www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 4 Issue 11 Nov 2015, Page No. 14953-14959

Programming Paradigms in the Context of the Programming Language

[1]Sonia Kumari, [2] Pratibha Yadav, [3] Kumari Seema Rani

[1, 2, 3]Shyama Prasad Mukherji College (For Women), University Of Delhi, India

[1] soniakumari.ducs@gmail.com, [2]pratibhamcadu2011@gmail.com, [3]raniseemabca@gmail.com

ABSTRACT

The choice of the first programming language and the corresponding programming paradigm is critical development of a programmer. In computer science, several programming paradigms can be recognized. There is the huge number of programming languages introduced over the last fifty years, the key issues in programming education remain the same and choosing appropriate programming language is still challenging. In this paper we overview some of the most important issues relevant for programming, the challenges in programming both in terms of programming paradigms and in terms of the programming languages. In this paper, we have also overviewed the concept of abstract machine and operations performed by the interpreter. Some results about the usage of programming language are also presented. paper, we overview the concept of abstract machine, interpreter and paradigms related to programming languages. Also, we discuss the challenges, in programming both in terms of programming paradigms and in terms of the programming languages.

Keywords
Programming Paradigms, Programming language, Abstract Machine, Interpreter.

1. Introduction

In the modern society, relying on information technologies, programming education is extremely important. It is clear that the choice of the first programming language and the corresponding programming paradigm is critical for later development of an IT professional. Over the last fifty years, there was thousands of programming languages introduced, belonging to several programming paradigms. However, despite the big number of programming languages, there are just a few truly important programming concepts and there are not many languages that survived for more than ten years. It is very important to detect what are suitable features of a programming language. It is important to consider these issues both in terms of individual programming languages and in terms of programming paradigms. Over the last decades, several programming paradigms emerged and profiled. The most important ones are: imperative, object-oriented, functional, and logic paradigm. In this

2. Abstract Machine

An Electronic, digital computer is a physical machine that executes algorithms which are suitably formalized so that the machine can understand them. Intuitively an abstract machine is nothing more than an abstraction machine is nothing more than abstraction of concept of a physical computer [21].

For actual execution, the algorithm must be properly formalized using the constructs provided by a programming language. In other words, the algorithms
We want to execute must be represented using the instructions of a programming language. This language will be formally defined in term of a specific syntax and semantics.

- The Syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. This applies both to programming languages, where the document represents source code, and mark up languages, where the document represents data. The syntax of a language defines its surface form. Documents that are syntactically invalid are said to have a syntax error.

- The Semantics means the meaning. Computer languages, semantic processing generally comes after syntactic processing, but in some cases semantic processing is necessary for complete syntactic analysis, and these are done together or concurrently.

3. INTERPRETER

The interpreter performs the operations that are specific to the language it is interpreting. However, even with the different types of languages, it is possible to define the type of operations and an "execution method "common to all the interpreters.

The type of operation executed by the interpreter and associated data structures, fall into the following categories [22]:

- **Operations for processing primitive data:** A machine, even an abstract one, runs by executing algorithms, so it must have operations for manipulating primitive data items. These items can be directly represented by a machine. For example for physical abstract machines, used by many programming languages, numbers (integer or real) are almost always primitive data. The machine directly implements the various operations required to perform arithmetic (addition, multiplication, etc.) There arithmetic operations are therefore primitive operations as far as abstract machine is concerned.

- **Operations and data structures for controlling for controlling the sequence of execution of operations:** Operations and structures for "sequence control" allow controlling the execution flow of instruction in a program. The normal sequential execution of a program might have to be modified when some conditions are satisfied. The interpreter therefore makes use of data structures (for example to hold the next instruction to execute) which are manipulated by specific operations that are different from those used for data manipulation.

- **Operations for controlling data transfers:** These operations are included in order to control how the operands and the data are to be transferred from the memory to the interpreter and vice versa. These operations deal with the different store addressing modes and the order in which operands are to be retrieved from store. In some cases, auxiliary data structures might be necessary to handle data transfers. For example, some types of machine use stacks (implemented either in hardware or software) for this purpose.

- **Operations and data structures for memory management :**
This concerns the operations used to allocate data and program in memory. In the case of abstract machines that are similar to hardware machines, storage management is relatively simple. A program and its associated data could be allocated in a zone of memory at the start of execution and remains there until the end, without much need for memory management Abstract machine for the common programming languages uses more sophisticated memory management techniques. In fact, some construct in these languages either directly or indirectly cause memory to be allocated or reallocated.

4. Challenges of Programming

Acquiring and developing knowledge about programming is a highly complex process. New programmers have to overcome a wide range of difficulties [11][12]. In this section we discuss

some of the goals and problems of programming education.

Goals, Objectives, and Outcomes

There are five overlapping domains that students should acquire in an introductory course [1]:

- General Orientation — the capabilities and applications of programs.
- The notional machine — an abstract model of the computer used for executing programs.
- Notation — the syntax and semantics of a particular programming language.
- Structures — the structuring of basic operations into schemas and plans.

- Pragmatics — the skills of planning, developing, testing, debugging, documenting etc.

Goals and objectives of a programming course can be summarized as follows [8]:

Main goals:
- Become familiar with the fundamental concepts of computer science.
- Develop proficiency in an engineering problem solving and design methodology.
- Understand the importance of advanced information technologies.

Main objectives:
- Use computers and application software as tools to solve problems.
- Analyze, design, build and test operational solutions.
- Acquire the foundation of algorithmic processes.
- Learn to exploit the educational and professional resources available on the Internet and World Wide Web.
- Develop a framework for considering the ethical implications of advanced.

5. Main Obstacles in Learning a Specific Programming Language

In acquiring syntax and semantics of a particular programming language, programmers are faced with problems arising from the fact that the designers of the language are domain experts that do not pay attention about the about the influence of the language design in the learning process [5][6]. According to, seven principles, often applied in the design of programming languages, could be the source of problems to novice programmers:

Less is more — this principle can appear in many different forms. The most
Obvious example is in Scheme language where exists only one data type (the list) and one operation (evaluation of the list). While this abstraction is very simple to explain and not difficult for the beginner to understand, it results in code difficult to read because of large numbers of nested parentheses and the absence of other structuring punctuation.

More is more — some programming languages provide too many features. Since most of the textbooks and compilers attempt to cover the full language, new programmers are forced to get informed about all of these features. For example, C++ provides over 50 distinct operators at 17 levels of precedence, Ada9X has 68 reserved words and over 50 predefined attributes, Modula 3 reserves over 100 keywords and some Lisp dialects define over 500 special functions.

Grammatical traps — Syntactic synonyms (two or more syntaxes that are available to specify a single construct), syntactic homonyms (syntactically the same constructs having two or more different semantics depending on context),
And elisions (the omission of a syntactic component) are very confusing from the new point of view.

Hardware dependence - The programmer is often forced to contend not only with syntactical and semantic issues, but also with the constraints of the underlying hardware. For example, in the programming language C the standard int type varies from 16 to 32 bit representations depending on the machine and the compiler implementation.

Backwards compatibility - This is a useful property from the experienced programmer's point of view, as it promotes reuse of both code and programming skills. The novice, however, can take no advantage of these benefits, and instead,

has to accept some counterintuitive rules (introduced for the sake of backwards compatibility).

Excessive cleverness -Some programming languages aim at providing features for easier programming based on clever support. However, by enabling large freedom and wide ranges for interpreting syntax rules, some of such features rather add to confusion of novice programmers. Sometimes, such "cleverness" is the cause of complete misunderstanding of supposedly simple concept.

Violation of expectations — Violations of syntactical and semantically expectations are the most undesirable features for the introductory programming language. For example, in programming language C/C++, the following code "if (x=1 || y<10) f ... g" is syntactically correct, although the condition involves the assignment operator (rather than the comparison operator). The condition is always evaluated to true (regardless of the values of |x| and |y|) while the value of |x| is silently reset to one.

Choosing Appropriate Programming Language

A programming language should have [5]:

- Simple usage of input/output operations,
- Readable and consistent language syntax,
- Small and orthogonal set of features,
- Clearly syntactically differentiated all programming constructs (even if they are similar in concept, functionality, or implementation).

6. Programming Paradigms

In computer science, several programming paradigms can be recognized. Moreover, the four main problem-solving approaches, i.e., programming paradigms, are recognized as fundamental [6][15]. Each of these approaches involves a distinct way of thinking and each is supported by a range of programming languages. These paradigms are:

- Imperative paradigm
- Object-oriented paradigm
- Functional paradigm
- Logic paradigm.

➢ Imperative Programming

The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940's. Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory. This is reflected in the design of t he imperative languages, with [16]

- States — representing memory cells with changing values,
- Sequential orders — reflecting the single sequential CPU, and
- Assignment statements — reflecting piping.

Imperative programs are sequences of directions (or orders) for performing an action. Imperative programs are characterized by sequences of bindings (state changes) in which a name may be bound to a value at one point in the program and later bound to a different value. Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings. Therefore, imperative programming is characterized by programming with states and commands
Which modify these states? Imperative programming languages provide a variety of commands in order to structure the code and to manipulate the states. Usually, in imperative programming languages, a sequence of commands can be named and the name can be used to invoke the sequence of commands. Named sequence
Of commands is called subprogram, procedure or function. When imperative
Programming is combined with subprograms it is called procedural programming. Imperative paradigm is supported by languages such as FORTRAN (introduced in 1954), COBOL (1959), Pascal (1970), C (1971), and Ada (1979)

➢ Object-Oriented Programming

The object-oriented programming is a generalization of imperative programming.
The conceptual model of this paradigm is developed from simulation of events. The main underlying idea of this model is: the structure of

the simulation should reflect the environment that is being simulated. If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations. As in the real world, objects interact—so, object-oriented programming uses message passing to capture interactions between objects [4].

A programming language supporting this concept and using objects is called object- based. Object -oriented programming languages support additional features, with the following most important ones:

- Abstract data type definitions are used to define properties of classes of objects.
- Inheritance is a mechanism that allows definition of one abstract data type by deriving it from an existing abstract data type—the newly defined type inherits the properties of the parent type.
- Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes.
- Dynamic binding of function calls supports the use of polymorphic functions. The identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects. It is focused on generality and reusability of the written code.

Comparing to other programming paradigms, object-oriented programming shifts the emphasis from data as passive elements defined by relations (as in logic paradigm) or acted on by functions (as in functional paradigm) or procedures (as in imperative paradigm) to active elements interacting with their environment.

Object-oriented paradigm is supported by languages such as Smalltalk (1969), C++ (1983), Java (1995), C#(2000) , Ruby(1995) , JavaScript(1995) , Ada (2012) , Jade (1996), Visual Basic(1998). Etc.

- **Functional Programming**

The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calculus. It allows the programmer to think about the problem at a higher level of abstraction—it encourages thinking about the nature of the problem rather than about sequential nature of the underlying computing engine[18][19].

A functional programming language usually has three main sets of components:

1. Data objects — such as a list or an array.

2. Built-in functions — for manipulating the basic data objects.

3. Functional forms — also called high-order functions, for building new functions
(Such as composition and reduction).

Functional programming languages are called applicative since the functions are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.

Functional paradigm is supported by languages such as LISP (1958), ML (1973), Scheme (1975), Miranda (1982), and Haskell (1987)

- **Logic Programming**

The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather
than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications [17][20]. The problem description is used by an inference engine to find a solution. More precisely, a logic program consists of:

- Axioms — defining facts about objects,
- Rules — defining ways for inference new facts,

- Goal statement — defining a theorem, potentially provable by given axioms

Logic programming is characterized by programming with relations and inference.
The programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computational resources than procedural and object-oriented languages.

Logic paradigm is supported by languages such as Prolog (1970), and Gödel (1994). Curry (1997) is a multiparadigm programming language merging elements
of functional and logic programming.

7. Statistics on Programming Languages

Statistical data can help in understanding the role of programming languages and paradigms in teaching of programming. Unfortunately, the exact statistical overview of the first programming language at different universities and colleges in the whole world is almost impossible to get. But there is some research related to the smaller patterns which could be used for making general conclusions. The data are being changed every year, so trends in using the programming language are more important than the present situation itself. Therefore, we will consider the situation and analyse the usage of programming languages in these years and make some conclusions.

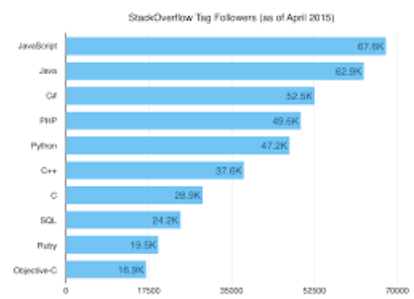

Figure 1 : Increase in the number of the users in each language in the year 2015.

The trends after the end of 20th century changed as the popularity of Java and JavaScript in IT industry significantly increased. There are a number of forums and blogs on Internet with polls concerning the best first programming language [9]. The results of these polls are interesting since they show the opinion of a wide range of people (not only experts), but these results are not discussed here.

8. CONCLUSION

Acquiring and developing knowledge about programming is a highly complex Process. Choosing right programming language is most important. This problem should be considered not only in terms of individual programming languages, but also in terms of different paradigms. In this paper we surveyed programming language paradigms in the light of computer science education, and discussed the problem of choosing a first programming language. it seems that nowadays the most popular paradigms are the procedural, with programming language C and procedural part of C++, the object-oriented, with languages Java , JavaScript and C++ etc. In any case, it should be always kept in mind that beside of specifics of some programming language one should focus on general programming ideas and concepts, while considering both basic and more advanced concepts.

REFERENCES

1. T. DeClue, “Object-orientation and principles of learning theory: A new look at problems and benefits”, Proc. 27th SIGCSE Technical Symposium on Computer Science Education, pp. 232–235–86, February 1996.

2. B. du Boulay, “Some difficulties of learning to program”, In: Soloway, E. and Spohrer, J.C. (Eds), pp. 283–299, Hillsdale, NJ:Lawrence Erlbaum, 1989.

3. Educational programming language, 2008, http://en.wikipedia.org/wiki/-
Educational programming language.

4. C. Ghezzi and M. Jazayeri, “Programming Language Concepts”, John Wiley and Sons, New York, 1996.

5. D. Gupta, “What is a good first programming language?” Crossroads, 10(4), 7–7, 2004.

6. L. McIver, "The effect of programming language on error rates of novice programmers", 2000.

7. L. McIver and D. Conway, "Seven deadly sins of introductory programming language design", Technical Report 95/234, 1995.

8. J. L. Murtagh and J.A. Hamilton, "A comparison of Ada and Pascal in an introductory computer science course", SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international
conference on Ada, pp. 75–80, New York, NY, USA, 1998. ACM.

9. Programming language trends, 2008, http://www.caffeinatedcoder.com/programming-language-trends/.

10. de M. Raadt, R. Watson, and M. Toleman, "Language trends in introductory programming courses", Informing Science InSITE, pp. 320–337, 2002.

11. A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion", Computer Science Education, 13(2), 137–172, 2003.

12. J. Rogalski and R. Samurcay, "Acquisition of programming knowledge and skills", Psychology of programming, pp. 157–174, 1990.

13. J. Spolsky, "The perils of Javaschools",2005,http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html.

14. M. W. van Someren, "Whats wrong? understanding beginners problems with Prolog", Instructional Science, 19(4–5), 257–282, 1990.

15. R. L. Wexelblat, "The consequences of one are first programming language", SIGSMALL '80: Proc. 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems, 52–55, New York, NY, USA, 1980. ACM.

16. P. Van Roy and S. Haridi," Teaching programming broadly and deeply: The kernel language approach", Informatics Curricula and Teaching Methods, 53–62, 2002.

17. A. Kumar, "Prolog for imperative programmers", J. Computing Sciences in Colleges, 17(6), 167–181, 2002.

18. R. Harrison, "The use of functional languages in teaching computer science", J. Functional Programming, 3(1), 67–75, 1993.

19. J. E. Howland, "Functional Languages and Introductory Computer Science", 1998.

20. A. M. Lopez, "Supporting declarative programming through analogy", J. Computing Sciences in
Colleges, 16(4), 53–65, 2001.

21. Basic Abstract Machine, 2015, http://en.wikipedia.org/wiki/Basic Abstract Machine.

22. Operations performed by the interpreter, 2015, http://en.wikipedia.org/wiki/Operations performed by the interpreter