ON COMPUTER LINGUISTICS AND COOKERY

Computer languages bear a certain resemblance to natural (i.e., human) languages. They are written using words, spaces and punctuation; they have a definite grammar and semantics. However, for the man in the street, computer languages are often seen as an esoteric gibberish that can be understood only by the initiated — a lingo without a Rosetta stone.

Yet, there is a simple analogy that helps understanding the nature of computer languages: food processing. As a matter of fact, a *program* (i.e., a text in a programming language) is very much like a cooking recipe. Both consist of a sequence of instructions that, when followed, produces interesting results such as tomorrow's weather forecast or a beef stroganoff. The programmer is the chef, who designs or decides on recipes. The computer is the line cook, who slavishly obeys and performs the chef's orders. Basic operations that can perform the computer are analogous to kitchen equipment. The computation data are the food ingredients. And the user of the computer is the guest enjoying his meal. Note that the same result (i.e., dish) can be obtained by different programs (i.e., recipes).

The main concept of programming languages is that of the *command*, e.g., "beat eggs in large bowl", or "add sugar and salt". Commands often include *control instructions*, that are conditions for performing or repeating an action: "if at high altitude, boil for five more minutes", or "heat until hot but not boiling". Names can be given to sequences of commands that are frequently used. Thus the routine preparation of a pâte brisée or chicken stock can be specified once and reused thereafter in many recipes by just mentioning its name. Enriching the language with new names is a crucial feature of programming languages, which typically comprise less than a hundred basic words.

Languages that mainly rely on commands are called *imperative*. Today, most computer applications are programmed using imperative languages. But there are other programming principles. For example, *declarative languages* express *what* to achieve more than *how* to achieve it. For example, given a heap of apples of different sizes, taking 2kg of apples can be expressed as "find a subset of apples so that the sum of their weight is within 50g to 2kg". In general, there might be zero, one, or many solutions to this kind of problem; the computer tries to enumerate them all. There can also be solutions that the computer cannot find in practice because there are far too many possibilities to try. Similarly, it might be hard to determine how to prepare a dish given only its final description. *Object-oriented languages* rely on another programming principles. They offer a support for modeling *objects* (whether concrete or abstract) as well as their functionality. They associate data and routines that operate on those data. For instance, an "apple object" can have a weight, a price, as well as standard procedures to peal it and to cut it into slices. *Parallel languages* let the programmer split a program into simultaneous *tasks.* For example, a line cook can prepare a pie crust while another one peels and cuts the apples. This reduces the total preparation time.

Though few programming principles have been invented, there are many computer languages that embody them. In fact, computer science priests must have been punished for deifying an apparatus, for the Babel malediction also affected computer languages. There exist today well over two thousand languages — although only a dozen of them are commonly used. However, most of them are actually made up from a combination of a small number of concepts; a large part of the diversity is only due to slight syntactic and semantic variations as well as specialized features.

The reason for such a variety is that computer languages have specific functions that natural languages do not. Natural languages fulfill two main functions: communication (i.e., conveying information, emotion, etc.) and conceptualization (i.e., forming, organizing and developing ideas). These two functions also exist for computer languages, but in a very limited form. Programming languages can be used to communicate only basic orders to the machine. You cannot tell a joke using a programming language. Computer languages also include facilities for structuring basic values into complex data. But only anthropomorphism can make us consider as ideas what merely are models.

Because all natural languages have rich communication and conceptualization abilities, they seldom have exclusive uses. French is not the only language for writing recipes. Correspondingly for computers, *general-purpose languages* can express solutions to any problem. Still, they usually come with a style of writing that makes programming more or less effective for certain types of problems. "What is this language good for?" is a question that makes sense for computer languages. "Universal languages", capable of elegantly treating any problem, are still the unreachable Holy Grail of a handful of researchers in programming languages.

Although natural languages are not specialized to particular uses, technical vocabularies do develop to capture precise concepts and provide practical communication means between specialists: "deglaze, then clarify". Similarly, *domain-specific languages* focus on definite application areas and cannot express all kinds of computations. However, in their domain, they enhance features such as productivity, usability by programming novices, expertise re-use, quality, safety, efficiency, maintainability[1], etc.

Such features are crucial in the computer industry. In other words, although one can speak for pleasure, programs are generally written for money. The choice of a programming language thus depends on its ability to *easily*, *efficiently* and *correctly* express how to compute solutions to given classes of problems. In terms of cookery, a language should be such that: it is easy to quickly write new recipes, even for people not well-versed in cooking; recipes can yield fast, cheap, and good quality food; and it is difficult to write recipes that make no sense. Some languages even help guaranteeing that a program meets a specification, i.e., that following a recipe will actually lead to the tasty targeted dish.

This correctness issue is an important concern because cash flow as well as the life of human beings can be threatened. You do not want to accidentally blow up an Ariane 5 rocket every day, nor to poison your guests because of an unsuccessful recipe. The fact is that writing a program without errors is difficult: putting together good ingredients does not systematically yield a good dish. Often, to make sure that the program is correct, the programmer has to explicitly control important program steps — there can be billions of them — and check if anything goes wrong, i.e., he has to taste the preparation at each step of the recipe. Fortunately, computers themselves can offer linguistic support to track common mistakes. They can check that programs parse according to the grammar of the language: “hot until heat” is discovered incorrect. They can also detect if the programmer is mixing apples and oranges: “chop water” makes no sense. Another concern is program termination. For instance, a recipe saying “beat until stiff” may not end if applied to yolks rather than egg whites.

Correctness is an issue because a computer is basically dumb. It does not have the enormous amount of common knowledge that humans accumulate in their life. It must be told everything, down to the most evident detail. The reason is that a computer mainly consists of one or several processors that only understand a primitive instruction language. This *machine language* (which consists of the famous 0’s and 1’s) is mainly based on elementary memory transfers, simple arithmetic operations and control instructions. Early computers were directly programmed using machine languages, but that was soon considered too rudimentary. Solving a problem by hand-writing a machine-language program required giving full particulars and manipulating obscure data. For example, one does not command “heat water” but “open cupboard, take pan, move pan under tap, open tap, etc.” More expressive languages were thus designed so that the programmer would not have to go into such tedious and trivial details. Indeed, up to a tenfold leap in productivity was observed. For this, automatic translators (also known as *compilers*) were developed to bridge the gap between machine languages and these new, “high-level” programming languages. Although even more advanced, today’s languages still require making explicit a lot of details. They also have little flexibility: knowing “spaghetti” gives them no clue about “spaghettini”. More importantly, computers do not have the ability to automatically abstract: knowing how to make an apple pie does not help them to make a pear pie. What is possible however is to explicitly tell them how to make an “*x* pie” and then ask them to follow the recipe either with *x* = apple or with *x* = pear.

Correctness also supposes a clear semantics. Natural languages are inherently ambiguous; the same word can have different meanings depending on the context and on the interlocutors. In practice, this is regarded as a minor issue — sometimes even as a richness — and people generally tend to understand each other. Concerning computer languages, the potential for misunderstanding is considered is serious flaw: “half a spoon a salt” has not the same effect whether it is understood as a teaspoon or a tablespoon. The same program should have exactly the same behavior on different machines — this is known as the *portability* problem. Unfortunately, that is often not the case. The reason is that for a single programming language to be understood by different processors, different translators must be developed. Because there rarely is a precise enough specification of the semantics of a programming language — some do exist though —, different companies have developed translators with slightly different semantics. As a result, the same program can have different behaviors on different machines. Moreover, these companies often have developed specific language extensions for their clients, which reduces the portability of programs. This phenomenon has given rise to programming language *dialects*. Just as dialects of natural languages reflect regional or social particularisms, dialects of programming languages reflect technical choices made by particular companies or specialists in a domain.

The search for languages capable of adequately and safely solving classes of problems explains the need for a variety of languages, but not how such a variety can actually exist. What makes diversity possible is that it is relatively easy to make up a new computer language. That is not true for natural languages.

Natural languages originate from our very humanity. They have been molded by man’s physiology (most of them are spoken) and perception of the world. Because man is versatile, languages are subject to constant and

[1] Maintenance can represent more than half of the total development cost of a program.

continuous evolution. Many factors of this evolution are purely linguistic. For instance, irregular verbs often happen to be frequently used words, which exposes them more to the likelihood of mutations. But most importantly, natural languages evolve for reasons that are sociocultural, historical, economical, political, etc. This evolution is not a conscious process. In fact, very few languages have been knowingly made up by men, and among these almost none have been actually used (cf. Esperanto). More generally, authoritative decisions have had little impact on natural languages. And institutions that make recommendations concerning correct language usage (e.g. Académie française in France, Accademia della Crusca in Italy) daily have to live with it. In any case, it requires generations for changes to take root.

On the contrary, computer languages have a very short history. They are neither spoken nor driven by society at large. They have been invented by small groups of people, sometimes even by a single person. Some languages, like *Java*, have become popular in a few years only thanks to good marketing campaigns. Habits and "cultural" barriers may slow down the diffusion of new languages though, especially when they involve new programming principles or operations. For example, recipes involving microwave ovens are still few and simple. Another reason is that programmers are willing to learn a new language only if it increases their value on the job market. You do not want to learn various ways to cook pork if you live in an Islamic country. Languages can be imposed by companies though, to increase productivity and interoperability, just as a natural language can be the instrument of national cohesion. Besides, there are international standardization actions to prevent the expansion of dialects, just as the promoting liters and kilograms, as opposed to spoons and cups, helps making recipes more precise and reproducible.

Given a language, there is also an "art of computer programming"[2]. But, just as there is an art of cooking, the art of programming relates more to craftsmanship than to actual art — Brillat-Savarin excepted maybe. Unlike for other forms of art, usefulness plays an important role in the aestheticism of programs and recipes. However, there is a sense of beauty besides the quality of computed results or prepared dishes. Beauty in a program stems from its internal structure, its style, as well as its presentation. An elegant program splits large problems into sensible sub-problems, which can possibly be reused in other contexts. An elegant program is also easy to understand and exploits well the programming principles and basic operations offered by the language. Finally, an elegant program is well presented: the text of the program is indented to highlight the sequencing and the importance of commands. Additional comments (written in a natural language) also explain the nature of the computations and the set of inputs for which the program produces interesting or meaningful results. Similarly, a fine recipe not only produces a succulent dish but also explains clearly each preparation step. It precisely lists the ingredients and quantities, the number of people it is intended for, as well as the cost, the difficulty, the duration of the preparation. As for other form of art, there is no satisfying metrics for program beauty. Programmers have their own writing style, which they are often proud of: "my program (or even my language) is better than yours". It is written that aestheticism must be a subjective concept.

That is the situation today. But what about tomorrow? Will the computers always stay dumb, with limited ability for communication and conception? Or will the line cook eventually become a chef? Regarding the "mechanics" of computer, little is to be expected: chemical and optical alternatives that are being considered for the future simply mimics the mechanisms that are realized today with electronics. Regarding computer languages, there are formal evidences that no programming language can solve problems that others cannot; they are thus all somehow equivalent. Therefore a breakthrough, if any, is going to require a major conceptual step. The present pace of progress in this area is such that human cuisine still seems to have a bright future.

Renaud Marlet, July 1999

For Philokles, 2-99

[2] After the name of a famous book series by Donald E. Knuth, one of the major living figures of computer science.