On The Need of A Linguistic Theory of Programming Languages

Eugenio M. Vigo
eugenio.vigo@upf.edu

Abstract

This paper claims that programming languages are a product of the same human faculty that generates natural languages and, therefore, they should also be taken in consideration by our discipline. So, as a first step in the construction of this new research program, this paper discusses the concept of linguistic competence within programming languages. I show that this is not a trivial question and that a linguistic view on this type of laguages reveals both similarities and differences between them and natural languages. This discussion is the starting point of a roadmap whose final goal is to answer the question of what makes programming languages languages and what do they tell us about language in general.

1 Introduction

Linguistics has been devoted, since its inception, to the study of languages and the human faculty of language. This, however, has been pursued restricting ourselves to natural languages, namely those languages that have originated in a spontaneous way as historical instances of the universal human faculty of language (Coseriu, 1987), and leaving aside those languages that have been created in a non-spontaneous way, with a specific goal in mind, namely those we usually call "artificial languages."

Artificial as they might be, artificial languages are also products of our faculty of language as they serve the same primary goal natural languages do: express concepts and make them communicable by means of abstract symbols. The more famous and widespread examples are Esperanto, Ido and Interlingua and others that have been created as *linguae francae*, each one with its own community of speakers. Other artificial languages, though, are created as part of a fictional world initially for a fictional community of

speakers, like the Elvish tongues devised by J. R. R. Tolkien for his fantasy novels. This type of languages are not as usable as English is in our world, but the facts are that the most prominent of these languages do feature a grammar and that there have been efforts to reconstruct them for further literary use.

Among artificial languages, programming languages are possibly the most interesting linguistic creation of our times. They are extensively used to express algorithms and concepts, there is an active interest in learning them, and there are more or less objective ways to measure the competence a programmer has in a certain programming language. Of course, these are formal languages that are designed with computation in mind, a fact that constrains programming languages in certain aspects like their relatively small amount of lexicon and their proportionally high amount of neological mechanisms that programmers are provided with in order to write their code.

I assume that programming languages are products of the same faculty of language that also produces natural languages and that they must be taken into account if we seek to understand human language as a whole. These languages may reveal properties of language that are yet unknown, for which they provide interesting possibilities of testing future hypothesis that "natural-like" artificial languages do not because of their marginal use centered on the development of the very same language. Programming languages, on the other hand, are used by programmers primarily for coding, i.e. for expressing computable ideas in algorithms, and as such means of communications, they ought to be objects of inquiry of linguists.

This paper is programatic by nature. The question that we linguists should try to answer first is what makes programming languages languages, as they are used by humans not only to "tell" a computer what to do, but are used to communicate with other programmers, as the growing Free/Open Source Software movement has shown. In such collaborative projects, ideas are communicated between humans using these programming languages in a worldwide scale, thus proving that there are community of "speakers" of these languages much like in the case of natural languages and unlike the more artistic type of artificial languages, whose usage is limited to philological experimentation. This opens up the question on what does it mean to be "competent" in a certain programming language. The question of competence proposed by Chomsky (1958) redefined our ideas about what is to "know" a language and paved the way to the discussion about the faculty of language itself, so it seems a good starting point also for this topic. As will be shown, defining what is to know a programming language is not entirely trivial and the answer to this question conditions the possible research paths that may be considered for the proposed linguistic research of programming

languages.

2 The Linguistic Nature of Programming (Languages)

Programming consists in describing an algorithm that is the solution to a problem that is computationally solvable, usually expressing the aforementioned algorithm in code that must follow the grammar of a given formal language. The languages used to write programs are highly specialized for this task and designed in ways that make them easily parsed by the interpreter, i.e. the computer program that is in charge of transforming the code written in the programming language to the instructions understood by the microprocessor, which vary across microprocessor architectures.[1]

It is possible to program a computer without using a programming language, but directly instructing the microprocessor what steps it must take to solve a specific problem. This requires the programmer to know, for example, that the *add* instruction in the x86_64 architecture, the one found in most desktop and notebook computers, is 00000000_2 (hexadecimal 00_{16}). If the programmer wants the computer to calculate $4 + 3$, this particular instruction requires the value 4 to be stored in a specific register (*rax*) in the microprocessor for the computer to be able to find it and add 3 to it. This means the programmer must first know the instruction to set that register to the desired value (4, in this example), namely *mov* or 00100001_2 (hexadecimal $A1_{16}$). The result of the sum will be stored in the *rax* register where the first operand was stored, according to the definition of the instruction. Certainly, there is Assembly code, but its structure is just a more human-readable 1:1 translation of binary code for each kind of microprocessor architecture, such that Assembly code for ARM processors translates into binary code that an x86_64 processor is unable to understand.[2]

[1] The distinction between *compiled* and *interpreted* languages is irrelevant for this distinction. Compiled languages require a "compiler" that transforms the source code into binary code that is interpretable by a runtime environment, which may be the operating system itself or some kind of virtual machine. Interpreted languages are those that only require using an "interpreter," which acts as a runtime environment. In this paper the discussion is centered around the fact that the computer acts as a potential addressee that must be able to understand the "message" it is being "told" to. It is irrelevant from a linguistic point of view whether this is the result of two distinct steps or just done in one, as it always results in the source code translated into the instructions that is understood by the microprocessor.

[2] There is an additional layer of complexity to all of this, namely the binary format used by the operating system to include other kinds of information to make the operating

Programmers use programming languages because algorithms map much better to expressions in a conventional system of symbols rather than "thinking like a computer does," namely in binary. Programming languages provide programmers a basic lexicon that is usually referred to higher-order concepts like variable types, native data structures, etc., and mechanisms that allow them to create new entities, because, unlike the use of natural and natural-like artificial languages (e.g. Elvish, Esperanto, etc.), programming generally requires creating new entities on top of the ones provided by the language or on top of entities created by others.

Learning how to program is done by learning a programming language. This is exactly as in the case of natural languages: a child that is learning how to speak learns to do so by learning a language, namely the native language. Every child is assumed to have the faculty of language that is natural to all humans, but this faculty is only manifested as soon as the child starts to utter his first expressions, i.e. utterances with an *intended* meaning, even if ungrammatical. The feedback from other adult speakers leads the child to learn the grammatical principles of his language, for example, by setting values of universal parameters as defended by Chomsky (1981, 1995) or by reordering violable principles in a priority hierarchy as defended by the OT view of language (Prince and Smolensky, 2004; Kuhn, 2003), among other explanations. This process continues until the child is said to know the language, i.e. when he has acquired the required level of competence in his language so that the expressions he produces comply with the minimum expected by the community to accept an expression to be considered well-formed (i.e. what is usually meant by *grammaticality*).[3] The very same occurs with the novice programmer, but with the crucial difference that the expected minimum is codified in a deterministic program that is going to accept the code or not: the interpreter.

Therefore, competence in programming implies competence in a certain programming language, exactly like linguistic competence implies competence in a language. A programmer is said to know the C programming

system work in a more efficient way; e.g. in a modern Linux-based operating system, ELF is the binary format most widely used, but this format is not directly usable in other types of operating system, regardless whether the architecture is the right one or not. I am leaving this problem aside because it does not belong to the discussion I intend to pursue here.

[3]Linguistic competence does not have its upper limit in grammaticality. Stylistic development, learning the conventions of each different textual genres, etc. are also part of the linguistic competence; otherwise, all children would be able to write academic papers as soon as they become minimally competent in their respective languages. The more competence is acquired, the more complex conceptions the speaker of a language will be able to express.

language, for example, if he is succesful in writing a well-formed program in C, i.e. a piece of code that a C interpreter may interpret correctly. In the particular case of C, that would be both the compiler that turns the source code into object code executable by the operating system as well as the C runtime that makes it possible to run that executable, because this language strictly separates both stages. If the code is syntactically ill-formed, the code will not be accepted by the C compiler, yielding a result that is akin to ungrammaticality in natural languages: the expression is unparseable according to the grammar of the language, thus it cannot even be considered to be part of that language (maybe part of another language, though). The code, however, may be grammatically well-formed, but if during the execution, namely the process by which the meaning of the code is decoded either by a computer or by a human that is able to follow the code, a semantic error is found, then the programmer might be considered not to be fully competent in that language because of having created code that is nonsensical or does not mean what was initially planned to mean.[4]

There is a subtle but critical point to make here. A programming language is always composed by a set of elements that are recognized by the compiler, which have to comply with the grammar of the language: primitive atomic expressions (mainly primitive data types), as means of combination of these primitive expressions with each other and as means of abstraction that take combinations of elements and subsume them into one single entity that behaves as a primitive expression but is actually not (Abelson and Sussman, 1996). While the first two kinds of expressions are respectively equivalent to the lexicon and syntax of a natural language, the third one does not fit any category posited by linguists, as in natural languages there is almost no such need to create new abstractions: the lexicon in a natural language already includes thousands of words that can be used for to refer to thousands of different entities and when a new word is "needed" to be coined, either an old one is extended for that purpose or a new one is created by using morphological processes such as affixation or nonconcatenative morphology, which operate on a lexical root to modify its meaning or confer it a certain grammatical category. On the other hand, in programming languages, the lexicon provided by the language itself is quite sparse and programming consists in creating new entities by describing their nature precisely, by coding the algorithm that represents the concept that entity actually means, i.e. it requires using means of combination and also means of abstraction in or-

[4]In natural languages, nonsensical utterances are sometimes accepted in favor of other criteria, e.g. literary quality or ease of communication in a certain context. This is impossible in the context of programming, as computers are only able to interpret expressions that follow the principles of Boolean logic.

der to encapsulate an aggregation of meanings into one single independent meaning for a new form that will blend in with the rest of other entities as if it was part of the language.[5]

The entities created by the programmer in some program blend into the set of the functions and data structures already provided by the language that are part of the so-called "standard library". A standard function in C like $fopen()$ is syntactically indistinguishable from a user-defined function that may be called $create\_file()$; the only way to know that the former is a standard function and the latter is not is by knowing that only the former is defined by the standard library. Both must be called by listing its arguments enclosed by parentheses that follow the name function and both share the properties that define a function in C or any language whatsoever. Moreover, the standard library is not implemented internally to the interpreter as keywords and the syntax of the language are, but in an external module that is accessed when executing the program, exactly like any other external module that implements functions and data structures for a certain specific task (e.g. calculus, graphical user interface creation, physics engines for games, etc.).

This leads to the question of what "knowing the programming language X" means. In other words, how is competence in a certain programming language measured. Of course, competence in any language (natural or not) implies knowing its syntax, so basic competence in a programming language implies the knowledge of its grammar. However, the parallelism between natural languages and programming languages gets lost when it comes to lexical competence. In natural languages lexical competence may or may not be part of the minimum competence, depending on whether a formalist or a cognitivist view on language is adopted, but, in programming languages, knowing which are the primitive expressions (which are lexical in nature: keywords, operators and other atomic expressions) is crucial to write a program so that it is accepted by the interpreter. The problem is that, at least from a sociolinguistic standpoint, no programmer will consider another programmer to be competent enough in some language if he is not able to use the standard library up to a reasonable level, because it is highly improbable that a program may be written in any language without any reference of any sort to some function or data structure defined by the standard library. So the linguist reflecting on these topics must make a choice here and decide whether the standard library is part of the minimum required competence in some programming language or not.

[5]The way new entities are defined in programming languages is certainly metalinguistic (Carnap, 1956; Jakobson, 1981): new entities are described by the same language that is being extended.

A formalist view would probably exclude the standard library from the knowledge required for claiming competence in some programming language. This, however, limits the linguistic study of programming languages to a purely syntactic angle. On the other hand, including the knowledge of the standard library into the definition of linguistic competence in programming allows discussing how the standard, preset yet not grammar-related entities of a language define the basic level of abstraction of a language, how the lack of certain elements in the standard library requires programmers to create new "de facto" almost standard libraries for a certain language (e.g. Qt or Boost for C++), how and why programmers choose different alternatives available among standard functions (e.g. *gets*() is considered an unsafe procedure in C, thus resulting in what is usually called a "bad practice"), etc. Without any intentions to extend this hypothesis to the study of natural languages, I believe the second option is more likely to be more fruitful.

This concept of linguistic competence allows studying programming languages as tools of communication between humans that form different communities of "speakers" of these languages. Code is not meant to be read by a computer, but also to be read by other humans. This is the reason why the standard library and well-known third-party libraries are better considered to be part of the language, rather than restricting the concept to what the compiler defines to be the language: level of competence in these defines the level of involvement of a programmer in a certain community of speakers. Moreover, it also allows the study of the pragmatics of programming, if we enter into studying why some code, despite being well-formed, is deemed inadequate ("obfuscated" or "unreadable") and the because of readability issues (e.g. not following indentation rules or guidelines for naming variables) or unusual, esoteric practices (e.g. using bitwise operations to perform simple arithmetics).

3 Future steps

Without a roadmap for future research, this theoretical proposal would be completely sterile. Competence is a central part of linguistic analysis, but as soon as the discussion about what means to be competent in a programming language is clarified and the implied question about what must be considered part of a programming language is answered, the discussion about the parts that conform the whole of a language of this type must begin.

In the first place, a semiotic theory is required. Linguistic expressions in natural language are fully abstracted away from the material reality of its meaning: they are "arbitrary" (Saussure, 2005) or, in Coseriu's (1977)

words, just "historically given," in the sense that the only possible explanation for a certain expression to mean what it means is due to the history of the expression, both past and present. In any case, it is evident that in natural languages there is no necessary cause-effect relation between the form and the meaning of an expression. This makes linguistic expressions in natural languages able to refer to any possible instance of reality by just using a certain expression or creating it without limits of any sort. Expressions in programming languages, on the other hand, are conditioned by the abstraction level of the language; a language like C includes memory addresses as part of its primitive entities and, unless a new programming language is implemented in C (e.g. the "standard" implementation of Python) by implementing a parser that hides away all elements of C from it, code written in C will always make references to those low-level concepts that are native to it (e.g. high-level C libraries like GLib or GTK+ require the use of memory pointers in their implementation of object-oriented programming). This does not limit programming languages in any way: assuming them to be Turing-complete, all of them are able to express the same programs, although the way programs are expressed in languages.

The discussion cannot be limited to semiotics or philosophical issues about programming languages. If the working hypothesis that these languages are also products of the same faculty of language that produces natural languages is true, the study of the grammar of programming languages should show similarities with the grammar of natural languages. However, for this to be possible, the study of the grammar of programming languages should not be understood as a study of the explicit formal definitions of the grammar that are then used for implementing compilers (i.e. their Backus-Naur Form), as that would be redundant, but an application of the concepts that are already used in linguistics onto the syntax, semantics and morphology of these languages.

Hopefully, research like this might be useful for the development of new programming languages; for instance, knowing the linguistic foundations behind formal languages might shed light over how humans express computational problems and whether the ways of expression chosen help finding a solution (namely, an algorithm) for a certain problem more easily or not.

Those, though, are accidents of the real main goal behind this research. The incorporation of this new type of data into the field of linguistics may be helpful to build a better theory of language, also for explaining the data we already know about natural languages. The current state of affairs in linguistics favors the discussion of specific data within the convenient theoretical framework that may give a principled explanation about them, but a jump forward is required in the theory of language in order to advance

our theoretical knowledge into a coherent set of rules that are common to linguists, akin to what mathematics is to physicists nowadays. This cannot be accomplished if artificial yet widely used languages like programming languages are systematically ignored from the discussion, regardless of their artificial origin as opposed to the spontaneous one of natural languages. If the generative hypothesis is to be taken seriously, programming languages and other linguistic artifacts created by us humans must be made part of the discussion.

References

Abelson, H. and Sussman, G. J. (1996). *Structure and interpretation of computer programs*. MIT Press, 2nd edition edition.

Carnap, R. (1956). *Meaning and necessity: a study in semantics and modal logic*. The University of Chicago, Chicago.

Chomsky, N. (1958). *Syntactic structures*. Mouton, The Hague.

Chomsky, N. (1981). *Lectures on government and binding: the Pisa lectures*. The Hague, Mouton de Gruyter.

Chomsky, N. (1995). *The Minimalist Program*. MIT Press, Cambridge, Mass.

Coseriu, E. (1977). *Tradición y novedad en la ciencia del lenguaje*. Editorial Gredos, Madrid.

Coseriu, E. (1987). *Gramática, semántica, universales: estudios de lingüística funcional*. Editorial Gredos, Madrid.

Jakobson, R. (1981). Linguistics and poetics. In Rudy, S., editor, *Selected writings*, volume III, pages 18–51. Mouton de Gruyter, The Hague.

Kuhn, J. (2003). *Optimality-Theoretic syntax: a declarative approach*. CSLI Publications, Stanford.

Prince, A. and Smolensky, P. (2004). *Optimality Theory: constraint interaction in generative grammar*. Blackwell Publishing, Malden, Mass.

Saussure, F. d. (2005). *Cours de linguistique generale*. Payot, Paris.