Understand Programming

by Set Lonnert

Bottle dryers are technical constructions: since 1914 a bottle dryer is also a *piece of art* by Marcel Duchamp. The computer is a technical construction, and maybe it's worthwhile altering the perspective, suggesting one further away from the regularly adopted technical one.

People usually communicate in civilized manners through interlocution with each other — other pleasant or unpleasant possibilities not to be mentioned. With the computer a linguistic link is dreary but mighty: power over the creature seems to us sometimes omnipotent. Other times though, it fails our anticipation: it wasn't at all as we *thought* it would be. When the computer and it's environment thoroughly been excavated as technical phenomena and found free of guilt, the blame is to be put on ourselves. It's laid out on our own design-technical ability — or rather lack of it. Surely a proper judgment. But unsatisfying as one component will be the scapegoat of something even more intriguing.

The traditional analysis render the conviction that, has a problem already been solved in the abstract, then implementation is of a secondary nature with its own difficulties to conquer. Separate concrete programs are at locations considerable distant from their birthplaces at mind-formed, conceptual structures of solutions. Such explanations are reasonable if considered looking through a 'technical' lens. No harm coming from the technical view as such. But the consequences from it, the applications out of proper proportions, qualify the view to confront severe critique.

Is there patience and tolerance on behalf of the reader, he or she may approach the sketch of a hypothesis containing even more fragility all in all, but as I suppose, also implicating consequences of some interest. A point of departure is the problem of program-design; here: *constructing programs as we thought of them.* Let's take this thesis very literally. To confabulate with the computer one ought to understand its language.

The Greek verb-phrase 'hermeneuein' has at least three distinct denotations: to say, to explain, to translate. Radically simplifying things, the wide famous professor of linguistics Noam Chomsky, designates innate grammatical structures sorting in words into sentences as 'linguistic competence'. Often it though happens, that we in utterance slip, and out comes in worst cases gibberish. Performance is apparently something else than competence, as we often 'know' how it should have

been spelled out. Below, as it is a more empirical sort of definition and also with a more narrow purpose, it opposes essential elements made in the observation by Chomsky. But it may in part more smoothly fit the correspondence between the mentioned concepts. The definition points toward a kind of 'modest program-language comprehension' on behalf of the programmer. Rather a mental disposition: the ability of giving program-interpretations.

(1) *To a-understand a programming language is the ability to hermeneuein in natural language what happens when and how in an arbitrary selected program written in the programming language, at reading the syntax.*

Sure, even the computers themselves of today would handle parts thereof, if we for instance replace the Greek term with 'translate'. On the other hand, they may not cope with, at the same time give descriptions of semantical errors, or understand what problem the program was suppose to deal with — as well as the programmer may not necessarily be aware of. Computer behaviour manifest itself, it appear, through programming languages and arbitrary chosen programs at the face of the programmer, and his or hers a-understanding. Discrete from ordinary solutions in computer science, the definition doesn't refer to for example operational or denotational semantics. The natural language coupled with common-sense could be in need of strengthened support from a collection well-comprehended concepts. Hopefully presentable with none or a minimum of mathematical foundations (or the like) beneath.

This isn't all there is to it. Yet another, another more problem-penetrating and active understanding must be added; because not only must one read, but also write programs.

(2) *To b-understand a programming language is the ability (principally) to construct or reconstruct a program in the programming language to an arbitrary given problem, under the assumption that one knows of a programmable solution, i.e. coming to be a program, corresponding with the problem.*

The opaqueness of "principally" within the parentheses, could be replaced by a restriction to what programs are considered legitimate (now applied to ability). This because one could *principally* know of programmable solutions to infinite problems with infinite procedures, but humans would be unable to finish them within their lifetimes. Take note of "reconstruction" which doesn't mean solely description, but something wider. More over, each problem have several solutions. From one there mostly derives several problem-solutions (v-solution-1, v-solution-2, ...) and a number of program-solutions (p-solution-1-1, p-solution-1-2, &c. from e.g. v-solution-1). The tacit assumption made, takes the origin of v-

solutions to be in the spirit of traditional empirical science: some problem, then followed by research, and finally solution. We form empirically based theories (or perhaps even more rational types) to animate reality on the computer.

Let's hesitate for a moment reflecting on a somewhat strained, egocentric and in fact rather deceptive example — as I haven't come up with one preferable. Assume I have the problem of locating certain books in my library. Inside ordinary libraries one could ask the librarian, look up dictionary entries, make use of bibliographies, and all sorts of catalogues &c. One makes ones orders and probably receive things ordered. Therefore a classical author-, title- and subjectindex, dialogue structure of the librarians and their clients &c. could very well be the natural foundations of a theory.

Now then, it is my personal library, and to my mind books are remembered by me not always by name of authors, but by pictures on the envelopes, colours, names of persons mentioned inside the books (sometimes fictitious), headlines and all sorts of, in usual cases to the ordinary man/woman in the street whoever it may be, inessential details. Also, I evaluate some books as of higher value than others. Certain writings are covered by my own definitions across wide subject areas, as 'historical literature', some very limited 'apart literature'. The solution of the problem, the theory of my capricious personal librarian nature, will be formed in observations of myself (psychologically) and by myself, when trying to select or locate the right topics out of the shelves.

The theory may be generalized in parts, more cases (persons) included and so forth, which becomes further research. Rational theory of librarian-customs is likely to be radically divorced in central aspects from my ego-psychological one. Likewise the program-solution, ought to at least in appearance, have a different character than simulation of traditional card-registers.

Regrettably the understanding appear only to be applicable to an ideal programmer having no limits other than what definition permits — a theoretical boundary. Therefore an important postulate will be: the understanding is supposed to be non-uniform over the implementation-extension (where the 'implementation-extension' of a programming language is stipulated as the class of programs written in it). The limits of understanding becomes clear as we are not able to handle any sort of complicated assignments which the definition allowed, even if we had time resources unlimited. So there is also the practical aspect: the practical restrictions drawn by the computer (e.g. the natural numbers not in realistic representations: *all numbers*, though potentially and formally as in a recursive definition), non-uniform understanding of the programming language extension (cf. PL1 and programmers knowledge

of all elements of the language) and defective understanding of other (operating) systems &c. Relevant is also interfacing-problems between programs and users.

Upon the above substructures, theories for revising programs could be raised.

(3) *To be able in solving some internal problems given program and programming language, one should a-understand the program-language of the program, and also be able to revise the program to a new p-solution according to the same v-solution the p-solution the program before corresponded with.*

Scilicet, to some problems it's necessary to comprehend what hidden v-solution it is to revise the program. (No surprise then, if the theory is correct in some sense, programs demands good documentation!) As already been mentioned, there are program-solutions in an almost perfect relationship with the a-understanding, where the problems lays outside its scope but e.g. efficiency within.

(4) *To solve other much internal problems of a given program in a programming language, one should a-understand the programming language of this program, and be able to revise the program to a new p-solution according to the sustained exterior function of the p-solution the program before corresponded with.*

The exterior function will only deliver exterior criteria. But it will demand the existence of all possible situations to be kept, for two programs in- and out-going information respectively, identical. Profound knowledge is sometimes inevitable criteria.

(5) *To be able in solving external problems of a given program and programming language, one should a-understand the programming language of the program and b-understand the programming language of the original problem, and also revise the program to a new p-solution according to a new v-solution the previous revision resulted in (more or less divorced the earlier).*

My personal aid is more or less remote from e.g. counterparts in university libraries. Certain internal problems of my program resolves without dividing attention upon a new theory. Anyone performing revision must though sometimes have knowledge of my personal theory, sometimes not as for example to make some search-routine more efficient. It could occur external problems. For instance, say a friend of mine, which doesn't react as I do, want to make some use of the personal tool of mine. Maybe revision of just some insignificant part, or maybe a complete revision, new research and new programs, will be found

necessary. Wrongly estimated revision could of course with this procedure lead to severe future problems.

Direct program-construction may be formulated as: iterated program revision of suggestions. One after the other desk-produced suggestion adds and become revised. They add up to series until at last a programmable solution reaches, to be further explored and revised in forms of programs ready-made. A definition of 'program-development' could be: evolutionary (increased) understanding of internal and external problems, followed by revisions of programs.

But something's missing. The highly important phase of construction was not explicated in (2), with the vacuous terms 'construction' and 'reconstruction'. Likewise the concept of 'being able to revise' was left unexplained in (3-5). *Revision* is foremost by it's nature linguistic, which doesn't automatically include all of the problems to be of such kind. *Construction* on the other hand utterly depends upon reality. But genetically: have they anything in common?

A pill for the cure could be by prescribing some sort of 'constructivism'. Constructive proofs are well known from mathematics, and in connection with Prolog one speaks today of construction. With somewhat vague hints one could say it would here mean some rule-system to generate programs. If it were to be applied in the context above, dependent upon the programming language, and also some praxis. Concerning praxis there is interaction between our thoughts, the desk-products and our interanimation (interaction & interreaction) with the computer. That is, the control of our social habits and relations, with the computer and between programmers, often understood as conventions or standards. For example it musn't be that revolutionary ideas perhaps would be suppressed in favour of solving problems good old ways. But it could be that we don't try to find new solutions to already satisfactorily solved questions. Concerning rule-systems, I imagine a mixture of empirical and rational rigid rules guiding the designer. Rules that will be based on elementary concepts from computer science become: variables-environments, names, organized data, operations, control &c. The foundation now probably change its centre main point formerly mathematical into a linguistic. Take the above mentioned 'control' for example, which is rather theoretically well acquainted through theories of recursive functions. Good, let us keep these, although they to the programmer may not be of any use in actual cases, they are indeed much to general. More important however is if the processor-unit was 'monarchical' of von Neumanns type, and each little action or flow of instructions needs to be declared as in for instance C. Compare this with situations of old-style sequential Prolog, where it's rather declared what's *not* in control (partly by 'not' and 'cut') or consider both the back-track mechanism and the ordinary forward processing mixed. Or think of

parallel architectures, still assuming languages depends very much on machine structures. So from the exterior point of view, abstract understanding of programs, we travel to the inside, penetrating deeper and deeper into the concrete exemplar of machine we are working on. Behaviour, sensible through change in program, unfolds control.

The question is whether the rules in their most rigid form (directly) from the linguistic point of view as I choose to call it, simply becomes the grammar of programming languages? In cases less strict, perhaps the rules oscillates to and from easier translations between theories and programs, and to and from more complex methods of analysis between theories and programs?

The purpose of all this proposed, comprehensible or not, was to avoid common 'inner' technical methods of assuming either some mathematics, or some abstract properties (like sequence-iteration-selection) of present programming languages (computers) &c. for the purpose of systematically solving the problem of constructing programs. Instead the initial idea was that it's from the outside, by side of the programmer facing the computer, and the *concrete* exemplar, where it *could* take off, and maybe also where it *ought* to?