Towards a Semiotic Framework for Programming Languages

A programming language is "an artificial formalism in which algorithms can be expressed. For all its artificiality, though, this formalism remains a language" [1]. Considering this property, Tanaka-Ishii motivates a semiotic analysis of the programming languages [3]. It is the only large-scale sign communication system not intended uniquely for humans: "The purpose is the communication of programs between computers, from man to computers, and also from man to man." [4].

We can arrange programming languages considering their proximity to the machine (low-level) toward the human (high-level). In the artificial ecosystem created by the Von Neumann machine the principal referent in low-level languages is the processor, with its specific binary dialect. Just upon this level we can recognize assembler languages, where a basic linguistic representation is introduced by using symbolic name for operations (e.g. STORE, LOAD, ADD) and by using addresses in order to refer to memory locations. Moving up we find high-level languages, in which the abstraction allows for the introduction of a "structured" form of linguistic representation based on the notion of "control flow" by means of conditionals and loop (e.g. IF, WHILE). Here we find an osmosis between human and machine semiotics, where the strict formal correctness of the machine side is balanced by the variety typical of natural languages.

In programmers communities a number of good practices has emerged in relation to the crucial notion of "readability" of the code. For instance, the so-called good practices prescribe the insertion of comments, i.e. natural sentences that are directed uniquely toward the ideal human-reader and that are eliminated in the compilation/interpretation process. Good practices prescribe the use of meaningful names for variables, where meaning depends on the ideal human reader.

All programming languages are Turing-complete. This means that all languages are able to express the same "things", but the variety of languages (thousands of languages in fifty years) demonstrates the need to express some of these "things" better (more easily, more efficiently) than others. In other words, there is obviously a connection to the Sapir-Whorf hypothesis in the relationship between programming language and what it can express. The explicit linguistic nature of programming languages, although little investigated by current semiotic literature, make them an interesting test bed for a theory of enunciation, that takes into account the roles played by the concepts of persona, time and space. From ethnographic point of view, many famous programmers consider the linguistic abilities, in the sense of natural language, as an important prerequisite to become a skilled programmer [2]. We briefly analyze the concepts of persona, time, and space in the imperative, functional and object-oriented paradigms.

Imperative paradigm is the older family of high-level languages: "... -imperative- here has to do with natural language: as in an imperative phrase, we say -take that apple- to express a command, so with an imperative command we can say -assign to x the value 1-" [1]. Here the subject of the enunciation "I" is an abstract entity, the subject-programmer who get into a relationship with an asymmetric and complementary subject "YOU", an abstract agent of calculus which implements the given orders. The memory is the reference space of the language: the names of variables establish a system of real mnemonic loci, in the double meaning of memory addresses for the machine and "placeholders" for the human interpreter. Thus, in the imperative paradigm, there is a clear opposition between an active dimension of the subject (agentive) and a passive dimension of the data (spatial). With regard to temporal dimension, imperative languages do not provide references to past or future. Every statement prescribes an action to be realized at the time of its enunciation: the sequence of enunciates coincides with the advancement of time.

In the functional paradigm, "computation proceeds by rewriting functions and not by modifying the state" [1]. Usually we can separate a functional programs into two sections: in the first section we
find a set of inter-related function definitions; in the second section there is a "request" to the "environment" (no specific "YOU" is present here) to compute the output value of a function over a specific input value. The programmer constructs an "imaginary geography" of functions that can partially or exhaustively be explored. In other words, the functions establish a space of possible relations, governed by a specific topology. This function space also absorbs the time: the first function-call is the trigger of a function-call tree into the space of the function. The basic assumption in the pure functional paradigm is that the order in which this graph is explored does not affect the final result of the computation.

The foundational assumption in the object-oriented paradigm is to represent the human conceptualization of the world. Following a classical Aristotelian perspective, the basic structure is a taxonomy of classes that organizes the world into objects, where an object is an entity with properties (its "attributes") and abilities (its "methods"). The program consists of two blocks: the first descriptive (metalinguistic) one which provides for the definition of classes; the second one in which the objects are invoked. The typical object-oriented syntax has the shape name.method(arguments), i.e. "subject! Do this in this way!". So, the "YOU" of the enunciation is not the calculation agent, but a plurality of possible receivers, i.e. the objects: this feature eliminates spatiality from the world, because data are encoded into object attributes. Considering time, the object-oriented paradigm provides a timeless description of classes (similar to functional paradigm) and moreover a "call", i.e. a sequence of instantiations and methods calls (similar to imperative paradigm).

Reference: