Anyway, TD6 and its modifications are not the program which we are looking for, although TD6 can be done to satisfy the definition (because the definition does not say anything about the efficiency of AI). The program which we are looking for is much closer to that one which is described in [6, 7]. The problem in TD6 is that it looks for a model of the world which consists from only one item. It is better if the model is a set of many items (the items can be Turing machines, final automata or logical formulas). When we make a theory in logic then it consist from a set of axioms and we can change smoothly the theory by modifying, adding or deleting one axiom. Any theory in logic is a model of some world. AI has to use similar models which can be modified smoothly.

Bibliography

[1] Dobrev D. D. A Definition of Artificial Intelligence. In: Mathematica Balkanica, New Series, Vol. 19, 2005, Fasc. 1-2, pp.67-74.

[2] Dobrev D. D. AI - What is this. In: PC Magazine - Bulgaria, November'2000, pp.12-13 (in Bulgarian, also in [4] in English).

[3] Dobrev D. D. AI - How does it cope in an arbitrary world. In: PC Magazine - Bulgaria, February'2001, pp.12-13 (in Bulgarian, also in [4] in English).

[4] Dobrev D. D. AI Project, http://www.dobrev.com/AI

[5] Dobrev D. D. First and oldest application, http://www.dobrev.com/AI/first.html (1993)

[6] Dobrev D. D. Testing AI in one Artificial World. Proceedings of XI International Conference "Knowledge-Dialogue-Solution", June 2005, Varna, Bulgaria, Vol.2, pp.461-464.

[7] Dobrev D. D. AI in Arbitrary World. Proceedings of the 5th Panhellenic Logic Symposium, July 2005, University of Athens, Athens, Greece, pp.62-67.

[8] Kolmogorov A. N. and Uspensky V. A. Algorithms and randomness. - SIAM J. Theory of Probability and Its Applications, vol. 32 (1987), pp.389-412.

[9] Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Series 2, 42, 1936-37, pp.230-265.

Author's Information

Dimiter Dobrev - Institute of Mathematics and Informatics, BAS, Acad.G.Bonthev St., bl.8, Sofia-1113, Bulgaria; P.O.Box: 1274, Sofia-1000, Bulgaria; e-mail: d@dobrev.com

PROGRAMMING PARADIGMS IN COMPUTER SCIENCE EDUCATION

Elena Bolshakova

Abstract: Main styles, or paradigms of programming – imperative, functional, logic, and object-oriented – are shortly described and compared, and corresponding programming techniques are outlined. Programming languages are classified in accordance with the main style and techniques supported. It is argued that profound education in computer science should include learning base programming techniques of all main programming paradigms.

Keywords: programming styles, paradigms of programming, programming techniques, integration of programming techniques, learning programming paradigms

ACM Classification Keywords: D.1 Programming Techniques – Functional Programming, Logic Programming, Object-oriented Programming

Introduction

Several main ***styles*** (or ***paradigms***, or models) ***of programming*** – imperative, functional, logic and object-oriented ones – were developed during more than forty-year history of programming. Each of them is based on specific algorithmic abstractions of data, operations, and control and presents a specific mode of thinking about program and its execution. Various ***programming techniques*** (including data structures and control mechanisms) were elaborated rather independently within each style, thereby forming different scopes of their applicability. For instance, the object-oriented style and corresponding techniques are suitable for creating programs with complicated data and interface, while the logic style is convenient to program logic inference.

Though modern programming languages [Finkel, 1996] usually include programming techniques from different styles, they may be classified according to the main style and techniques supported (e.g., programming language Lisp is a functional language while it includes some imperative programming constructs).

Nowadays, for implementation of large programming project, techniques from different paradigms are required, mainly because of complexity and heterogeneity of problems under solution. Some of them are problems of complex symbolic data processing, for which programming techniques of functional and logic languages (e.g., Lisp [Steele, 1990] or Prolog [Clocksin, 1984]) are adequate. The other problems can be easily resolved by means of popular imperative object-oriented languages, such as C++ [Stroustrup, 1997].

Below we explain our point that acquirement of programming techniques of all main paradigms belongs to background knowledge in the field of computer science. Accordingly, learning of modern programming languages should be complemented and deepened by learning of programming paradigms and their base techniques.

Programming Paradigms

The ***imperative*** (procedural) programming paradigm is the oldest and the most traditional one. It has grown from machine and assembler languages, whose main features reflect the John von Neuman's principles of computer architecture. An imperative program consists of explicit commands (instructions) and calls of procedures (subroutines) to be consequently executed; they carry out operations on data and modify the values of program variables (by means of assignment statements), as well as external environment. Within this paradigm variables are considered as containers for data similar to memory cells of computer memory.

The ***functional*** paradigm is in fact an old style too, since it has arisen from evaluation of algebraic formulae, and its elements were used in first imperative algorithmic languages such as Fortran. Pure functional program is a collection of mutually related (and possibly recursive) functions. Each function is an expression for computing a value and is defined as a composition of standard (built-in) functions. Execution of functional program is simply application of all functions to their arguments and thereby computation of their values.

Within the ***logic*** paradigm, program is thought of as a set of logic formulae: axioms (facts and rules) describing properties of certain objects, and a theorem to be proved. Program execution is a process of logic proving (inference) of the theorem through constructing the objects with the described properties.

The essential difference between these three paradigms concerns not only the concept of program and its execution, but also the concept of program variable. In contrast with imperative programs, there are neither explicit assignment statements nor side effects in pure functional and logic programs. Variables in such a program are similar to those in mathematics: they denote actual values of function arguments or denote objects constructed during the inference. This peculiarity explains why functional and logic paradigms are considered as non-traditional.

Within the ***object-oriented*** paradigm, a program describes the structure and behavior of so called objects and classes of objects. An object encapsulates passive data and active operations on these data: it has a storage fixing its state (structure) and a set of methods (operations on the storage) describing behavior of the object. Classes represent sets of objects with the same structure and the same behavior. Generally, descriptions of classes compose an inheritance hierarchy including polymorphism of operations. Execution of an object-oriented program is regarded as exchange of messages between objects, modifying their states.

Table 1. Features of the main programming paradigms

Paradigm	*Key concept*	*Program*	*Program execution*	*Result*
Imperative	Command (instruction)	Sequence of commands	Execution of commands	Final state of computer memory
Functional	Function	Collection of functions	Evaluation of functions	Value of the main function
Logic	Predicate	Logic formulas: axioms and a theorem	Logic proving of the theorem	Failure or Success of proving
Object-oriented	Object	Collection of classes of objects	Exchange of messages between the objects	Final state of the objects' states

The object-oriented paradigm is the most abstract, as it's basic ideas can be easily combined with the principles and programming techniques of the other styles. Really, an object method may be interpreted as a procedure or a function, whereas sending of message as a call of procedure or function. Contrarily, traditional imperative paradigm and non-traditional functional and logic ones are poorly integrated because of their essential difference.

Distinguishing features of the main programming paradigms are clarified in Table 1.

Programming Languages and Programming Techniques

Each algorithmic language was initially evolved within a particular paradigm, but later it usually accumulates elements of programming techniques from the other styles and languages (genesis of languages and relations between them are shown in Fig.1).

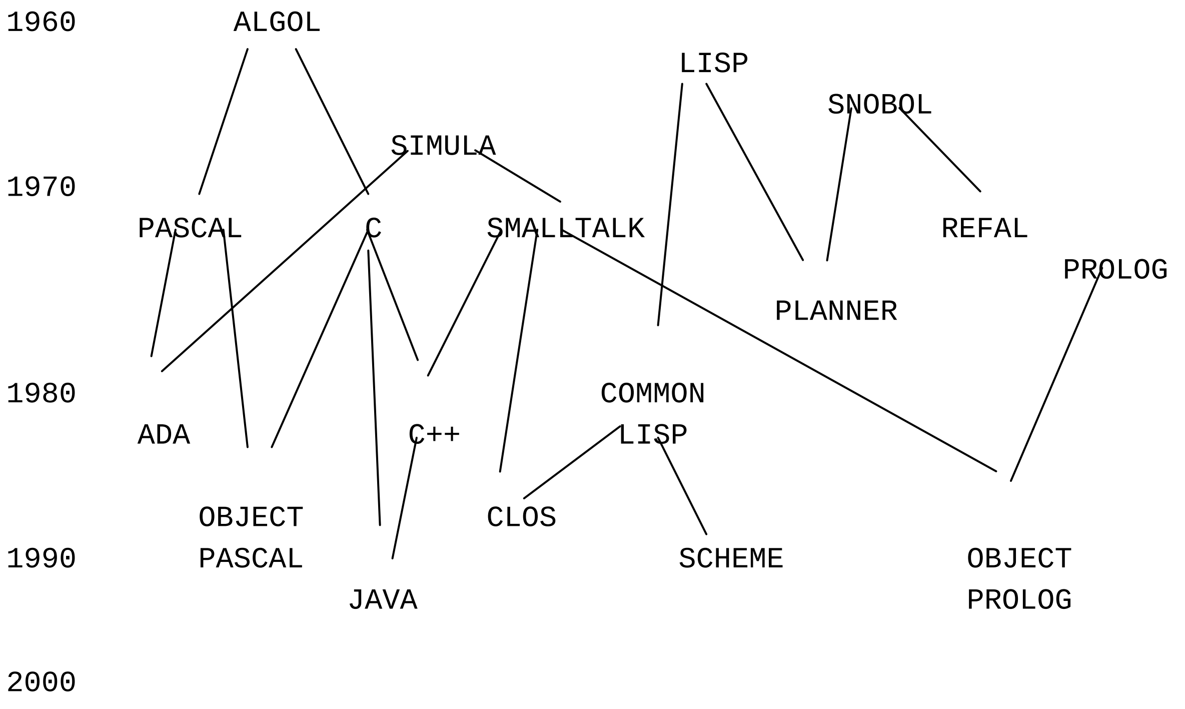

Fig. 1. Genealogy of programming languages

Hence, as a rule, most languages include a kernel comprising programming techniques of one paradigm and also some techniques from the other paradigms. We can classify languages according to paradigms of their kernels. The following is a classification of several famous languages against the main paradigms:

– Imperative paradigm: Algol, Pascal, C, Ada;
– Functional paradigm: Lisp, Refal, Planner, Scheme;
– Logic paradigm: Prolog;
– Object-oriented paradigm: Smalltalk, Eiffel.We could notice that Smalltalk [Goldberg, 1982], the first object-oriented language, is not popular because of complexity of its syntax and dynamic semantics. But its basic object ideas (abstraction of object's state and behavior, encapsulation and inheritance of state and behavior, polymorphism of operations) are easily integrated with the principles of programming languages of the other styles. For this reason, the object-oriented paradigm became widespread as soon as it was combined with traditional imperative paradigm. To be more precise, it became widespread when it was embedded into the popular imperative languages C and Pascal, thereby giving imperative object-oriented languages C++ and Object Pascal.

Analogous integration of object-oriented principles with programming techniques of the other paradigms has led to object-oriented variants of non-traditional languages. For example, the language Clos is an object-oriented Lisp developed on the base of Common Lisp [Steele, 1990], the popular version of Lisp. Modern programming languages, which are combinations of two paradigms, are:

– Imperative + Object-oriented paradigms: C++, Object Pascal, Ada-95, Java;
– Functional + Objects-oriented paradigms: Clos;
– Logic + Object-oriented paradigms: Object Prolog.

Programming techniques elaborated within the traditional imperative paradigm and imperative languages, are well known [Finkel, 1996]: control structures include cyclic and conditional statements, procedures and functions, whereas data structures comprise scalar and compound types – arrays, strings, files, records, etc. Programming techniques of imperative object-oriented languages also includes object types and corresponding techniques, such as virtual procedures and functions.

Programming languages based on non-traditional paradigms provide rather different data types and control structures [Field, 1988], they also differ from traditional languages in the mode of execution: interpretation instead of compilation applied for imperative and imperative object-oriented languages.

Unlike imperative languages, logic and functional languages are usually recursive and interpretive, and most of them are oriented towards symbolic processing. Besides recursion, programming techniques developed within these languages include:

– flexible data structures for representing complex symbolic data, such as list (Lisp) or term (Prolog);
– pattern matching facilities (Refal, Prolog) and automatic backtracking (Planner, Prolog);
– functionals, i.e. high order functions (Lisp, Scheme);
– mechanism of partial evaluations (Refal).

Programming techniques elaborated within corresponding programming style and programming languages have its own scope of adequate applications. Functional programming is preferable for symbolic processing, while logic programming is useful for deductive databases and expert systems, but both of them are not suitable for interactive tasks or event-driving applications. Imperative languages are equally convenient for numeric and symbolic computations, giving up to most of the functional languages and Prolog in the power of symbolic processing techniques. The object paradigm is useful for creating large programs (especially interactive) with complicated behavior and with various types of data.

Integration of Programming Techniques

Nowadays, imperative object-oriented languages C++, Java, and Object Pascal supported by a large number of developing tools are the most popular choice for implementation of large programming projects. However, these languages are insufficiently suitable for implementation of large software projects, in which one or several problems often belong to the symbolic processing domain where non-traditional languages, such as Lisp or Prolog, are more adequate. For instance, development of a database with a complex structure (approx. a hundred of various relations) and with a natural language interface (queries written as sentences from a restricted subset of natural language and responses in a convenient NL form) involves the following problems to be resolved:

Problems	*Suitable programming languages*
Syntactic analysis of NL query	Refal
Semantic analysis of the query	Lisp, Prolog
Processing of the query	Prolog
Elaboration of response	Lisp, Refal
Modern user interface	C++, Object Pascal, Java
DB managing operations	C++

On the right hand of the problems, corresponding adequate programming languages are indicated. Evidently, languages oriented to symbolic processing are preferable for syntactic and semantic analysis of natural language queries, as well as for generation natural language phrases expressing responses. We suppose that semantic analysis of the queries may imply some logic inference, which is available in Prolog.

Thus, in order to facilitate implementation of programming projects an integration of programming techniques from different languages and styles is required. In particular, it seems attractive to enhance the power of popular imperative object-oriented languages with special data structures and control mechanisms from non-traditional languages.

As far as the necessity of integration of various programming techniques arisen long before the appearance of popular object-oriented languages, two simplest ways were proposed for solving the problem. The first way suggests creating in the source language some procedural analogue of the necessary technique from another language. This way is labor-intensive and does not preserve the primary syntax and semantics of built-in techniques.

Another way of integration involves coding each problem in an appropriate programming language and integrating of resulting program modules with the aid of multitask operating system (for example, via initiating its own process for each module). This way is difficult to realize because of closed nature of implementation of non-traditional languages and incompatibility of data structures from different languages (e.g., data structures of Prolog and C++ are incompatible).

However, the first way of integration was recently developed for integrating various programming techniques on the basis of an imperative object-oriented language, such as C++ or Object Pascal. The key idea of the method proposed in [Bolshakova and Stolyarov, 2000] is to design, within the given object-oriented language, special classes of objects and operations modeling necessary data and control structures from another language. The method was successfully applied for building functional techniques of the programming language Lisp [Steele, 1990] into the C++ language [Stroustrup, 1997], resulting in a special library of C++ classes that permits to write Lisp-like code within a C++ program.

We should also note that the second way of integration, i.e. direct integration of programming codes written in different languages, now becomes perspective in connection with the development of Microsoft.NET platform, which permits compiling and linking such different codes.

Learning Programming Paradigms

Necessity of integrating various programming techniques and languages within the same software project is the claim of modern programming. Therefore, a profound education in the field of computer science should be based on learning programming techniques of different paradigms. This implies learning several different algorithmic languages, as none of languages can comprise all possible techniques from various programming styles. Our point is that courses on modern programming languages included in typical curricula should be complemented by special lectures devoted to programming paradigms and intended to compare their base programming techniques and to explicate distinguishing features of the paradigms. Another option to deepen the knowledge of programming techniques and programming languages is to enrich a general course on programming languages with education material on the main programming paradigms. Since the 80s a similar course is read at Algorithmic Languages Department of Faculty of Computational Mathematics and Cybernetic in Moscow State Lomonossov' University.

The importance of learning programming paradigms is also explained by the fact that we cannot know the future of popular modern languages. During the history of programming, many languages became dead, while some other languages have lost their popularity, so some modern languages may have the same destiny. However, the main programming paradigms will be the same, as well as their base programming techniques, and thus their learning is a permanent constituent of education in the field of computer science.

Conclusion

We have outlined main programming paradigms, as well as programming techniques and programming languages elaborated within them. Programming techniques of traditional imperative paradigm essentially differ from techniques of nontraditional ones – functional and logic. They have different scopes of applicability, and for this reason necessity to integrate techniques of different paradigms often arises in programming projects. Accordingly, a profound education in computer science implies acquirement of programming techniques of all main paradigms, and usual learning of modern programming languages should be complemented by learning of programming paradigms and their base programming techniques.

Bibliography

[Bolshakova, Stolyarov, 2000] Bolshakova, E., Stolyarov A. Building Functional Techniques into an Object-oriented System. In: Knowledge-Based Software Engineering. T. Hruska and M. Hashimoto (Eds.) Frontiers in Artificial Intelligence and Applications, Vol. 62, IOS Press, 2000, p. 101-106.

[Clocksin, 1984] Clocksin, W.F., Mellish C.S. Programming in Prolog, 2nd edition. Springer-Verlag, 1984.

[Goldberg, 1982] Goldberg, A., Robson D. Smalltalk-80 – the language and it's implementation. Addison-Wesley, 1982.

[Field, 1988] Field, A., Harrison P. Functional Programming. Addison-Wesley, 1988.

[Finkel, 1996] Finkel, R.A. Advanced Programming Language Design. Addison-Wesley Publ. Comp., 1996.

[Steele, 1990] Steele, G. L. Common Lisp – the Language, 2nd edit. Digital Press, 1990.

[Stroustrup, 1997] Stroustrup, B. The C++ Programming Language, 3rd edition. Addison-Wesley. 1997.

Authors' Information

Elena I. Bolshakova – Moscow State Lomonossov University, Faculty of Computational Mathematics and Cybernetic, Algorithmic Language Department; Leninskie Gory, Moscow State University, VMK, Moscow 119899, Russia; e-mail: bolsh@cs.msu.su