Bull. Nov. Comp. Center, Comp. Science, 38 (2015), 67–90
© 2015 NCC Publisher

Programming paradigms in higher education

L. V. Gorodnyaya, T. A. Andreyeva

Abstract. The paper concerns a topical problem of System Informatics, namely, the study and development of the methods of analysis, comparison and formal definition of the programming paradigms. The importance of this topic arises from the increase in the number of new-generation programming languages oriented towards the application and development of modern information technologies.

Keywords: programming languages, programming paradigms, programmers' training, educational programming languages, language concepts, implementation structures, parallel programming, very high-level languages.

1. Introduction

This paper introduces the variety of programming paradigms (PPs) and approaches to their support in programming languages and systems (PLSs), focuses on historically important and conceptual programming languages (PLs) that demonstrate the key ideas and practical effects of their realization. The programming styles and languages characteristic to the paradigms under consideration reflect the evolution of the programming technologies (PTs) used for solving problems in System and Applied Informatics: from means of machine programming on the verge of hardware to very-high-level languages and high-performance programming systems (PSs) including means for supporting the software life cycle (SLC).

Programming paradigms were the central point of the *1978 ACM Turing Award Lecture* by Robert Floyd, where he drew attention to the importance of this concept in the context of programmers' training [10]. Being the authority who has laid foundations of the *Program Analysis* and *Program Verifications* theories and created several very effective methods of data processing, Floyd believed that it was important to highlight the PP's influence upon the success of programming projects. He emphasized the role of studying various PPs and ways of their support in PLs. To start with, these are top-down design, step-by-step solution improvement, and problem reduction to simpler subproblems. Next, there occurs a shift from specific machine-level objects and functions to more abstract ones that are helpful for thinking through top-down projected modules.

From his programming experience, Floyd has made an interesting observation: the programming art includes the enrichment of the paradigm repertoire in use. Floyd's reasoning was guided by Thomas Kuhn's works and discussions with authorities in Programming (Edsger Dijkstra, Niklaus

Wirth, David Parnas, Donald Knuth, John Cocke, and Marvin Minsky). Floyd analyzed the effect of various popular programming languages (FORTRAN, Lisp, APL, Algol, Planner, COBOL, and Pascal). His attention was drawn to the possibility of writing the recursive co-programs formulated conveniently in terms of non-determinism whose ineffectiveness can be mastered in practice by macro technique. Robert Floyd highly appreciated numerous examples built at MIT, as they showed the potential of programming in Lisp, and, most of these, the transition from elementary lists to universal data structures (DSs) that can process programs as data.

Robert Floyd aimed the study of Programming at acquiring skills in solving problems that arise at all phases of a programming project. Such teaching can be based on the studying of the semantic support of paradigms at the system implementation level and of paradigm manifestations in the software life cycle. During three decades since Robert Floyd's *The Paradigms of Programming* lecture, the number of various programming languages and systems has increased from hundreds to tens of thousands; while the number of paradigms is not so large: different authorities count from twenty to forty paradigms. There are reasons to believe that the need for the enrichment of the paradigm repertoire in programming pointed out by Floyd is related to the dynamics of knowledge representation specific to succession of bottom-up and top-down stages in problem specification.

Many famous scientists have made great contribution to the paradigmization of programming languages. Among them are such coryphaei in Informatics as John McCarthy, John Backus, Peter Naur, Jacob T. Schwartz, Edsger Dijkstra, Niklaus Wirth, Robert Floyd and others [3, 7, 10, 29, 34]. Among Russian scientists, it is necessary to mention Andrey Ershov and Sviatoslav Lavrov [8, 56]. A preview of the problem of computer languages classification was made in [2, 42, 43].

It can be pointed out that, in the period from *Computing Curricula 2001* [55] to *Computer Science Curricula 2013* [54], recommendations show tendencies to shift the teaching of programming paradigms down towards the first year of education and to pay more attention to parallel programming. At Novosibirsk State University, these tendencies are reflected by the *Programming paradigms* course [60], which covers the concept of paradigms, the methods of definition of programming languages and systems, the abstraction levels of programming languages including low-level languages, and the main programming paradigms of high-level and very-high-level languages for parallel programming. Also, the paper describes an approach to the teaching of programming paradigms and the educational programming languages and systems required for this teaching.

2. Conception

Different approaches to information processing that were formed and accumulated while creating and applying the programming languages and systems are called programming paradigms. Today, experts recognize more than a score of PPs. Many PLs can be *referred* to five to eight paradigms. The study and clear classification of the already existing and new PPs must help to reasonably choose and design computer languages while forming programming projects and developing IT [5, 16, 17, 20, 36].

The evolution of PPs reflects the usability of language concepts and implementation structures availed in creating complex programming systems. It must be said that the vogue rate of PLs differs from the rate of PLs availed in successful projects. Some difficulties in the research into PPs arise from the ambiguous classification of programming means. Many languages *refer* to several paradigms; sometimes, a language is unreasonably affiliated with a vogue paradigm. That is why we not only compare paradigms by DSs and admissible means of their processing but also take into consideration the criteria and boundaries of their successful application. Moreover, we consider requirements imposed on the exploration level of the problems solved and the abstraction level of the concepts used [49, 51, 60].

For example, low-level programming is characterized by the feasibility to implement effective solutions at the cost of using common access to poorly protected DSs. Meanwhile, the peculiarity of high-level languages is a DS hierarchy whose components are protected against uncontrolled interactions of independently created program fragments. The very-high-level means (specification languages, parallel languages, knowledge representation systems and so on) focus on the completeness of the space of implementation solutions. Complexity of the space is mastered by factorization over separate directions of problems in the field of design and application of long-living programs [26, 37].

Besides educational, amateur, and experimental programs, the world of programming has produced a wide spectrum of various instruments for various conditions of program design and application. Independently of this variety, there exists a purely programming line of increasing the programming productivity. The main milestones of this line are bound to the crystallization of certain PPs; they look like creation of certain PLs, invention of new ways of PS implementation, nascence of new PTs:

1. Creation of FORTRAN was accompanied by the incipience of the separate compilation technology.
2. Lisp gave life to the symbolic computation technology and functional programming.
3. Development of C as an operation system implementer resulted in the

machine-dependent porting technology.

4. Usage of the educational language Pascal helped to formulate the maxims of structural programming, which plays the role of the methodologically dominant paradigm in educational programming.
5. PROLOG is bound to the logical programming paradigm and to implementation means for logical derivations based on the partial solution definition sufficient for practice.
6. Process organization in Simula67 and object implementation in SmallTalk80 impelled object-oriented programming and the technology of program decomposition over the class hierarchy, which allows extensions that do not distort previously debugged definitions.

Most of these approaches, except for Lisp, regard a program as a static object. Nonetheless, in reality, the program develops and can be partly modified.

We have to point out that many authors of modern PTs do not approve of the authorities' recommendations about the programming style and technique because of the misconception that these make programming practice helpless. Here one must keep in mind that ITs develop quickly and so their potentialities are recognized later. Additionally, since parallel programming is laborious, methods for verifying compilation and optimization of program components should be developed. Implementation of these methods requires means for macro-based code generation and automated program transformations that support property validation for the combinations of the components validated earlier [47, 52, 53].

In general, the process of program development can be represented as a sequence of steps in three directions:

- refinement of the problem's statement and solution methods,
- improvement of the solution's program text,
- augmentation of the data set used as debugging tests.

If such a sequence achieves a state where the refined entities correspond, then debugging is claimed to be over. Making the debug process convergent is the most important problem of a programming technology. When the inter-accordance is achieved, the problem's final refined statement can be both a generalization and a restriction of the initial problem.

It should be noted that the extending class of programming problems more and more actively attempts to cover application fields for which no programmable algorithms are available yet. Solution methods for such problems are on the level of preliminary study. It is known that the progress of the study level of problem statements is not as monotonous as the growth of the implementation laboriousness of program versions. Namely, it is easy to design a prototype in order to show the advantages of an idea. It is much more difficult, however, to create an experimental testing field for the full exploration and valuation of the problem's solution means and methods and for the determination of its implementation limits. As a rule, the laboriousness of the practical version is intuitively underestimated, which leads to a shortage of explorations and often results in the necessity to repeatedly verify the problem solution.

3. Programming languages and systems

A programming paradigm is a tool forming the professional conduct that guarantees the reliability of the ITs in mass use. The Vienna method for the definition of programming languages has thoroughly studied the problem of PLs and PSs description. This method was developed in the late 1960s. Its main idea is the definition of programming language semantics by an abstract syntax (AS) and an abstract machine (AM) [28]. Languages that share AS are semantically equivalent: they are comparable in terms of the laboriousness of program debugging. There are two styles of the PL semantics definition: the semantics of value computations and the semantics of memory alterations. Specifics of the compilation process are complicated even for simple languages; therefore, the specification of the compilation process is often described in terms of language-oriented abstract machines (AMs) [21]. Programming languages that share an AM are semantically equipotent: they provide a basis for the comparable effectiveness of computation processes.

While studying the requirements for program compilation and analysing the compiler definition schemes, one can notice that, for many PLs, such a definition can be presented by means of the same PL. There are two different approaches to the organization of the compilation process: to select as a unit to be compiled either the whole program or only some of its functions or procedures. The compilation of the whole program creates an independently executable code whose functioning depends on the input data only. The separate compilation of functions and procedures assumes that a subroutine's executable code is a reusable component to be embedded in various programs. Including subroutines in a PS implements the problem-oriented extension of a PL.

The study of the various schemes of partial, mixed, and lazy evaluations and meta-compilation shows that it is expedient to combine such schemes within one PS in order to exploit their advantages on the different exploration levels of the problems solved. Partial evaluation allows execution of a program with a lack of input data. Operations which have data are executed and a residual program is created: it can be executed later with the rest of the data to produce the same result as the initial program could have produced from a full data set. Mixed evaluation allows an arbitrary marking-out of the program into executable and suspended parts. The routes not blocked by suspended actions are executed and a residual program is created: it can be executed after the suspended actions have been de-blocked. Lazy evaluation is only performed if the operation result is necessary for other actions; the results are saved in memory to avoid further recalculations. Meta-compilation processes a program together with its typical data set [1, 19, 23, 31, 35].

Traditionally, a programming system may contain an interpreter-compiler pair. Any interpreter contains elements whose implementation can be described in machine terms: memory structure, binary tree implementation, and so on. Any compiled program contains interpretable components, for example, calls to a file system and other OS elements. In practice, the advantages of interpretation become evident when a program is being debugged, while these of the compilation are obvious when a ready program is being exploited. However, this topic is worth a more detailed discussion.

To classify PPs, it is important to define the smallest educational implementation kernel. As a result, the set of all PLs can be split into the classes of implementationally similar and substantially comparable languages having the same (or, to be more exact, equivalent) semantic basis. Sometimes, the PL's kernel contains some concepts that are not represented directly in the PL implemented. Such concepts expand a language and have a considerable influence on understanding the mechanisms of effective programming. The implementation of experimental educational programming languages and systems often uses unwinding, which minimises initial laboriousness by excluding the formal redundancy of the PL's means. A kernel can be separated out; all the rest is programmed methodically on its basis.

The analysis of PPs is based on implementation pragmatics (IP). IP affects all levels of the PL definition, but mainly represents solutions in a certain memory management. This specifies solutions and maxims declared in the AM's definition. First of all, the memory protection solutions differ from the memory finiteness-considering solutions. The implementation pragmatics that supports different PLs includes:

- In functional programming (FP): lists, garbage collector, atom property lists;
- In imperative-procedural programming (IPP): side effect, vectors, data types (variable – value);
- In logical programming (LP): differential lists, serial search, backtracking;
- In object-oriented programming (OOP): references, virtual and abstract methods and classes, multiple inheritance.

PLs that share an IP are implementationally equivalent and comparable in terms of the laboriousness of PS implementation. Implementation pragmatics is a specification of operational semantics, starting from four main semantic systems (monads): data procession, data storage, data structure, and data processing control in programming systems. Moreover, to predict the efficiency of a PP selected, exploitational pragmatics (specific application conditions) is also taken into account as an expert valuation of the requirements imposed on the results of using the PP.

4. Abstraction levels

On studying the method for defining the LP's paradigmal characteristic in the form of a specification of the interactions of the main semantic systems[1] (such as data processing, data storing, data structuring, and data procession controlling), the following three paradigm levels are distinguished, showing the extension of the language support of the program life cycle and an increase in the implementation complication of a PL definition:

- low-level coding,
- high-level programming,
- programming on the base of very high-level languages.

The main characteristic of low-level coding is the hardware approach to computer management, aimed at the access to any hardware resources. The main attention is paid to hardware configuration, state of the memory, control-transferring commands, the event queue, exceptions and failures, reaction time, and reaction success [5, 24, 32, 33]. To store data and programs, the global memory and automaton model of data procession control are used.

High-level programming allows the definition of data structures that reflect the nature of a problem to be solved and actively uses the domain hierarchy of data structures and their processing procedures. This hierarchy follows the structure-logical managing model that allows the convergence of the program debugging process. For a short time, even in micro-programming, Pascal and C got the upper hand over the assembly language as a preferable tool.

Programming in very high-level languages is aimed at the representation of regular, effectively implementable data structures whose processing allows transformations of data and program representation, use of similarities and constructions that guarantee high computing productivity and reliability of the development of programs fitted for variations of architecture solutions.

[1]The *semantic system* concept was introduced by Sergey Lavrov [56].

For each paradigm, there exist programming languages corresponding to it (the so called *referring*, or *relating* languages); in many PLs, several paradigms are represented or implemented. On studying and specifying the paradigmal characteristics of such languages, it is natural to represent the language definition as an aggregation of monoparadigmal sublanguages that are fragments of the initial language.

As a rule, the actively exploited programming languages and systems (PLSs) absorb tools from different paradigms, which impedes their study. To be precise, almost any PL allows a package library that supports the required paradigm. It is convenient to use definitions of several monoparadigmal languages in order to ascertain to what extent the language studied supports the paradigm. Some conceptual languages were selected while the main programming paradigms were described. The comparative description of the exploitational and the implementational pragmatics of the main paradigms results in the following method of the definition of a PL's paradigm:

- The semantic basis definition: decomposition of the language into fragments in order to define the basic means of the language and its implementation kernel.
- Decomposition of the language's semantic basis into main semantic systems with the least complexity and probably their description in terms of the conceptual languages.
- The normalized definition: determination of the language's AM formally sufficient for building extensions equivalent to the initial language.
- Comparison of the obtained definition with the descriptions of the known paradigms and conceptual languages.
- Determination of the language level and its place in the life cycle of programs and programmers' activities (goals and tasks) and also of the base languages exploited for its design and implementation as recommendations for the selection and application of the PL and its PS.

In Russia, the studies of the paradigmal characteristics of programming languages and systems lay in Andrey Ershov's sphere of influence and originate from discussions held in the 1970s at conferences and seminars of various experimental projects aimed at the development of the means, methods, and techniques for an effective and reliable implementation of a programmer's toolkit. At that time, the number of languages to study was a little more than two or three hundred and so it was much easier to decide on their key ideas. Now the issue of a classification of programming languages, computer languages, and information systems has become much more urgent [42, 64].

5. Low-level languages

Programming (or coding) in the low-level languages (LLLs) is associated with one-level data structures (DSs) determined by architecture and hardware[2]. Data and programs are stored in the common global memory with an arbitrary access. In principle, the utmost program effectiveness is achievable, but program debugging is complicated by the *low start – high finish* combination. In other words, it is easy to succeed in first exercises, but it is difficult to create a program product and to maintain it competently. Distinguishing for LLLs is univocal accordance between a program and the process generated by its running. Therefore, the LLL's operational semantics can be analysed on the level of an abstract machine, which fully determines the characteristics of programs and processes designed with the help of a LLL. As a rule, three registers are enough to define the abstract machine of an LLL: the "result", the "program", and the "memory" (or the "result", the "context", and the "program").

Traditionally, LLLs include machine-dependant assembly languages, macro processors, machine-oriented languages, and script languages [5, 24, 32, 33]. For these LLLs, it is characteristic to signify all actions explicitly. A program is an arbitrary mixture of commands, which can occupy almost any position. All fragments of data and programs are accessible. For data and data structures, all basic means of their representation in memory and a control scheme of their processing are predefined; this helps to *refer* a LLL clearly to a certain paradigm. Other approaches to the machine-oriented effective programming are also interesting.

Data processing with the help of programs written in assembly languages comes to an imperative machine-oriented model for controlling the execution process of program-generated actions. An assembly language operates such data as addresses and values. The paradigm of low-level coding in assembly languages is aimed at considering any specific features of computer architecture. Architecture is often defined as a set of user-accessible resources. These are the command system, the common registers, the processor status word, and the address space. The assembling process consists of

a) reserving memory for the sequence of commands making up the program being assembled,

b) matching identifiers used in the program with their memory addresses,

c) mapping assembly commands and identifiers onto their machine equivalents.

Programming in an assembly language requires knowledge of commands in use, their operands and results. Low-level programming is successful

[2]The assembly language Elbrus and autocode Engineer are counter-examples showing that the pure hardware-based estimation of the language level is insufficient.

if computer architecture has been studied in detail, which finally leads to its understanding. In general, maxims and ways of programming do not depend on the language. The main requirement is the ability to think logically. The imperative programming style of LLLs is inherited by most high level programming languages (HLLs) supporting procedural-imperative and object-oriented programming.

Forth gives an example of computations over a stack. It can be regarded as the kernel language with the possibility of almost unlimited problem-oriented expansion. A well written program in Forth is a specialized virtual machine that can be expanded further as the problem statement develops. The Forth interpreter sorts words by belonging to the dictionary [5]. The memory processing is based on a stack. The program execution is a dialog over the stack. Each command "knows" what to get from the stack, what to transform this into in order to produce the result of the program, and what results to push onto the stack. The Forth programming system contains the interpreter-compiler pair, and the compiling technique is highly effective. The system uses the unified program representations of data and commands: these are word sequences. Data are positioned before operations that process them. An operation is a word known to the system. Data are just pushed onto the stack, and an operation gets them from there, according to the number of its operands.

The macro technique gives powerful but not always safe means to increase a PL's expressiveness. For macro processors, the LLL semantics usually accompanies the string processing tools, in the open procedure style. A program is a flow of macro definitions and macro calls. Macro names can be regarded as base means. A macro processor is often exploited together with an assembler (a macro assembler) and other PLs. Macro expansions can use local and global variables, nested scopes, and recursion. A macro processor can be built into a compiler; it can be an autonomous tool of a programming system, such as a text editor, optimiser or debugger; or it can exist independently as a universal general-purpose tool. As an instrument for extending a programming system, a macro processor must be developable by its nature. The macros' main assignment in programming systems is to provide flexibility and portability of programs applied in different conditions. Many difficulties in such a use of macro technique arise from type-checking on the source code level. Conceptually, the macro technique is close to the production programming style, mark-up languages, and text rewriting systems, which are actively growing today as hypertext languages for developing sites and information services. As a matter of fact, Lisp's special functions defined with the FEXPR and the FSUBR indicators act as macro definitions, i.e., they perform open argument substitution [29].

The script languages for an OS look like a cross between macro assemblers and HLLs. The difference appears in the data processed and processing commands.

- Files play the role of data: files are independently behaving objects that are subject to the external influence.
- Files do not have to exist during their processing. Files can be used in several processes simultaneously.
- The command execution is regarded as an event. This event can be both successful and unsuccessful. Also, there exist external events.
- Reaction to an event is programmed as the event's handler to be executed independently from other handlers: it is a separate process.
- The program of a process can be aimed not at achieving a result in a finite time but at supporting the ceaseless service of object-processing tasks.
- A process can be active or postponed. Processes can rival for common objects. The synchronization of processes and generation of slave processes are possible.
- The program of a process looks like an object; it is created as a data element, and can be applied as a command.
- The consequent commands in the program do not have to be executed in exactly the same order.
- A command can be executed during several time intervals between which other commands are executed.

As a result, the design of programs for organising the interaction of processes differs from that of the ordinary sequential programs at a very deep level and requires the implementation of data structures for the queues that regulate access to objects. The two most frequently used models are a supervisor that manages interactions of several processes and an automaton that can multiply itself when processes ramify. In both cases, program execution comes to an infinite cycle analysing the current events whose appearance turns on the handlers corresponding to these events. The termination problem is solved outside the language at the level of basic tools or externally by interrupts. During its design and debugging, any program is executed against the background of an operation system that manages the data input-output processes in order to demonstrate the course of data processing. Therefore, the minimum context of a program to be debugged is the standard input-output accessible by default [33].

Very interesting is the design of the adjustable macro assembler Sigma aimed at porting programs across different architectures, which can be used to support architecture-independent highly productive computations.

It is not difficult to determine the LLL paradigms: the key idea is obvious, and the semantic systems are more or less isolated in the language definition. The main difference is in the instantiation of the *value* concept and of the spectrum of tools for enlarging the program units.

The data representation and data processing methods amassed in LLLs were largely inherited by the implementation methods in HLLs, which helps to localise the study of these methods. The practice of programming in LLLs is important for education. Universities which prepare highly skilled programmers and drill winners of international programming contests start education from teaching programming in assembly languages and in script languages.

6. Main programming paradigms

While analysing the paradigms of the high-level programming languages (HLLs), the following characteristics have to be considered:

- To make programs laconic, implicit forms of concept representation are practised.
- Most often, expressions use a pre-calculation scheme over definite-size scalars or complex values (first, operands are evaluated; next, the result of operations is calculated).
- Types of forks and cycles, categories of functions and procedures are multifarious.
- Data types are constructed according to firm rules set in a PL and are implemented in accordance with the patterns usual to its PS.
- Schemes of calculation management are often fixed in a PL and are firmly implemented in its PS.
- Interaction and compliance of means and methods relevant to different semantic systems but implemented in one PS are defined according to traditions and precedents; in the code, their implementations are hidden or scattered and not structured.
- Effectiveness of programming is based on the knowledge of the implementation methods of the values and handlers of data structures in memory.
- As a rule, the result of a program is scattered over several values; still, many HLLs have expressions and functions that form only one result.

Standard imperative-procedural programming (IPP) considers information processing as a finite sequence of local changes of memory. It is typical for IPP that the *program* and *values* concepts are divided in regard to the static methods of data type checking and program optimisation during compilation. It is natural to realize the common interpretation mechanism for a standard program as an automaton with separate name tables for variables that can alter and for labels and procedures that are constant. The result of a program is formed as a sequence of the alterations of data placed at certain addresses. Therefore, a programmer has to study thoroughly all side effects both of the program's fragment in work and of the adjacent ones. This has proved to be a serious obstacle against a speed-up.

Functional programming (FP) regards data processing as a composition of their mappings with the help of universal functions. From this point of view, a program is no more than a data variant. From a practical standpoint, in standard PLs, any constructions can be introduced as functions that augment the initial programming system, which turns them into fully legal means of the functional approach. Such representation does not mean that concepts are lumped together: vice versa, it keeps all concept limits and builds an enveloping space where these concepts are regularized and can interact in accordance with the formal definitions of different function categories. In many papers, the correlation between the potentials of the imperative approach and of the functional one was studied. The formal co-reducibility (with some minor limitations on the programming technique) was proved [51]. The functional programming languages (FPLs) are aimed at the full type checking during the execution of programs that allow dynamic analysis, postponed calculations, and an automatic memory management (*garbage collection*).

We ought to point out the difference between the interpretations of FP maxims formed in the programming theory and practice. This difference appeared in the 1980s, when standards for Lisp were made; two standards were approved: "LISP1" for the academic Lisp and "LISP2" for the industrial one. Theoretically, it is sufficient to study purely functional, laconic program representations. The behaviour of these programs does not depend on side effects and the program result can be obtained by reducing its representation. The reducing processes can be selected according to the calculation strategy. In practice, the most laborious is program debugging in a specific programming system supporting a definite calculation strategy and allowing an explicit management of the program execution. The optimising compilation can reduce a purely functional program to its formal result, not generating its executable code.

The imperative organisation of calculations in accordance with the principle of immediate and obligatory execution of each command is not always effective. There exist many non-imperative models of process management. The processes can be interrupted or postponed and then restored, renewed, or cancelled. Organisation of such management sufficient to process optimisation and programming is implemented with the help of the so-called *lazy*, or *called-by-need* evaluation. The main idea of such evaluation is to reduce function calls to their calculation receipts that include function closures in a certain context.

Logical programming (LP) reduces data processing to selecting an arbitrary result-producing composition of definitions (facts, equations, predicates, etc.). Formula processing is the basis: calculations are understood as operations over a formula. In case of a failure, other definitions are processed. Representation of variants is similar to the definition of forks without a predicate that controls the choice. Its implementation resembles that of variant records or unions in common HLPLs. As a rule, the concepts of an algorithm and a program are connected to the determined processes. However, they do not become very complicated if non-determinism limited by a finite number of variants (so that at each moment only one of them exists) is allowed. Such an approach is efficient when solving problems associated with changing the attitude to problem stating and to valuing its solution methods in the course of coding, debugging and exploiting the program.

Object-oriented programming (OOP) regards information processing as partial processing of objects by means of reacting to events with the help of the methods selected according to the type of data to be processed. The assortment of private methods used in a program is structured according to the class hierarchy of the objects to be processed by these methods. It means that constructions defined in a program can be locally modified while common schemes of the information procession are fixed. Due to this, programs can be modified by declaring new classes and by adding new methods for processing the objects of separate classes without radical alteration in the previously debugged program. Thus, the OOP reflects the evolution of approaches to data structure organisation on the level of problems and their solving programs, starting from the imperative-procedural programming paradigm. The OOP introduced concepts of information hiding, inheritance of definitions down the class hierarchy, and implementation polymorphism of functions and operations.

In shifting from the standard IPP to the OOP, ways of program organisation show radical changes induced by a growth in hardware capacity. The OOP alters conventional programming in many ways. Instead of creating a separate program that operates with a bulk of data, one has to cope with data that have their own behavior while the program is reduced to the interactions of *objects*, which are a new data category. In order to compare the OOP and the FP, Paul Graham, in the description of the Common Lisp standard [13], proposed to consider the model of the object-oriented language (OOL) embedded into Lisp and gave an 8-line example of OOL implementation on the basis of hash-tables. As a matter of fact, the inheritance is provided by a single Lisp feature: the recursive version of GETHASH. However, in Paul Graham's opinion, Lisp was always an OOL.

The typical formulations of problem statements draw borders between the domains of different paradigms.

- Imperative-procedural programming: *There exists an algorithm for the solution of a relevant problem. It is necessary to prepare a program for this algorithm's implementation with effective spatial-temporal characteristics on an available hardware.*
- Functional programming: *The knowledge domain is known. It is necessary to select a symbol data representation for this domain and adjust a system of universal functions suitable for various data-processing programs that will solve relevant problems from this domain.*
- Logical programming: *A collection of facts and relations showing a relevant problem is given. It is necessary to reduce this collection to a form sufficient for getting answers to relevant queries about this problem.*
- Object-oriented programming: *A hierarchy of classes of objects, which supports efficient methods of solving problems in some knowledge domain, is available. It is necessary to easily adjust this hierarchy to solving new relevant problems from this domain or from its extension or variant.*

Practical problems often include all these wordings as subproblems, which results in supporting different paradigms simultaneously while PLs are designed and PSs are built. We ought to draw attention to the fact that the most successful and long-living programming languages and systems are multi-paradigmal.

The main difficulty in shifting to new programming paradigms is the lure of an easy-way, the tendency to simulate quickly the accustomed programming means and methods. A more effective way is to study them as alien worlds. It is easier to accept unusual ideas as a self-standing theory or an intellectual game, which not only leads to known and interesting problems but also provides an advantage of elegant solutions and deep understanding.

In a PL, one can recognize the main paradigm and the additional ones. In referent PLs, parts that fully correspond to one paradigm can be identified. Descriptions of the main HLPL paradigms are easily simulated by FP means. The main differences are in the variations of the discipline of access to separate name categories. Parts referent to an additional PP can play the role of this paradigm's model when it is compared with other languages.

7. Parallel programming

Today, the expansion and evolution of the system of base concepts necessary for the rational development of the process management systems on modern hardware result in an urgent need to form the parallel programming paradigm (PPP) [6, 18, 22, 25, 30, 48]. The diversity of parallel computing models and the widening spectrum of available architectures should be regarded as a challenge for the developers of PLSs helpful in creating compilation methods for multithreaded programs for multiprocessor configurations. The language has to allow representations of all models of parallelism that appears at the problem statement level and can be implemented on the existing architecture.

Shifting to parallel algorithms induces a revision of many concepts. New concepts reflecting the phenomena and effects insignificant for conventional sequential algorithms are introduced. For instance, there are special problems that arise from the necessity to regard the special features of the multilevel memory in multiprocessor systems. These problems are of no importance in sequential programming: they are solved by a compiler that has static information about the types of data in use and can optimize the program if necessary. Also, the use of a PPL as the source code language does not guarantee that the program will be suitable for easy paralleling.

Only recently, studies performed at the Institute for System Programming, Russian Academy of Sciences, began to demonstrate a serious approach to the problems of parallel programs debugging; theories of processing the information prone to distortion and comparative debugging methods using the concepts of a sample program and a distributed debugger scheme are being developed. A common parallel programming paradigm that would bind together the means and methods for creating and developing parallel programs has not been formed yet.

Programming in very high-level languages (VHLLs) is aimed at the long-living solutions of urgent and complex problems. The life cycle becomes longer due to the representation of generalized solutions having a degree of freedom in the full spaces of acceptable adjacent components that have been or will be implemented. The implementation shrinkage of the family of processes allowed by the VHLL semantics contradicts its aims and concepts for the following pragmatic reasons:

- The solutions to be programmed are at a high abstraction level.
- The problems to be solved depend on unpredictable external factors.
- The basic tools and/or algorithms use parallelism.
- The pragmatic requirements to the computation rate and capacity are relevant.
- The dynamically reconfigurable multiprocessor complexes are in use.

As a rule, parallel programming languages (PPLs) include tools characteristic for different paradigms. Nonetheless, parallelism is important as such and is not just an addition to traditional programming. However, reports given at conferences on compilation methods continue to concern separate, historically formed, implementation means for languages and systems adapted only for the initial transfer to the world of parallel programming. They include the *just-in-time* compilation, separation of purely functional subsets and forms with the only assignment, reversible compilation and interpretation, transaction memory, action accessibility analysis, loop transformations over large arrays, and so on. As a rule, the operational semantics of a programming language is based on the definition of an abstract machine which, in the case of multiprocessor configurations, naturally becomes an abstract complex (AC), i.e. a construct of abstract processors.

Separation of the scheme level makes it easier to include verification mechanisms (pattern matching or conformity to axioms) into the scheme of program development. On their basis, it is possible to check the program plausibility, logical deduction of properties, and execution of inductive and deductive constructions. The road to the automation of parallel programming runs through the development of a constructing and debugging system for language-dependant libraries of program transformations whose compilation is supported by the use of verified program components and by code generation adjustable to certain hardware. A valuable solution of parallel programming problems requires the creation of more specialised tools. Some of their implementation mechanisms can be studied in the form of experimental development of an educational PPL.

8. Teaching of programming paradigms

Being an enthusiastic supporter of Robert Floyd's ideas, Academician Andrey Ershov attached high importance to studying and teaching programming paradigms [8]. Due to efforts of Ershov and his associates and followers, the paradigmal approach to the teaching of programming was accepted at Novosibirsk State University (NSU). Many courses (e.g., *Logical programming*, *Functional programming*, and *Parallel programming*) focus at studying the corresponding programming paradigm and languages in detail. Along with these courses, only the *Programming paradigms* course familiarizes students with the methods of paradigm description, analysis and comparison. The course has been taught to the third-year students of the NSU Information Technologies Department (ITD) since its foundation in 2000 and at the NSU Mechanics and Mathematics Department (MMD) since 2002 [48].

By the fourth year, ITD students already have an idea of the most common PPs (the procedural PP and the object-oriented PP) and some experience in applying them. Yet not many of the students realize, for example, that they work within the domain-specific metaprogramming paradigm when they use Microsoft Excel to prepare spreadsheets or Open Office to develop HTML-pages. The same is true of the MMD students using Mathematica or Maple systems.

The lectures delivered during the course introduce a wide variety of existing PPs, then systemize and generalize this information by analyzing and comparing some paradigms. Their specific features are illustrated by the fragments of the *referring* languages though the course is not aimed at learning these PLs in detail.

The practical part of the course focuses on studying the parallel PP and the functional PP.

The main way to study PPs in practice suggests studying several PLs *referring* to them. These languages may be educational [50] and, at best, monoparadigmal. Students solve the same problem in different PLs and within different PPs. In this case, it is important to require that data representation and data processing correspond not only to the nature of the problem but to the *referring* paradigm as well. Thus, programs solving the same problem and written in, for example, C++, Lisp and Prolog must differ both in syntax and in structure. As a result, students not only get ideas of different PPs but also become acquainted with several PLs.

Another way to study and compare PPs in practice uses only one multiparadigmal PL. The richest set of suitable features has Lisp, which supports almost all PPs: from functional to object-oriented and from procedural to generic. This also allows students to learn this language in detail.

Using just one multiparadigmal PL prevents sliding down to a familiar PP, which often occurs while working with several PLs: programs are written with the procedural means embedded almost in every PL. Therefore, when solving the same problem in several PLs, students have to show additionally that the opted ways of data representation and data processing correspond to the PP being studied.

Students' individual work is also important. The *Programming paradigms* course includes the following learning techniques: independent learning a new PL's syntax, solving problems in this PL, and finally, doing written reports and giving talks about the specific features of this PL.

It was noted that the first autonomous attempts to study a new PL referring to an unfamiliar PP inevitably lead to using new syntax with old programming methods. To restructure thinking and to familiarize themselves with a new programming style, students need time. In this case, it is useful to solve serial tasks [41, 9, 20, 39] where more complex solutions are derived (sometimes after just a slight modification) from the simpler solutions written earlier.

Inertness of thinking mostly shows itself when students start getting familiarized with the first paradigm that differs markedly from a familiar one. Yet, henceforth they learn new paradigms more and more easily: inertness and rejection of an unfamiliar programming style give way to the fervor of solving new "puzzles". According to Robert Floyd [10], such fervor is a must-have for any highly skilled programmer.

The Internet University of Information Technologies has a vast collection of educational materials for programming (more than a hundred courses). Among them are courses that cover the main programming paradigms [58, 59, 60, 62, 63]. In addition to these courses, there is an interesting philosophical essay concerning methods and styles of programming [61]. Also, there are new works that represent a review of fundamental paradigms, with parallel programming given the highest priority[63], and argumentation in favor of the importance of teaching of programming paradigms, which results from the difference between the number of programming paradigms (a score or two) and the number of programming languages (tens of thousands). Useful materials (for example, short courses on some programming paradigms and languages) are also available at the *Coursera* [65] and other sites.

9. Educational programming languages and systems

Time has come to teach Computer Science and Programming from the view of parallelism. Shifting to parallel computing does not make the initially difficult teaching of Programming much more complicated. Acquaintance with the educational problems of parallel programming and with the parallelism models that occur in educational and experimental programming languages and systems shows that the teaching of programming skills has to cover the distance from the level of base means for controlling interacting processes to the level of programs for high-productive computing [18, 22, 30, 33, 34, 64].

Creation of an educational high-level language aimed at parallelism, at studying language means supporting the function arguments field similar to the iterations field, and at an action and scheme typification supporting computations, memory reorganization, memory modification, duration registration, memorization, flows construction, dynamic performance registration and so on, is a matter for the future [11, 12]. Also, it is important to prevent students from addiction to traditional sequential programming. Within the educational parallel language, it is advisable to divide the computing sequence and the sequence of placing results into memory elements. These sequences may not coincide [14, 27, 38, 39].

To choose a programming paradigm means to choose a conceptual scheme for problem stating and solving, an instrument for "literate" description of facts, events, phenomena, processes, and particular and common concepts. Different approaches to information processing that have been accumulated and exist now as programming languages and systems are called programming paradigms. Analysis and efficient classification of the existing and new computer paradigms make choice of programming languages for building up new programming projects and inventing new information technologies well founded.

The aim of the design of an educational programming language intended for teaching complex computational models and parallel programming is to select mechanisms supporting experiments in designing new programs, languages and paradigms oriented towards educational research projects in the field of distributed information systems. Educational experiments should cultivate a habit to use program verification methods since without these program reliability and program capability are questionable. Such a training is a topical issue because the solutions of well-studied problems continuously turn into standard component libraries or immediate-use application toolkits. Real programming always contains new subproblems, which result in a next turn of programming and debugging.

References

[1] Aho A. V., Hopcroft J. E., Ullman J. D. Data Structures and Algorithms. – Addison-Wesley, 1983.

[2] Anureev I.S., Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. On the problem of computer language classification // Bulletin NCC. Series: Computer Science. – 2008. – Vol. 28. – P. 31–42.

[3] Backus J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs // Communs. ACM 21. – 1978. – Vol. 8. – P. 613–641.

[4] Bratko I. PROLOG Programming for Artificial Intelligence. – 2000. – ISBN 0-201-40375-7.

[5] Brodie L. Thinking Forth (PDF Online book) / Bernd Paysan, ed. – 2004. – ISBN 09764587-0-5.

[6] Cann D. C. SISAL 1.2: A Brief Introduction and Tutorial. – California, May, 1992. – (Tech. Rep. / Lawrence Livermore National Lab., Livermore; UCRL-MA-110620).

[7] Dijkstra E. W. Structured Programming (EWD-268). – E.W. Dijkstra Archive. Center for American History, University of Texas at Austin.

[8] Ershov A. P. Aesthetics and the human factor in programming // Communs. of the ACM. – 1972. – Vol. 15, No. 7.

[9] Field A. J., Harrison P. G. Functional Programming. – London, 1990.

[10] Floyd R. W. The paradigms of programming // Communs. of the ACM. – 1979. – Vol. 22, No. 8. – P. 455–460.

[11] Gorodnyaya L. On the language for basic learning of parallel programming // Proc. Ershov Informatics Conf. PSI Ser., 8th Edition. Internat. Workshop on Program Understanding, Novososedovo, Russia. – 2011. – P. 18–24.

[12] Gorodnyaya L., Shilov N. Educational value of teaching parallel programming paradigm // Proc. Ershov Informatics Conf. Workshop on Educational Informatics, Novosibirsk, Russia. – 2011. – P. 1–6 (In Russian).

[13] Graham P. ANSI Common Lisp. – Prentice Hall, 1996.

[14] Harvey B. 1997 Computer Science Logo Style. – MIT Press (3 volumes). – ISBN 0-262-58148-5, ISBN 0-262-58149-3, ISBN 0-262-58150-7.

[15] Henderson P., Hoare C.A.R. Functional Programming: Application and Implementation. – Prentice-Hall, 1980.

[16] Henner C.R. A Simple Set Theory for Computer Science – Toronto, 1979. – (Prep. / TR N 102).

[17] Higman B. A Comparative Study of Programming Languages. Reader in Computer Science. – London and American Elsevier, 1969.

[18] Hoare C.A.R. Communicating Sequential Processes. – Prentice Hall Internat. Ser. in Comput. Sci., 1985. – ISBN 0-13-153271-5 hardback or ISBN 0-13-153289-8 paperback.

[19] Hopgood F.R.A. Compiling Technology. – London: Macdonald, 1970.

[20] Hudak P. Conseption, evolution and application of functional languages // ACM Computing Servys. – 1989. – Vol. 21, No. 3. – P. 359–411.

[21] Iliffe J.K. Basic Machine Principles – London: Macdonald, 1968.

[22] Iverson K, E. A Programming Language. – New York: John Wiley & Sons, Inc., 1962. – ISBN 0-471-43014-5.

[23] Knoop J. Compiler construction // Proc. 20th Internat. Conf., CC 2011. – Lect. Notes Comput. Sci. – 2001. – Vol. 6601.

[24] Knuth D. The Art of Computer Programming. Vol. 1–4. – Addison-Wesley, 2005–2011.

[25] Kotov V.E., Marchuk A.G., Vishnevsky Yu.L. MARS – a hierarchical heterogeneous modular system // Proc. IFIP TC 10 Working Conf. on Fifth Generation Computer Architectures. – Amsterdam: North-Holland Publishing Co., 1986. – P. 277–289.

[26] Lehman M. M. Programs, life cycles, and laws of software evolution // Proc. IEEE. – 1980. – Fasc. 68, No. 9. – P. 1060–1076.

[27] Levinsky J. L. The GROW Book. – San Diego, California, Computer Systems Design Group, 1980.

[28] Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languages. – Vienna, 1968. – (Tech. Rep. / IBM Laboratory; TR 25.087).

[29] McCarthy J. LISP 1.5 Programming Manual. – Cambridge: The MIT Press, 1963.

[30] Odersky M. The Scala Language Specification Version 2.7.

[31] Pratt T. W., Zelkowitz M. V. Programming Languages Design and Implementation. – Prentice Hall PTR, 1999.

[32] Prechelt L. An Empirical Comparison of C, C++, Java, Perl, Python, Rexx and Tcl for Search/String-Processing Program. – Universitat Karlsruhe, 2000. – (Tech. Rep. / Fakultat fur Informattik; 2000-5).

[33] Ritchie D.M., Tompson K. The UNIX time-sharing system // Bell System Technical J. – 1978. – Vol. 57, No. 6. – P. 1905–1929.

[34] Schwartz J. T. Set Theory as a Language for Program Specification and Programming. – Courant Institute of Mathematical Sciences, New York University, 1970.

[35] Strachey Ch. A general purpose macrogenerator // Computer J. – 1965. – Vol. 8 (3). – P. 225–241.

[36] Tanenbaum A. S. Structured Computer Organization. – Englewood Cliffs, New Jersey: Prentice-Hall, 1979. – ISBN 0-13-148521-0.

[37] Van Tassel D. Program Style, Design, Efficiency, Debugging and Testing. – Prentice Inc. California, 1978.

[38] Weinberg G.M. The Psychology of Computer Programming. – New York: Van Norstand Reinhold Comp., 1971.

[39] Wetherell Ch. Etudes for Programmers. – Prentice Hall Inc. Davis, 1978.

[40] Yourdon E. Death March. The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects. – Prentice Hall, 1997. – ISBN 0-13-748310-4.

[41] Andreyeva T.A. Serial tasks in programming // Proc. Fifth Internat. Andrei Ershov Memorial Conf. Workshop on Educational Informatics, July 9–12, 2003, Academgorodok, Novosibirsk, Russia. – IIS SB RAS, 2003. – P. 2–4 (In Russian).

[42] Andreyeva T.A., Anureev I.S., Bodin E.V. et al. Computer languages as device for presentation of scientific and professional knowledges // "Telematika-2008". – Sankt-Peterburg, 2008. – P. 77–78 (In Russian).

[43] Andreyeva T.A., Anureev I.S., Bodin E.V., et al. Educational value of the computer lnguages classification // Applied Informatics. – 2009. – No. 24. – P. 18–28 (In Russian).

[44] Andrei Ershov – a Scientist and a Person / Ed. by A.G. Marchuk. – Novosibirsk: SB RAS Publishing, 2006 (In Russian).

[45] Gorodnyaya L.V. Introduction to parallelism for children // “Telematika-2008”. – Sankt-Peterburg, 2011. – Vol. 2, Sec.D. – P. 323–324 (In Russian).

[46] Gorodnyaya L.V. To automation of parallel programming // “Scientific Service at the Internet”. – http://agora.guru.ru/abrau2012/pdf/239.pdf (In Russian).

[47] Gorodnyaya L.V. Parallel programming paradigm: models, languages, systems // VII Siberian Conf. on Parallel and High-performance Calculations. – Tomsk: TGU, 2013. – P. 17–18 (In Russian).

[48] Gorodnyaya L.V. Parallel programming paradigms at universities // “Scientific Service at the Internet”. – 2008. – P. 180–184 (In Russian).

[49] Gorodnyaya L.V. Programming in education of computer scientists // Problems of Specialization at Education. – Novosibirsk, 1998. – P. 115–124 (In Russian).

[50] Gorodnyaya L.V. Functional programming: style, method and potential // ”Cosmos, Astronomy and Programming” (Lavrov’s Days). – Sankt-Peterburg: Sankt-Peterburg State University, 2008. – P. 46–53 (In Russian).

[51] Gorodnyaya L.V. Functional Aproach to Definition of the Programming Paradigms. – Novosibirsk, 2009. – (Prep. / IIS SB RAS; No. 152) (In Russian).

[52] Gorodnyaya L.V., Marchuk A.G. Development of the models of parallelism at the high level lenguages // “Scientific Service at the Internet”. – 2013. – http://agora.guru.ru/abrau/2013/ (In Russian).

[53] Shilov N.V., Gorodnyaya L.V., Bodin E.V. Paradigms of parallel programming: to teach or not to teach (that is the question) // “Scientific Service at the Internet”. – 2011 (In Russian).

[54] https://www.acm.org/education/CS2013-finalreport.pdf

[55] Computing Curricula 2001. Computer Science. – Final Report (December 15) – IEEE Computer Society. – https://www.acm.org/education/curric_vols/cc2001.pdf.

[56] Svyatoslav S. Lavrov’s Archive. – http://ershov-arc.iis.nsk.su/archive/eaindex.asp?lang=2&tid=79

[57] Peter Van Roy. Classification of the Principal Programming Paradigms. – http://www.info.ucl.ac.be/˜pvr/paradigms.html.

[58] Andreyeva T.A. Programming in Pascal. – http://www.intuit.ru/studies/courses/41/41/info (In Russian).

[59] Gorodnyaya L.V. Fundamentals of Functional Programming. – http://www.intuit.ru/studies/courses/1109/204/info (In Russian).

[60] Gorodnyaya L.V. Paradigms of Programming. – http://www.intuit.ru/studies/courses/1109/204/info (In Russian).

[61] Nepejvoda N.N. Styles and Methods of Programming. – http://www.intuit.ru/studies/courses/40/40/info (In Russian).

[62] Soshnikov D.V. Logical Programming. – http://www.intuit.ru/studies/courses/558/414/info (In Russian).

[63] Mejer B. Essence of Object-oriented Programming. – http://www.intuit.ru/studies/courses/71/71/info (In Russian).

[64] Site with Materials on the Mozart System That Supports the Educational multi-paradigmal programming language Oz. – http://sourceforge.net/projects/mozart-oz/.

[65] Free On-Line Courses from the World'S Leading Universities. – http://www.coursera.org/ and http://ru.coursera.org/.