д.1

Аннотация. Мы представляем, возможно, первое приближение метрик языков программирования, которые представляют собой спектр из более чем 70 уникальных и тщательно собранных измерений, по которым можно сравнивать любые два языка. Основываясь на метриках, человек может самостоятельно определить "лучший" для него язык и продемонстрировать, как сложные чувства, такие как "простота" и "легкость в использовании", часто встречающиеся в продвижении и спорах о том какой язык лучше, могут быть разложены на четкие и измеримые части. Мы разместили коллекцию в виде отдельного файла с открытым исходным кодом (здесь в качестве приложения), чтобы каждый мог принять участие в поиске новых и интересных измерений, используемых в практике, исследованиях, и разработке языков программирования. Метрики могут найти свое применение для сравнения языков, определения требований, создания рейтингов, советов разработчикам языков, а также просто для получения представления о возможностях в существующих языках программирования.

Ключевые слова: языки программирования; метрики; сравнение; анализ

Для цитирования: Файзрахманов Т.Р. Система метрик для языков программирования. Труды ИСП РАН, том 34, вып. 6, 2022 г., стр. 67-84. DOI: 10.15514/ISPRAS-2022-34(6)-5

Благодарности. Мы благодарим Евгения Зуева (Университет Иннополис), Николая Шилова (Университет Иннополис), Николая Кудасова (Университет Иннополис) и анонимного рецензента ИСП РАН за полезные предложения, дополнительные ресурсы и исправления.

1. Introduction

The hot debates in comparison of programming languages have been known for years. "What is the *best* programming language?" – is one of the most frequently asked questions when one encounters a plethora of available options: 100 languages with thousands to millions of active users worldwide up to 8,945 in total (Table 1). Despite such a precious freedom to choose, in practice, leads to the *paradox of choice* where the more options you have, the more time it takes to settle on a final decision.

Scientific community in the analysis and comparison of programming languages have tried to "nail down" this question multiple times giving an *objective* answer to both: "Who is the best?" in general [1, 2] or for a particular area [3, 4]. However, besides inaccessible scientific jargon for an ordinary language user, no single research can cover so many languages with so many purposes at once.

In this work we propose an alternative approach. Instead of searching for an abstractly best language among an ever-growing number of options and realizing in advance that the choice still depends on numerous factors (user preferences, current infrastructure, etc.), we simply suggest collecting *top language aspects*. In other words, the aspects (further "*programming language metrics*") that have always influenced our positive and anti-choices, and proven to be useful over a long period of time.

The definition of *best*, then, (Section 4) will be a simple formula: *subset* of aspects that must be necessarily included in a language plus their *weights* that distribute the consideration importance within.

Табл. 1. Количество языков программирования
Table 1. The number of programming languages

Total	
(the total number of languages ever created)	
8,945	HOPL Historical Encyclopedia (till 2005) [5]
4,217	Programming Language DataBase [6]
Notable	
(languages that are influential or proved their existence)	
878	Rosetta Code's List of Programming Languages [7]
690	Wikipedia's List of Programming Languages [8]
560	Available in GitHub's Advanced search [9]
370	"Primary" in the annual GitHub report [10]
Popular	
("top" languages with thousands to millions of active users)	
52	IEEE Spectrum index [11]
50-100	TIOBE Programming Community index [12]
42	StackOverflow Developer Survey [13]
28	PYPL Popularity index [14]

2. Related works

Our filter out criteria for works in the comparison and analysis of programming languages were studies that (1) directly identify the list of programming languages metrics, (2) describe possible ways for an ordinary user to measure them, and (3) give methods to define "best" in terms of given metrics and measured scores. Within those constraints, we found no previous work. However, [15] written by Jean E. Sammet 50 years ago is the closest published study.

Although Sammet has not provided a generalized list of metrics with methods for applying them in defining the best option, we admit that (1) she stressed the importance of a language assessment to be *dependent* on numerous factors, e.g. on a viewpoint of a user, implementor, or application area [15, p. 245]; (2) even though her work was found *retrospectively*, the method presented in **Metrics use** (Section 4) is already implicitly used in evaluation of languages for "a user wishing to write a payroll program" [15, tables I, V, and VI].

The rest of the studies were primarily focused on:

- Taking a limited number of metrics (e.g., "running time" or "memory usage") and evaluating the *best* language among the existing few. For example, within a specific area (bioinformatics [3], robotics [4], economics [16]) or generically ([1, 2, 17-19);
- Introducing a *specific* metric/s (e.g., "popularity", "impact" [20], "syntactic complexity" [21], "structural complexity" [22]) without further *generalization* with other metrics.

In this work, we considered all types of studies to make the table of metrics as complete as possible.

3. Method

In collecting metrics, we followed no particular method or order. We tried to scrutinize as much literature and resources as we could, which can effectively cooperate in eliciting useful and easy-to-distinguish metrics from an end-user perspective. Besides referenced literature, it involves taking a list of notable languages (Table 1) going through the websites of each, reading the advertising text, specifications, language references, and they-provided comparison with other competitors.

3.1 Terminology

There are many words to describe *dimensions* by which objects, be they programming languages or apples, can be *differentiated* with each other. To make our mappings between words and meaning clearer throughout the text, we want to explicitly distinguish the following terms:

- ***Dimension*** (*aspect, property, indicator, attribute, characteristic, criteria, parameter) –* a *quality* associated with an object
- ***Metric*** – a set of qualities *and* a method of quantifying (measuring) it
- ***Feature*** – a quality that beneficiary distinguishes one object *from others* in its class

We use "dimension" (and its synonyms) to signify the most atomic aspects of a language, "metric" as a set of dimensions that can be represented *numerically*, and "feature" to simply provide a colloquial language used in languages advertising or keywords to search solutions in the web.

Our terminology implies that *quality* of an object is taken for granted and means what common sense suggests us. *Quantity* represents the state of being in a certain quality. For example, "five apples" can be considered as immeasurable qualitative state of having "five" (not used here) or as a *measurable quantitative* state of being in 5 pieces. Binary states (used in the metrics of a type "language supports *X*") are also accounted as measurable quantitative states of the amount of two ("red apple" is 1 and lacking the "red" is 0).

3.2 Metrics vs. features

A common practice to think of any consumer product, which we believe any programming language essentially is, is in terms of "features". Features can be considered as a *common language* that are used by both product creators in advertising as well as end-users in product perception.

In this work, it was tempting to present neutrally-oriented list of metrics in terms of features as it is the most popular way for a computer language to be promoted and picked up in the wild. However, if we decided to do so, the table would have become *suggestive*, implying that those features are rather *requirements* for a "perfect language" we are seeking for, than the *dimensions* by which we simply want to compare. Table 2 shows the difference. Thus, we decided to keep the list neutral and add features (whenever possible) right under the metric names to simply provide an additional information.

Табл. 2. Разница между метрикой и «фитчей» языка
Table 2. The difference between a language metric vs. feature

Language Metric	**Language Feature**
Definition and Purpose	
A *method* to measure (quantify) a quality of an object in a neutral manner	A *quality* of an object that demonstrates an advantage(s) over others in its class
The main difference	
A metric cannot be introduced without a method of measuring it	A feature, similar to a feeling, can be introduced even if there is no clear way to measure it
Examples	
Performance (mostly nouns)	*Fast* (mostly adjectives)
Number of seconds required to compile and/or run a program	Whatever the numbers are, it feels really fast comparing to others
Compiler size	*Lightweight*
Lines of code or size in bytes of a language compiler or VM	We may not know exactly but the language weights really nothing comparing to others

4. Result

We introduce the full list of metrics in Appendix A. We have made the appendix *self-contained.* It is a completely separate document with its own description, legend, references, and contributors list. We wanted to make it easily printed, shared, and updated independently of the article itself. As such, some of the parts that are already present in this article might be duplicating in the appendix.

4.1 Disclaimer

We do not pretend the list to be *complete* nor we believe it is reasonable to do so. As the field progresses, there will be always new unique ways to measure language aspects, similar to those of software metrics. The attempt is to make at least a *first* approximation of what programming language metrics might be, what one can measure in general, and how they can be used in practice.

5. Metrics use

In this section, we introduce a simple method of how to define and compute your "best" language using a simple table, the list of metrics, and the measurement data.

5.1 Background

Being able to compare similar objects around us and picking the "best one" among available options is one of the essential cornerstones for an *effective* everyday life. Being able to compare *programming languages* and choosing the best one for a particular problem is probably the essential cornerstone for an effective *programmer* life.

Languages are often advertised and perceived in terms of intuitive *feelings* such as "simple" (Python [23]), "fast" (C [24]), "delightful" (Elm [25]) or even "magical" and "sacred" (LISP [26]). Those feelings, collectively, make us prefer one language over another and, thus, shape our *favorite choices*. However, when it comes to the precise definition of what those feelings actually *mean*, how they can be assessed in practice, and how they affect our final choice(s), the details always elude from the scene. Programming languages are very versatile inventions, and to understand why the same language can be considered as the *best* for one and the *worst* for another, we need a *method* of dismantling complex feelings, features, and the notion of "best" into something that can be effectively *measured*.

5.2 Procedure

To demonstrate the method, we will be using a simple example. Suppose our goal is to find the "best fit" language out of the three: L1, L2, L3 (names are chosen deliberately abstract to eliminate language affections). The question is "How do we know what language is the best among selected?" To do so, we first identify *dimensions* by which they can be compared.

Step 1. *Skim through programming language metrics and pick the ones that "feels" right, essential, or important*

Suppose we selected Popularity, Documentation, Standard Library, Performance, and Expressivity (further as Popl, Docs, Stdlib, Perf, and Expr for brevity). Then,

Step 2. *Create a table where rows are languages and columns are metrics*

In our case, the table will look like the following:

	Popl	**Docs**	**Stdlib**	**Perf**	**Expr**
L1	-	-	-	-	-
L2	-	-	-	-	-
L3	-	-	-	-	-

Before we start fulfilling the table,

Step 3. *Distribute the importance (weights) per each of the metric*

We do so *before* fulfilling the table because metric importance affects not only the computing procedure for the final choice but how *carefully and precisely* should we measure the scores. For example, according to some ranking, we may find out that L1 has 2nd place in Popl, and L2 – 20th. If we decided a place in Popl isn't *that* important for us, we may no longer waste our time trying to refine the scores by other rankings, we can simply move on to measuring something else that is *more* important. So, let us say we decided to make the distribution as follows:

	Popl	**Docs**	**Stdlib**	**Perf**	**Expr**
Weight	5%	15%	30%	40%	10%
L1	-	-	-	-	-
L2	-	-	-	-	-
L3	-	-	-	-	-

As we can see the sum of all weights is equal to 100%. In practice, however, when we, say, have 15 metrics, it becomes difficult to distribute importance manually to sum them back to 100%. Instead, we suggest simply giving metrics a "place" or "points" (say, from 1 to 10), and compute the corresponding percentages automatically. For example:

	Popl	**Docs**	**Stdlib**	**Perf**	**Expr**	
Weight	2	4	5	8	3	
Sum:	2 +	4 +	5 +	8 +	3	= 22
Normalize:	2/22	4/22	5/22	8/22	3/22	
Weight (result):	=0.09	=0.18	=0.23	=0.36	=0.14	
In %	9%	18%	23%	36%	14%	

Despite using points, we were able to "normalize" them back to percentages so that they sum up to 100% and roughly correspond to our previous manual distribution.
As a result, the table may look as follows:

	Popl	**Docs**	**Stdlib**	**Perf**	**Expr**
Weight	2	4	5	8	3
In %	9%	18%	23%	36%	14%
L1	-	-	-	-	-
L2	-	-	-	-	-
L3	-	-	-	-	-

The second (grayed out) row is optional and can be removed completely. However, we recommend to keep it and, with the help of spreadsheet software, use it to "interactively" adjust points so that the computed weights in percentage looks *desirable*.

Step 4. *Measure metric scores for each of the language and fulfill the table. Make sure all scores have a numeric value*

Step 4 must be the most difficult and important one as everything else depends on its data. However, measurements details are out of scope of this article. We will simply assume we were able to get the results that are more or less "accurate":

	Popl	**Docs**	**Stdlib**	**Perf**	**Expr**
Weight	2	4	5	8	3
In %	9%	18%	23%	36%	14%
L1	2pl	7p	82pkg	3.1ms	300L
L2	20pl	3p	117pkg	1.7ms	155L
L3	13pl	5p	63pkg	0.5ms	170L

We do not need to fit our metric scores into a particular system of units or scale. Scores can be completely "raw". What is important is that they are *numerical*. For example, Popl can be a place in some ranking as TIOBE [12]; Docs can be a sum of abstract points (say, +1 for coverage, readability, nice-looking, etc.); Stdlib – a number of available packages in it; Perf – time in milliseconds needed to run a test-bench program; and Expr could be the lines of code for the program we run in Perf.

Step 5. *Set the polarity for each of the metric (e.g., "higher is better" or "lower is better", where binary metrics are not the exception) and place them on a separate line or near the names*

We used higher ↑ and lower ↓ is better at the end of the names:

	Popl↓	**Docs↑**	**Stdlib↑**	**Perf↓**	**Expr↓**
Weight	2	4	5	8	3
In %	9%	18%	23%	36%	14%
L1	2pl	7p	82pkg	3.1ms	300L
L2	20pl	3p	117pkg	1.7ms	155L
L3	13pl	5p	63pkg	0.5ms	170L

Before we continue, we may do an additional step:

Step 5.1 (optional) *Identify best scores per each of the column, and write them out on a separate "Best [score]" line*

This step is completely optional and serves rather as an intermediate phase. It shows how close *visually* (by counting highlights) each language approximates to the best sampled scores:

	Popl↓	**Docs↑**	**Stdlib↑**	**Perf↓**	**Expr↓**
Weight	2	4	5	8	3
In %	9%	18%	23%	36%	14%
Best	**2pl**	**7p**	**117pkg**	**0.5ms**	**155L**
L1	**2pl**	**7p**	82pkg	3.1ms	300L
L2	20pl	3p	**117pkg**	1.7ms	**155L**
L3	13pl	5p	63pkg	**0.5ms**	170L

It's time, however, to calculate how *actually* close each language approximates to the best sampled scores considering the weights. In other words, "Who is the best?" among our three. Depending on how we would define "best", we might have two strategies. First is to compute proximity relative to the *unreal* best scores (0th place in Popl and 0ms in Perf). Second – relative to the *real* "Best" scores taken from the previous table (2nd place in Popl and 0.5ms in Perf). We will take the first strategy and remove "Best" line altogether. We do so because (1) taking the second strategy doesn't change the order of "winners", (2) we found it simpler to compute, and (3) we want the highlighted line, which is now is used by "Best", to be taken by the real winner (L1, L2, or L3).

Step 6. *Calculate the final score for each of the languages using the following algorithm 1:*

```
For each row [language]:
   For each column [metric in a language]:
      Take score value $v$
      Take maximum value in column $max$
      Take metric weight $w$
      If column polarity is 'higher is better':
         Compute $v/max \times w$
      Otherwise ('lower is better'):
         Compute $|v/max - 1| \times w$
         Add the result to language score $S$
   [After all metrics are processed]
   Language score $S$ is ready
   Put $S$ on a separate column 'Score'
[After all languages are processed]
Evaluation is completed
Best language is the one with the biggest $S$
```

Algorithm 1. Best Language Evaluation

Procedure visually:

	Popl$^{\downarrow}$	Docs$^{\uparrow}$	Stdlib$^{\uparrow}$	Perf$^{\downarrow}$	Expr$^{\downarrow}$	Score
L1	2pl	7p	82pkg	3.1ms	300L	0.42
	|2/20 – 1| ×0.09 =0.081 +	7/7 ×0.18 =0.18 +	82/117 ×0.23 =0.16 +	|3.1/3.1 – 1| ×0.36 =0 +	|300/300 – 1| ×0.14 =0 =	0.421
L2	20pl	3p	117pkg	1.7ms	155L	0.54
	|20/20 – 1| ×0.09 =0 +	3/7 ×0.18 =0.077 +	117/117 ×0.23 =0.23 +	|1.7/3.1 – 1| ×0.36 =0.16 +	|155/300 – 1| ×0.14 =0.068 =	0.535
L3	13pl	5p	63pkg	0.5ms	170L	**0.64**
	|13/20 – 1| ×0.09 =0.032 +	5/7 ×0.18 =0.128 +	63/117 ×0.23 =0.124 +	|0.5/3.1 – 1| ×0.36 =0.3 +	|170/300 – 1| ×0.14 =0.06 =	0.644

Procedure formally:

$$S_{\text{best}} \begin{cases} S_1 = m_1[w_1] + m_2[w_2] + \ldots + m_j[w_j] \\ S_2 = m_1[w_1] + m_2[w_2] + \ldots + m_j[w_j] \\ \vdots \\ S_n = m_1[w_1] + m_2[w_2] + \ldots + m_j[w_j] \end{cases},$$

where

$$m_j = \begin{cases} \dfrac{v_j}{max} & \text{, if polarity is "higher is better"} \\ \left|\dfrac{v_j}{max} - 1\right| & \text{, otherwise ("lower is better")} \end{cases} \qquad (1)$$

and

- S_{best} is the best language score among *n* languages;

- S_n is the final score for the language *n*;
- w_j is the weight (importance) of the metric *j*;
- m_j is the computed score value of the metric *j*;
- v_j is the "raw" score value of the metric *j*;
- *max* is the biggest numeric value for *j* among $S_{1\ldots n}$.

We call (1) as the "Formula of Choice". We read it as following: the best language S_{best} among available $S_{1\ldots n}$ is the one which has the biggest sum of metric scores $m_{1\ldots j}$ given their weights $w_{1\ldots j}$. So, after we have computed all the language scores, we can finalize our table with:

Step 6. *Sort the table by Score in descending order, and add a Place column enumerating languages from 1*

	Popl$^{\downarrow}$	Docs$^{\uparrow}$	Stdlib$^{\uparrow}$	Perf$^{\downarrow}$	Expr$^{\downarrow}$	Place	Score
Weight	2	4	5	8	3		
In %	9%	18%	23%	36%	14%		
L3	**13pl**	**5p**	**63pkg**	**0.5ms**	**170L**	**1**	0.64
L2	20pl	3p	117pkg	1.7ms	155L	2	0.54
L1	2pl	7p	82pkg	3.1ms	300L	3	0.42

Column "Place" will give us an ability to keep/see language places even if we decide to sort the table by other columns (e.g., shuffle languages by the largest number of packages).

5.3 Discussion

When we originally looked at the data, we were expecting L2 to become our "top" language. However, the calculation gave it the 2nd, which made us suspect an error in calculations. After a closer look (and double-checking estimates), we understood that L2 is simply *3x times slower* than the winner in Perf, which we decided to be the *most* important aspect in the table. Even though, Stdlib of L2 is larger *almost twice*, its importance is still *lower*. When it comes to the rest of the metrics, they seemed simply compensating each other: L2 is slightly better at Expr, however, slightly worse at Docs, whereas Popl felt completely discarded due to its very low importance.

These slightly unexpected results led us to draw the following conclusions:

1) Weights have to match the *actual expectations* of the author (originally, they have been put artificially without author's internal agreement).
2) Even if weights were in a perfect harmony with us, such cases cannot be excluded, which would require us to start *metric refinement*. The latter means what we have said at the very beginning – the more important metric is, the more effort one should invest into its score elicitation.

5.4 Metric composition

For the sake of simplicity, at the very beginning of the subsection 4.2 (Step 1 and 3), we used metric only as *independent* variables to form our comparison. However, using Step 6 and the "Formula of Choice", we can elaborate the method. We can *compose* metrics as if they are *building blocks* for defining other "high-order" metrics, or (as we are interested in this section) *features* and *feelings*. For example:

"Easy to learn" = Expressiveness$^{\uparrow}\langle 6\rangle$ + Documentation$^{\uparrow}\langle 4\rangle$ + REPL$^{\uparrow}\langle 2\rangle$;

"Easy to write" = Syntactic complexity$^{\downarrow}\langle 5\rangle$ + Code formatting$^{\uparrow}\langle 3\rangle$;

"Easy to run" = Compiler portability$^{\uparrow}$ + Supporting platforms$^{\uparrow}$;

"Easy to debug" = Error hints$^{\uparrow}\langle 7\rangle$ + Compilation speed$^{\downarrow}\langle 3\rangle$;

"Easy to find help" = *max* (Popularity$^{\uparrow}$, Technical support$^{\uparrow}$);

"Fast" = Runtime speed$_{\text{Rust}}$ / Runtime speed$_{\text{C}}$ (How fast is C relative to Rust)

where

- *"Aspect"* = $\text{metric}_1^p\langle\text{weight}_1\rangle\ op\ \text{metric}_2^p\langle\text{weight}_2\rangle$ ...;
- *"Aspect"* can be a new (composite) metric, feeling, or feature;
- Angle brackets designate *weight* of the metric (in relative units, say, from 1 to 10);
- Lack of brackets means all metrics share the same importance;
- *p* is the polarity of the metric: higher ↑ or lower ↓ is better;
- *op* is how we want to combine metrics to produce new aspect (e.g., by summation, division, taking max, etc.)

For example, we defined an *"Easy to learn"* as a sum of three aspects: Expressiveness of code, available Documentation, and the presence of REPL. The importance within was distributed using relative points to get percentages automatically (as we did in Step 3). In our case, they come to 50%, 33%, 16% accordingly. *"Easy to run"* we defined as the sum of Compiler portability and Supporting platforms. We missed weights, which mean they are distributed equally: 50% and 50%. Finally, *"Easy to find help"* is simply the metric that is best manifested in the language: either Popularity or direct Technical support, where the weight of whichever metric is chosen will be 100%. The rest of examples should be self-explanatory.

These simple rules of composition give us unlimited power in defining feelings, features, and other high-order metrics that would otherwise be difficult to express. We believe that the idea of combining metrics could be an interesting tool for making *sound arguments* in the endless emotionally-driven language debates. This perspective could make metric composition to be uncharted territory for further research and exploration.

6. Conclusion

The current work presents possibly the first approximation of programming language metrics. We provided an open-source document (to which we welcome to contribute) with over 70 unique programming language aspects that can be used to pragmatically compare languages, define requirements, create rankings, and have an overview of available language features. We have presented the "Formula of Choice" to determine "best language" for your own needs using a simple table with few calculations. We have presented the method of *metrics composition* to decompose complex feelings and features, such as "simplicity" and "easy to use", into more measurable pieces.

We hope that this information can serve as a useful guidance for the analysis and comparison of programming languages in the never-ending debates and constantly emerging options.

References / Список литературы

[1] Nanz S., Furia C.A. A Comparative Study of Programming Languages in Rosetta Code. In Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 778-788.

[2] Prechelt L. An Empirical Comparison of Seven Programming Languages. Computer, vol. 33, issue 10, 2000, pp. 23-29.

[3] Fourment M., Gillings M.R. A comparison of common programming languages used in bioinformatics. BMC Bioinformatics, vol. 9, issue 1, 2008, article no. 82, 8 p.

[4] Pembeci İ., Hager G. A Comparative Review of Robot Programming Languages, Report CIRL-Johns Hopkins University, 2003, 29 p.

[5] Pigott D.J., Axtens B.M. HOPL; Online Historical Encyclopaedia of Programming Languages. Available at: https://hopl.info/, accessed Aug. 11, 2020.

[6] Programming Language DataBase. Available at: https://pldb.com/, accessed Aug. 13, 2022.

[7] Programming Languages. Rosetta Code. Available at: https://rosettacode.org/wiki/Category:Programming_Languages, accessed Nov. 04, 2022.

[8] List of programming languages. Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=List_of_programming_languages&oldid=972639589, accessed Aug. 13, 2020

[9] GitHub | Advanced Search.GitHub. Available at: https://github.com/search/advanced, accessed Nov. 04, 2022.

[10] The state of open source software | Top Languages. Available at: https://octoverse.github.com/2022/top-programming-languages, accessed Nov. 16, 2022.

[11] Top Programming Languages 2022. IEEE Spectrum. Available at: https://spectrum.ieee.org/top-programming-languages-2022, accessed Dec. 20, 2022.

[12] TIOBE Programming Community index. Available at: https://www.tiobe.com/tiobe-index/, accessed Dec. 20, 2022.

[13] Stack Overflow Developer Survey 2022. Available at: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022, accessed Dec. 20, 2022.

[14] Carbonnelle P. PYPL PopularitY of Programming Language index. Available at: http://pypl.github.io/PYPL.html, accessed Jul. 16, 2020.

[15] Sammet J.E. Problems in, and a pragmatic approach to, programming language measurement. In Proc. of the Fall Joint Computer Conference, 1971, pp. 243-251.

[16] Aruoba S.B, Fernández-Villaverde J.A Comparison of Programming Languages in Economics. Working Paper 20263. National Bureau of Economic Research, 2014, 20 p.

[17] Boom H.J., de Jong E. A critical comparison of several programming language implementationsю Software: Practice and Experience, vol. 10, issue 6, 1980, pp. 435-473,

[18] Alomari Z., Halimi O.E. et al. Comparative Studies of Six Programming Languages. arXiv:1504.00693, 2015, 71 p.

[19] Al-Qahtani S.S., Pietrzynski P. et al. Comparing Selected Criteria of Programming Languages Java, PHP, C++, Perl, Haskell, AspectJ, Ruby, COBOL, Bash Scripts and Scheme. Revision. arXiv:1008.3434, 2010, 149 p.

[20] Delorey D.P., Knutson C.D., Giraud-carrier C. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects, In Proc. of the Second International Workshop on Public Data about Software Development, 2007, 5 p.

[21] MacLennan B.J. Simple metrics for programming languages. Information Processing & Management, vol. 20, no. 1, 1984, pp. 209-221.

[22] MacLennan B.J. The Structural Analysis of Programming Languages. Report NPS52 81-009, Naval Postgraduate School, 1981, 37 p.

[23] van Rossum G. Python reference manual. Technical Report CS-R9525. NCWI (Centre for Mathematics and Computer Science), 1995, 59 p.

[24] Kernighan B.W., Ritchie D.M. The C programming language. 2nd edition. Pearson, 1988, 272 p.

[25] Elm - delightful language for reliable web applications. Available at: https://elm-lang.org/, accessed Nov. 04, 2022.

[26] G. Steele. Common LISP: the language. 2nd updated edition. Digital Press, 1990, 1029 p.

A. Appendix. Programming language metrics

This appendix is a collection of over 70 unique *programming language metrics*. The purpose of this section 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 analyze 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 and contributors personal experience. The collection is open-source and can be downloaded as a separate PDF file at https://github.com/timfayz/language-metrics.

A.1 Contribution

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 an issue at https://github.com/timfayz/language-metrics. Please, specify your full name, public email, and affiliation if necessary.

A.2 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.

A.3 Metrics table

v0.4 (Updated 22 Dec, 2022)

	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 according to metric's polarity
	User Experience		
1.	**Familiarity**↑	› *N* of years coding in the language	+ "It is easier to code in because I already know the language"
2.	**Similarity**↑ 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**↑ *Popular* *Mainstream* *Rich set of libraries* *Community support* Python, JavaScript	› Rank of the language in popularity ranks/surveys: TIOBE [12], PYPL [14], IEEE Spectrum [11], StackOverflow Developer Survey [13], GitHub's State of Octoverse [10] If manually: (the order reflects an ease of checking)	+ "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"
3.	**Popularity**↑ (cont.) *Popular* *Mainstream* *Rich set of libraries* *Community support* Python, JavaScript	› Wiki page is available › Is in "Popular" category at GitHub's search [9] › Reddit community is available + *N* of members › *N* of questions at StackOverflow [27] › *N* of packages at GitHub [9] › *N* of references/tutorials in web searches › *N* of books written › YouTube videos are available › Job openings are available	+ "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**↑ *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	+ "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**↑ *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 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 my questions concerning the language" – "It is almost impossible to use and learn language without a well-written documentation"
6.	**3rd-party Resources**↑ *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**↑ *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 many applications without switching to unreliable 3rd-party libraries that might be buggy or no longer supported"
8.	**3rd-party Libraries**↑ *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**↑ *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" (if ↓)
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**$^{\uparrow}$ *24/7 Technical support*	› Language provides a service with direct human-based technical support	—
	Language Special Features		
14.	**Garbage collection**$^{\circ}$ *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**$^{\uparrow}$ *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’m so annoyed with the constant type errors that I simply cannot write programs”
16.	**Memory safety**$^{\uparrow}$ *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**$^{\uparrow}$ *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**$^{\uparrow}$ *Exception handling* C++, Python, Java	› Language provides mechanisms for handling 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**$^{\uparrow}$ *Parallel computing* *Multithreaded* *Coroutines* C/C++, Go, Erlang	› Language supports any form of parallel execution and multithreaded computing: › Heavyweight threads (also native, or operating system threads) › Lightweight coroutines (also fibers, generators, or “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**$^{\uparrow}$ *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**$^{\uparrow}$ *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**$^{\uparrow}$ *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**$^{\uparrow}$ *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**$^{\uparrow}$ *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 dynamically (at run-time) access meta-data of language constructs (eg. get a name of a class, variable, function, etc.), which allows me to write a 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 —— + “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”	+ “In lazy language it is possible to define infinite lists and elegantly handle streams of data” – “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 how 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 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 app or a 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) from *N* other languages without wrappers or special API	+ “My code can easily use the libraries of other language(s)” [*]FFI (Foreign Function Interface) – a language is capable to call functions written in another language providing so called “bindings” (primarily to C)
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↑** *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↑** *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°** 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 do not satisfy my (professional/low-level) needs" – "GOTO statements can be easily abused by unskilful programmer and lead to notorious Spaghetti code"
	Language Implementation and Programs		
41.	**Compilation speed↑** *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↑** *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↓** *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↓** *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 does good optimizations, emits efficient code and probably suitable for embedded systems"
45.	**Compiler/VM size↓** *Lightweight* Lua	› Size of language compiler or VM in LOC/bytes	+ "Language is lightweight, minimalistic and (possibly) embeddable"
46.	**Executable size↓** *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↑** *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↑** *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↓** *Simple to use* Go	› *N* of `$ language commands` › *N* of command line `--options`	+ "Command Line Interface of the language is easy and simple to use and remember"
50.	**Self-hosting↑** *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↑** *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°** *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°**	› 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°** *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°** 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°** 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°** *Used by professionals* *Hardened by industry* R in statistics Matlab/Python in scientific computations	› Language became one of the standard tools used in a certain domain —— – "Language is not safe to invest time because if I use it, I'll stuck in its domain" + "I'll be able to do what other professionals do"	+ "Language is safe for time investment because other people in my domain use it already" + "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)"
58.	**Platform-orientation°** *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↑** *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 ↑) – "Language is difficult to write, read, and maintain; code grows fast" (otherwise)
60.	**Syntactic complexity↓** *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 ↓) – "Language is bulky, complex, bloated and has a steep learning curve" (otherwise)
61.	**Syntactic coherence↑** *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 ↑) + "Code is clean, consistent and easy to follow" (otherwise)
62.	**Semantic complexity↓** *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 number of constructs, everything is composable with each other, and there are little/no special rules or exceptions”
64.	(Syntactic/Semantic) **Homoiconicity**° *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**° *Inspired by X* *Designed from scratch* 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**° *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” – “Language has become huge, bulky and slow-moving because its design is now dispatched to (big) standardization committee rather than (small group of) 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**° *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**° *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**° *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**° *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”

Information about the author / Информация об авторе

Timur Rasimovich FAYZRAKHMANOV – software developer, PhD student, researcher. Research interests: programming languages, knowledge representation and processing, knowledge organization and reuse, formalization, generic systems modeling, World Digital Mathematics Library, graphics, alternative development environments, block-based approach.

Тимур Расимович ФАЙЗРАХМАНОВ – разработчик ПО, аспирант, исследователь. Сфера научных интересов: языки программирования, представление и обработка знаний, организация и повторное использование знаний, формализация, обобщённое моделирование систем, Всемирная цифровая математическая библиотека, графика, альтернативные среды разработки, блочный подход.