Method of Programming Languages Analysis

Lidia Gorodnyaya[1,2][0000-0002-4639-9032]

[1] A.P. Ershov Institute of Informatics Systems (IIS), 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia

[2] Novosibirsk State University, Pirogov str., 2, Novosibirsk 630090, Russia

lidvas@gmail.com

Abstract. The purpose of the article is to describe the method of comparison of programming languages, convenient for assessing the expressive power of languages and the complexity of the programming systems. The method is adapted to substantiate practical, objective criteria of program decomposition, which can be considered as an approach to solving the problem of factorization of very complicated definitions of programming languages and their support systems. This method is aimed to issues arising in connection with the problem of measuring the characteristics of languages and programming systems that affect the complexity of software development and the productivity of software applications. Paradigmatic models of languages and programming systems can be useful in systematizing programming languages, assessing their similarities and differences, which allows us to build concise definitions regarding such models. This makes it possible to stratify the presentation of the features of the semantics of programming languages into autonomously developed components in the process of step-by-step development of experimental programming systems and the formation of schemes for studying and teaching system programming. requirements.

Keywords: Definition of Programming Languages, Programming Paradigms, Definition Decomposition Criteria, Semantic Systems

1. Introduction

Too high rate of development of computer, telecommunication and information technologies, in which quick intuitive or volitional decisions that give economic benefits dominate, imperceptibly turned out to be an obstacle at the formation for objective metrics for assessing the quality of programmed solutions and the effectiveness of the tools used. Another obstacle is associated with the duality of assessing the complexity of programs, more precisely, with discrepancies in assessing the complexity of application and the complexity of software development – the external and internal complexity of programs. The border between them is difficult to recognize.

According to J. Weinberg, the author of the unique monograph "The Psychology of Computer Programming" [1], the IBM firm in the early 1970s conducted a study of the spectrum of the complexity of programming depending on experience, age and abilities.

It turned out that on simple problems the spread is 1 in 28, and almost regardless of experience and age. On complex tasks, a positive dependence on abilities, experience and age is noticeable with a slightly smaller spread, about 1 in 10, noticeably in favor of experience and age. Moreover, the speed of completing tasks rarely contributes to the quality of solutions; more often, on the contrary, hasty decisions in programming lead to a lot of labor costs when debugging and operating programs. No less scatter was noticed in assessing the quality of programmable solutions; no positive dependence of program performance on the complexity of programming was revealed.

Fr. Brooks in the early 1970s noted the vulnerability of estimating the complexity of programming in metrics such as "person-month". Not only does the productivity of individual people depend a lot on what, but a lot can happen unpredictably in a month. In this regard, the effect that manifested itself during the mass production of sites in the 1990s is interesting, which then could be observed directly. In this area, it was possible to structure the work in such a way that each performer received an assignment in the morning for one day, only sometimes for a week. With such a scheme, production went on quite steadily. The only pity is that this niche was quickly exhausted.

Various references mention from 20 to 70 programming paradigms (PP) [2, 3], the list of which is modified and expanded depending on the relevance of certain programming problems and fashion for the features of popular IT. The boom of language creation in the field of problem-oriented programming languages (PL), which marks the transition of programming practice from the accumulation of experience at the level of library modules to the accumulation of sublanguages, shows the importance of paradigmatic differences in language definitions when choosing tools for practical work. Another trend is seen in the renewed interest in the creation of monoparadigm PL that are more convenient for research and study. The development of methods for analyzing the definitions of PL in order to establish their paradigmatic characteristics can provide a basis for making recommendations on the choice of tools when creating new software systems and developing IT.

Descriptions of modern PL usually contain a list of 5–10 predecessors and a number of PP supported by the language [1, 2]. In this article the method of representation of paradigms features of PL definition at the level of semantic systems is considered [3]. Using the method of paradigms analysis, it is possible to build a space of constructions supported in the definitions of programming languages and systems (PL&S). This space can be the source structure in the selection of criteria of decomposition programs based on the development of statements of problems in the programming process of their solutions [4], a variety of types of semantic systems of PL and their extensions in the implementation of programming systems (PS) [5]. The technique is shown on the material of four classical programming paradigms without an excursion into the wider space of paradigms, especially new ones, which have not yet received support in well-known programming languages and recognition in the form of examples of debugged programs. The analysis of DSL-languages, which it makes sense to consider as a new meta-level in the field of programming linguistics, is left for the future.

The concept of "programming paradigm" does not have a strict definition, so the question arises about the belonging of new approaches in programming to the set of PP and the ordering of such a set. Programming paradigm is manifested as the way of

thinking associated with the compromise between the characteristics of tasks, methods of their solution in the form of programs, quality criteria of programs adopted in PP and decision-making priorities in the programming process. Such feature of PP allows to understand a paradigm choice as process of acceptance, representation and debugging of decisions at statement of different tasks therefore it is natural to carry out systematization of PP on comparison with priorities and variations of schemes of statement of tasks and methods of their decision.

This article presents an attempt to propose some methods for evaluating programmable solutions and measuring the characteristics of programs and PL based on the conceptual complexity in a form that makes it possible to predict the complexity of programming and thereby navigate the dynamics of the modern space of tools and solutions. Such a methodology for programming systems (SP) can influence the further development of information technologies, if we involve the observations of practitioners who have noticed the significance of the volume of the intuitive work of the programmer and a large number of parameters in assessing the quality of the SP implementation, as well as ideas of conceptual complexity, vertical stratification of programs and semantic systems [4]. The concept of "Programming Paradigm" here concretizes the formulation of P. Wegner "a style of thinking that determines the rules for classifying programming languages in accordance with certain conditions that can be tested". Priorities of decision-making, conditioned by the objectives of the problem being solved and the requirements for its solution, are considered as the conditions to be checked.

After the introduction has noted the vulnerability of the integrality of direct measurements of programming labor in such metrics as "man-month", the second section considers superficially measurable parameters of programming labor and program performance. Most of them almost do not reflect the dependence of the programming result on the decisions made by the programmer and the choice of PL constructs. The third section analyzes the relationship between the complexity of programming and the degree of study of the problems being solved. An underestimation of such a relationship usually leads to systematic errors in forecasting the forthcoming labor costs. In the fourth section, the role of a programmer's qualifications is noted, showing the relevance of developing methods for measuring the characteristics of programs that are important for practice, as well as creating a system of continuous retraining of personnel, whose qualifications must keep up with the pace of progress of IT. The urgency of issues related to the certification of professional qualifications of programmers, especially system programmers is considering. The fifth section is devoted to the conceptual complexity of programming tools. It can be assessed within the framework of a paradigm-semantic decomposition, the results of which are represented by concept tables that can work roughly like a visiting card of a language or programming system. The cellulas of such a table can be associated with elements of an introductory series of debugged examples illustrating the meaning of the concepts of PL. The sixth section shows an example of presenting the results of comparing PLs with an estimate of the distance between languages, and the seventh section describes the structure of a measuring bench, the development of which is aimed at supporting the work on the analysis and comparison of PLs, which is necessary to create a methodology for measuring the contribution

of programmable solutions to program performance. In the conclusion, the hypothesis is expressed that the formula for the complexity of programming can be formalized and made understandable by using some images from mass-used information systems to characterize the elements of the PL concept table.

2. Programming complexity parameters

The lexicon of modern practical programming uses the concept of "programming language" as "an extended subset of the programming language (PL), which is the input language of a typical programming system (SP), operating on the basis of a specific equipment configuration" The difference is that the SP usually accompanies the implementation of the PL with an extensible set of library modules. As a result, the differences visible in practice between languages and programming systems (L&SP) are smoothed out.

In addition, direct measurements of the complexity of programming and the productivity of programs almost do not reflect the dependence of the result on the decisions made by the programmer when choosing the PL designs and the pragmatics of the SP. Although programmable solutions are presented in terms of PL, its influence dissolves in a very complex that inherits the performance of the programming and operation systems and hardware. Thus, there is a problem of creating a method that makes it possible to identify such dependencies by combining direct measurements with the results of expert assessments of the features of PL&S, possibly differing from the assessments of the PL. A number of problems of this kind look like a difference in programming paradigms.

Programming paradigms can be distinguished by the priorities of the categories of semantic systems in the programming process, noting the paradigm differences in the general concepts in each category. The differences between programming paradigms can be expressed as follows. Data are addresses and stored values representation in imperative procedural programming (IPP), stored methods and object signatures appear in the object-oriented programming (OOP), be binding with any data in the functional programming (FP) instead of addresses in memory, and to the identifier in the logic programming (LP). In IPP and OOP, operations are mostly unary or binary, and in FP and LP there is also arbitrary arity. True datum in LP include the special data “ESC”, which allows to distinguish normal predicate values from failure in calculations, and FP can use any data other than “NIL” as truth. Data structures in the IPP cannot be considered as values representation processed by the basic means, and in the FP such structures are processed without special restrictions.

Thus, in addition to preferences on the features of the problem statements, one can see differences in the schemes for determining functions for different categories of semantic systems depending on the software. It should be noted that the transition from PL to PS is usually accompanied by an increase in the number of supported PPs, which, when defining the Haskell language, led to the concept of “monad”, which allows any PL to achieve practicality, which is usually done with the help of library modules.

For practice, it is useful to describe the derivatives of PP relative, expressing the

difference with the base PP. So, IPP derivatives distinguish different methods of representing data in memory and organizing sequential processes generated by the program, OOP derivatives give various concretizations of the concept of “class of objects”, FP derivatives represent variations in the methods of organizing calculations, and LP derivatives may use different approaches to mitigate the dependence of obtaining results on excessive or insufficient determinism.

Any programming paradigm can be supplemented with additional forms, such as declarativeness, abstractions, specification languages, etc., mainly solving problems such as “scaffolding,” that is, the aim of these forms is not an alternative or opposed representation of programming tools and methods, but temporary structure which used to support setting the boundaries of the behavior of programs, highlighting the processes that are convenient for practice.

3. The degree of study of the tasks being solved

The classification of programming paradigms can depend on 1) the space and degree of knowledge of the problems being solved and 2) the potential of technical devices that support programming paradigms, which reflects the operational and implementation pragmatics of the PL that support these paradigms. As new problems appear, new PPs are formed, the recognition of which by specialists requires a significant time, usually 10–20 years after the emergence of tools. The operational pragmatics of a programming language strongly depends on the space of the problems being solved and the practicality of their solutions. The implementation pragmatics of programming systems depends on the range of hardware that can be controlled in PL terms (processor, files and peripherals, networks and servers). In terms of the degree of study, the following spaces of problem statements differ significantly, affecting the choice of methods for solving problems and the complexity of their programming:

- new;
- research;
- practical;
- exact.

New problem statements are characterized by the absence of an accessible precedent for solving the problem, the novelty of the means used, or the inexperience of the performers. **Research** problem statements are usually complicated by the requirements of originality and versatility, which can be demonstrated in a computer experiment. **Practical** problem setting is aimed at relevance and convenience of multiple use of ready-made solutions. **Exact** problem statements include testing the limiting capabilities of the tools used, associated with the degree of organization of the created program and the rank of the implemented solutions. The spread of labor intensity, depending on the degree of novelty or knowledge of the problem statement, is usually about 1 to 8. An underestimation of such a spread usually leads to systematic errors in forecasting the forthcoming labor costs.

Descriptions of modern programming languages (PL) usually contain a list of 5–10 predecessors and a number of programming paradigms (PP) supported by the language

[1, 2]. In this article the method of representation of paradigms features of PL definition at the level of semantic systems is considered [3]. Using the method of paradigms analysis, it is possible to build a space of constructions supported in the definitions of programming languages and systems (PLS). This space can be the source structure in the selection of criteria of decomposition programs based on the development of statements of problems in the programming process of their solutions [4], a variety of types of semantic systems of PL and their extensions in the implementation of programming systems (PS) [5]. The technique is shown on the material of four classical programming paradigms without an excursion into the wider space of paradigms, especially new ones, which have not yet received support in well-known programming languages and recognition in the form of examples of debugged programs. The analysis of DSL-languages, which it makes sense to consider as a new meta-level in the field of programming linguistics, is left for the future.

The concept of "programming paradigm" does not have a strict definition, so the question arises about the belonging of new approaches in programming to the set of PP and the ordering of such a set. Programming paradigm is manifested as the way of thinking associated with the compromise between the characteristics of tasks, methods of their solution in the form of programs, quality criteria of programs adopted in PP and decision-making priorities in the programming process. Such feature of PP allows to understand a paradigm choice as process of acceptance, representation and debugging of decisions at statement of different tasks therefore it is natural to carry out systematization of PP on comparison with priorities and variations of schemes of statement of tasks and methods of their decision.

4. Qualification levels and training for programmers

The mission of system programming is to create tools that improve the quality of information systems, including the search for solutions to ensure the reliability and security of information technology and improve the skills of programmers. Programming results, integrated as an IT industry, are rightly explained by a systematic approach to solving massively demanded problems. A characteristic feature of the systems approach as the leading programming method is the transition to the class of problems in the meaningful analysis of problem statements. Class boundaries are established by choosing a problem-solving process. The results of meaningful analysis are ultimately embodied in a form that is convenient enough for their mass use and borrowing: software documentation, programming languages, computer command systems – architecture, operating systems, programming systems, software tools, program libraries, algorithms, application guides. This allows you to move on to mass application or assembly of solutions to similar problems without deep analysis of each of them.

The use of computers and telecommunications in the implementation of programs, generating mechanisms and solutions to specific problems allows a tempting opportunity, instead of directly solving problems, to rapidly switch to the practice of using ready-made recipes, regardless of the labor costs to achieve a good understanding of the problem. The result of this enthusiasm for this opportunity is well known – the excessive complexity of the software, which leads to the unsatisfactory reliability of its

application and the growth of overhead costs for its study, maintenance and porting to new architectures. This can be seen as a rationale for the need for a more fundamental approach to programming, especially to systems programming.

Extensive development of IT noticeably outstrips human abilities to quickly master new capabilities of IT hardware and system tools that go beyond the user level supported by suppliers of tools and software products. In modern conditions of the increasing dependence of all spheres of life on automation based on IT, attention should be paid to the role of IT specialists' qualifications in the field of system programming and support for post-university professional development of specialists in this vital system of society, comparable to medicine and education.

The general solution to educational programming problems can be summarized as follows. In the rapidly changing world of IT the main goal of training future programmers is to bring them to the level of transition to self-learning and invention or creation solutions for new, previously unknown tasks. This will free the educational process from the race for production technologies that become obsolete before university graduation, and will provide a way to quickly master new products without excessive labor costs for forgetting irrelevant or outdated ones. It is known that primary learning is 3–5 times easier and more reliable than replacing outdated skills.

5. Semantic systems of basic paradigms

Considering the systematization of the paradigmatic features of the definition of PL at the level of semantic systems [3], it is convenient to classify language concepts by statement of tasks and language tools used to solve them. Even in last times, Nicholas Wirth noted the importance of matching the problem statement and the tools used to solve it, especially if you can catch the likeness of the processed data structures and their processing algorithms, which is now called homoiconicity. Based on this correspondence, it is possible to build a space of constructions supported in the definitions of PSL and compared with the complexity of the formulations of successfully solved problems. The resulting space can be the initial structure when choosing criteria for decomposing programs, taking into account the peculiarities of the development of problem statements in the process of programming their solutions [4], expanding the semantic systems of PL and their refinement when implementing PS [5].

When considering any semantic systems, it is important to do noted the difference in the nature of the performance of the functions of such systems in different complexes. So, for any data set D representing values of arbitrary nature, function schemes F are realistically distinguishable for calculation methods, memory access tools M, control features of computing C and communication, or reversible complexation and structuring of data S. This leads to an idea of the main categories of semantic systems for differently implemented types of functions. Historically, at the hardware level, such categories of semantic systems have had a cumulative effect in the “DEMCS” order – the representation of numbers, an arithmometer, a calculator with registers, an analog analyzer with control system, a computer. Each hardware subsystem can interact with each other.

6. Results of paradigms and programming languages analysis

Analysis and comparison of a large number of PL of different levels allow to identify the most significant characteristics for the expression of paradigm specificity of a wide class of PL. The multiparadigmality of long-lived and new PLs shows the need for more precise detailing of the dependencies between old and new ones. The programming paradigm as a way of thinking is associated with a compromise between the features of the tasks being solved and the methods for solving them using programs. The most objective programming concepts are associated with architectural models, with methods for implementing a joint project, and with the classification of problems to be solved. To show the features of software, it is convenient to single out conceptual monoparadigmal languages, models or sublanguages and provide criteria for the successful use of software with evaluating the results using examples of programs that was confirmed by programming practice. [5]. From the vast set, a small number of PLs can be distinguished, attracting attention with interesting combinations of visual means and semantics that affect the development of the main PPs.

Comparing a pair of languages (PL1 and PL2), one can single out their mutual common intersection (PL1 ∩ PL2) and two asymmetric differences (PL1, PL2) or (PL1 ∩ PL2). For the "predecessor-descendant" pair, a clearer understanding of the differences arising during the development of PL appears. Thus, it is possible to see not only the similarities and differences between the two languages, but to show on developing or time-coordinated languages what is lost or added and whether something fundamentally new has appeared.

For example, when comparing and assessing the similarity-differences at the level of the basic semantics of the Lisp (1960) and Pascal (1970) languages, it can be stated that there is a similarity of mutual five semantic systems, the difference is eighteen, some of which are absent in the Pascal language, and some are added in comparison with the semantic systems of the Lisp language. More specifically, the comparison results for semantic systems are as follows.

- **In both** the Lisp and Pascal languages, you can:

- to declare your distinguished names;
- compare values;
- associate names with a representation of their meaning;
- use a hierarchy of expressions, procedures and data structures;
- enjoy access to many data through one notation.

- **The pragmatic difference** between Lisp and Pascal:

- in Lisp, except for NIL, atoms are initially undefined, and in Pascal, self-defined constants mean the index of occurrence into a array;
- in Lisp, the result of a predicate is an ordinary value of NIL or something different from it, and in Pascal, a special type of boolean constants False, True appears;
- Lisp does not require special concepts for memory organization, an associative list is just an ordinary data structure, and Pascal requires not only the concept of "memory allocation for variables", but they are also divided into labels and described variables associated with data types, and the organization of their storage not defined

in language, implementation dependent;

- control of the order of computations in Pascal is performed by different mechanisms, expressed using keywords and labels, and in Lisp expressions are either to be evaluated or blocked;
- Lisp expressions are prefix, and Pascal expressions are infix;
- Lisp builds data structures from binary nodes, while Pascal builds data structures from adjacent blocks of registers.

1 **Losses** in the transition to the Pascal language of what existed in the Lisp language:

- the ability to determine meaning along the fly and in dynamics;
- the right to use any operation or function as a predicate;
- the Von Neumann principle of architecture: equal presentation in memory of programs and data;
- freedom from considering the priorities of operations;
- automate GC memory reuse (garbage collection)

• **Added** in Pascal, was absented in Lisp

- the clarity of the meaning of the names;
- memory allocation according to name descriptions;
- controlled border between calculations and comparisons;
- the ability to use data ordering, go to "neighbors";
- separation of data structures and computation control structures;
- inheritance of habits to the forms accepted in arithmetic expressions;
- the ability to transfer control and assignments to name addresses and vector indices.

To assess the extent to which these additions are new, a finer analysis of functional similarity and implementation constraints is needed, the result of which can be illustrated by examples.

It can be concluded that almost all categories of semantic systems are represented in both PLs with variations in implementation technique and application practice, up to the involvement of standard libraries. The choice of the goals of mutual semantic systems coincides, the mechanisms for implementing a noticeable number of systems differ, when switching from Lisp to Pascal, some directions in the space of solutions for new problems are lost, but many diagnostic situations at the static analysis of program expanded.

7. The structure of the measuring stand

Analyzing the definition of a programming language begins with an understanding of the objectives, requirements, and the programming paradigms supported by the language. As a result, it is possible to single out conceptual tables characterizing monoparadigmatic sublanguages of PL, the work with which is reduced to the following parts:

1) Processing of concept tables.
2) Accumulation of comparison results.

3) Filling for the training series of examples.

4) Testing and performance measurement.

The processing of concept tables and the generation of matrices is needed to support expert work in comparing different PL&S systems and forms for filling the database with fragments of debugged programs, performed as selections, clippings, and rearrangements.

The accumulation of the results of comparison and expert assessment of different PL&S, their storage, laconic presentation and the conclusion of some statistical and analytical characteristics are aimed at analyzing the inheritance and mutual influence of languages during development.

Filling a training series of comparable fragments of debugged programs on different PL&S in a form convenient for experiment and measurements, as well as for the formation of teaching aids and reference books.

Testing and measuring the performance of the studied PL&S systems, organizing storage, statistical analysis and presentation of measurement results are necessary for the objectivity of understanding the dependencies.

Of course, it is not so difficult to construct such estimates for two or three dozen Pls. For the operational work on the analysis and comparison of modern PLs, the number of which already exceeds tens of thousands, it is necessary to automate even simple transformations and transitions to the representation of paradigmally semantic characteristics of PL&S in the form of program fragments with use of experimental stands that already give access to a noticeable number of PLs [7].

Any programming paradigm can be supplemented with additional forms, such as declarativeness, abstractions, specification languages, etc., mainly solving problems such as “scaffolding,” that is, the aim of these forms is not an alternative or opposed representation of programming tools and methods, but temporary structure which used to support setting the boundaries of the behavior of programs, highlighting the processes that are convenient for practice.

8. Conclusions and Outlook

There is a hypothesis that the general formula for assessing the labor intensity of programming can be expressed as the sum of the products of estimates of the degree of study of the problem being solved (from 8 to 1), the qualifications of the programmer (from 1 to 28), and the conceptual complexity of the main works associated with the developed program. In this trio, the conclusion of the assessment of the programmer's qualifications is especially incomprehensible. It depends not only on education and experience, but very much on the abilities and intuitive mechanisms, the manifestation of which may clearly contradict the traditions of the general educational system aimed at controlling static knowledge instead of encouraging the dynamics and prospects of development. There exists a project management technique that recommend achieving reliability of work without "star" programmers. Nevertheless, among practitioners, a fairly reliable intuitive assessment of the professional potential of familiar programmers is usually formed.

Most long-lived programming languages support the practical paradigms of imperative-procedural, object-oriented, and functional programming. A growing number of languages support declarative, reflexive and meta-programming. New languages complement these paradigms with separate networking tools, leading to the formation of paradigms for remote access, parallel processes, supercomputer computing, and large data visualization. The complexity of multi-paradigm PLs can be overcome through the style of separate description of the paradigms supported in them [8].

The proposed methodology can be used to assess the complexity and complexity of programming, especially if supplemented by dividing the requirements for setting tasks in the fields of application into academic and industrial, and by the level of knowledge into clear, developed and complicated difficult to certify requirements.

Basic programming paradigms can be distinguished by ordering the main categories of semantic systems, and derivatives – by the difference between individual categories of semantic systems from the basic paradigm. Any programming paradigm can be enriched with additional paradigms for representing restrictive conditions for the functioning of programs. For this reason, they cannot be opposed to the actual PP. The classification of programming paradigms can depend on the degree of knowledge of the expanding space of tasks to be solved and the progress of available technical means that occurs within the framework of a stable class of tasks to increase efficiency.

The works of E.M. Lavrishcheva [8] and Peter Wegner [6] should be mentioned as related works. E.M. Lavrishcheva presented a fairly complete overview of programming paradigms that is relevant for programming technologies [8] and P. Wegner performed a very serious analysis of OOP, methods for supporting this paradigm, and its comparison with other classical PPs [6].

This work was partially supported by the Russian Foundation for Basic Research, project No. 18-07-01048-a.

References

1. Weinberg, G.M.: The Psychology of Computer Programming. New York: Van Norstand Reinhold Comp., 1971.
2. https://www.levenez.com/lang/. Computer Languages History.
3. http://progopedia.ru/. Encyclopedia of programming languages (171 languages and 31 paradigms).
4. Gorodnyaya, L.V.: O predstavlenii rezul'tatov analiza yazykov i sistem programmirovaniya. Nauchnyy servis v seti Internet: trudy XX Vserossiyskoy nauchnoy konferentsii (17–22 sentyabrya 2018 g., g. Novorossiysk). M.: IPM im. M.V. Keldysha, 2018, https://keldysh.ru/abrau/2018/theses/46.pdf.
5. Lavrov, S.S.: Metody zadaniya semantiki yazykov programmirovaniya. Programmirovaniye, (6), 3–10 (1978).
6. Peter Wegner: Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess. 1, 1 (August 1990), 7–87, https://pdfs.semanticscholar. DOI: http://dx.doi.org/10.1145/.
7. Gorodnyaya, L.V., Demidov, S.V., Kirichenko, M.D., Tkachenko, D.D.: Proyekt informatsionnogo stenda PRIZMA dlya predstavleniya izmerimykh kharakteristik yazykov i sistem programmirovaniya. Informatsionnyye i matematicheskiye tekhnologii v

nauke i upravlenii, 105–119, DOI: 10.38028/ESI.2020.17.1.008.
8. Lavrishcheva, E.M.: Programmnaya inzheneriya i tekhnologii programmirovaniya slozhnykh sistem. Uchebnik dlya vuzov. M., 2018. 432 p.
9. Gorodnaya, L.V.: The programming paradigm. Textbook. Doe St. Petersburg, 2019. ISBN 978-5-8114-3565-4, 232 p. (in Russian).