This roundtable was organized as a pre-launch event of the ANR PROGRAMme project. It was organized conjointly with the international symposium Unix in France and in the United States: Innovation, diffusion and appropriation.

Its aim was to offer a first opportunity to give members of the project the opportunity to present some of the major challenges, methods insights and/or issues related to "program" from different perspectives (historical, philosophical, theoretical, formal) and in different (related) contexts (logic, computing hardware, programming languages, software systems). It also aimed at exploring and determining some common points of concern across the different team members.

Some basic questions tackled were:
- What and who determines the meaning(s) of "program"?
- What are the "meanings" of "program"?
- How does a particular understanding of "program" restrict and determine a given social, technical and/or scientific practice, and conversely?
- What is the connection between program, language, algorithm, machine and/or user practice?
- In how far are programming practices determined by (a) machine architectures (b) formalism and logic (c) use
- What does it mean for two programs to be "equal"?
- What is a “simulation”?
- What does it mean for a “program” to be patentable?

The following team members were present at this event, indicating by * those who gave a short talk during the roundtable:

Selmer Bringsjord* (Rensselaer AI & Reasoning Lab), Maarten Bullynck (Université de Paris 8)*, Felice Cardone (Università degli Studi di Torino)*, Edgar Daylight (Siegen University)*, Liesbeth De Mol (CNRS, UMR 8163 Savoirs, Textes, Langage)*, Marie-José Durand-Richard (Sphère, Université Paris-Diderot)*, Jean Lassuège (CNRS, Institut Marcel Mauss, EHESS), Simone Martini (Università di Bologna)*, Elisabetta Mori (Middlesex University), Pierre Mounier-Kuhn (CNRS, Centre Roland Mousnier)*, Elisabetta Mori (Middlesex University)*, Camille Paloque-Berges (CNAM)*, Maël Pégny (UMR 8590 IHPST), Giuseppe Primiero (Middlesex University)*, Franck Varenne (Université de Rouen)*

Besides, there was also quite some active participation from some of the participants of the Unix symposium and the public in general, including: Clem Cole (Intel/Usenix), Samuel Goyet, Thomas Haigh (University of Wisconsin/ Siegen University), Ksenia Tatarchenko and Henri Salha (UMR 8590 IHPST). It is expected that in the future some of these participants might get more involved with the project.

In what follows, the different contributions from the team members are summarized or more or less given as presented, ordered according to the roundtable programme.

Liesbeth De Mol, Introduction

What is a computer program? As all of you know, that is the apparently simple question that brings us all together today at this prelaunch event of the PROGRAMme project. The question is a very basic one not just from the perspective of computing as a "discipline" -- for instance as software engineering or as computer science -- but also from the broader perspective from which computing
is seen as an ubiquitous technique of our society. Indeed, today particular understandings of "program" -- what it is or should be -- might not only affect, for example, our choice of one programming technique over another but also who should be considered responsible if something goes wrong.

So, what, then is a computer program? Is it just some piece of code set-up on a particular computing device - a PDP-11 for instance or an iPhone -- and intended to be used as a means to controlling the behavior of the physics of that device in order to fulfill some computational function such as the sorting of a list or the tracking of the user of the device -- Or, should we rephrase the question, strip it of its temporal and spatial contingencies and turn it into the theoretically more appealing question "What is an algorithm?" Indeed, given the foundational ambitions underpinning a project that has a question as its title, wouldn't it have been more appropriate to refer to a question that is more open to a theoretical and even mathematical analysis? It would certainly fit better with an idea of a computer /science/ which is first and foremost mathematical. Indeed, it was exactly to give computing a more mathematical and/or theoretical foundation, in its struggle for disciplinary independence, that "algorithm" started to take on its role as one of the central notions of the computing discipline in the late 1960s and early 70s. From that more historical perspective, "algorithm" as used within the modern context, historically derived from issues related to "programs" and not conversely and so "algorithm" is anchored in and shaped by "program". In the meantime, this preference of "algorithm" over "program" has become part of the scientific and popular discourse and so, for instance, in the last CNRS Newsletter we do not read "Des programmes contre des images truqués" but "Des algorithmes contre des images truqués" and in a recent article on the Guardian which critically analyzes some of Facebook's practices, it is again an algorithm, not a program, which is supposed to consist of 60m lines of code and is described as a "palimpsest, where engineers add layer upon layer of new commands". Today, so-called "Algorithms" and the abstractions they assume are being sold as the essence of so-called computational thinking while tech firms like Google happily follow that tendency by talking about their "programs" as if they were algorithms. Indeed, one does not talk about the Google search program but the Google search algorithm, PageRank, referring not to the patent holder, Stanford, but the inventor.

Now, while the very idea of abstraction which is implied in the transition from "program" to "algorithm" obviously is a fundamental and successful technique of computer science, the use of "algorithm" /as if/ it is "program" hides the fact that it isn't just about mathematics. First of all, "programs" are also physical processes when written on and ran by a computing device and so have a complex temporality and spatiality that does not simply reduce to the notions of time and space as we find them in computational complexity theory. And it is also because programs (and not algorithms) are physical that one should, for instance, not forget about the geographical location of the cloud server one is using and the privacy laws that might apply there. Moreover, "programs" are also developed, commercialized, used, shared, tested and maintained by people and so are not just physical and mathematical but also socio-technical. By consequence, "programs" (and not algorithms) are also shaped by commercial interests, ideologies, biases, assumptions and, ultimately, carry with them responsibilities.

The choice for "program" (and not "algorithm" or "software" for that matters) as the central "object" of this research project is thus rooted in the acknowledgement that it is something which gathers different modalities in order to bridge the gap between mechanized symbol manipulation and human practices. More particularly, and as is probably already clear, we identify three non-exclusive modalities of "program": as a symbolic text "program" has a mathematical and linguistic modality – a formal modality for short; as something that is stored and executed by a machine, it has a physical modality; and as something that is made and used by people it has a sociotechnical modality.
It is my conviction (and in fact the deeper drive behind this project) that by not acknowledging the significance of each of these three modalities and how they shape and have shaped different practices, that by restricting the meaning of “program”, we not only strengthen disciplinary fences already constructed, but, ultimately, we run the risk of a too reductive notion which, ultimately, becomes a power tool that is far removed from any real meaning.

It is the purpose of this project to offer a coherent analysis and pluralistic understanding of “program” using a combination of historical insights and philosophical as well as theoretical analyses. The general aim is to retrace the different meanings of “program” through its own history and, by so doing, identify, render transparent and systematize the different understandings and practices underpinning “program” from the framework of the three modalities. By so doing, it becomes possible to (re-)construct fundamentals of “programs”, which open-up rather than restrict the meaning of program.

Clearly, this kind of project is very broad and requires an interdisciplinary team with all the risks that brings with it. In fact, I think one of the biggest challenges of this project is exactly that interdisciplinarity -- it will ask a lot from each of the team members to engage with perspectives and methods they are not necessarily used to and might not always agree with. Yet, I think it is the only way to develop a foundation of "programs" that is not too reductive and so tries to gather rather than divide the existing disciplinary approaches within computing and its history and philosophy.

This roundtable is the first gathering of such different perspectives in the framework of this project and it is my sincere hope that it will open up a deep discussion about what "program" is or should be. We will have 11 team members who will each formulate a problem, propose an approach or simply make a stand for a particular viewpoint to be discussed, offering not just a general impression of the diversity of issues of this project, but also the first opportunity to develop a common set of concerns that will have to be faced as the project continues.

Simone Martini, Programs as inscriptions

Programming languages are not only the “tools of the trade” of the computer professional. They are the common language in which computer science is made, and they form an important part of that science's metalanguage. The design of Algol started with the purpose of the “publication of computing procedures in a concise and widely-understood notation”, in such a way that they could be “mechanically translatable into machine programs for a variety of machines” (Backus et al., 1958). The availability of such languages made possible the easy communication of new algorithms, and allowed reasoning on those algorithms (and, more generally, on programs). Bruno Latour, with genial insight, explains in this way the relationship between a new science and its language:

He writes this with reference to the analysis done by F. Dagognet on chemistry, but he could have been talked of computer science as well. Specifically, we may follow (Latour, 1986) in an enumeration of the characteristics of written programs and which specific purposes this language achieves. Programs:

• are “immutable mobiles.” 1 The text of a program may be exchanged on a variety of media, and it does not change its semantics in this process. This should be contrasted with an actual computation (the execution of a program by one of its interpreters), which is much less mobile and requires a lot of infrastructure to be moved and, especially, to be maintained consistent after the move.

• are flat. Written programs are simple two-dimensional Euclidean objects, representing complex time-dependent phenomena, which are left implicit in the text. This absence of the
time from a (standard) program is what makes them handy to conceive, understand, and
manipulate.
- their scale may be changed at will. The availability of programming languages of different
granularity and expressibility, and, moreover, the possibility to automatically translate a
program from one language to another, allow for the description

**Pierre Mounier-Kuhn, The notion of program in early technical texts and the
controversies about the patentability of computer programs**

The presentation briefly addressed two main questions:

1) **What do we learn from the notion of program in early technical texts written by the first
French computer developers (1949-1965), particularly in three locations:**
- The Institut Blaise Pascal, where Louis Couffignal designed an electronic, externally programmed,
calculator in 1947-1951.
- The Grenoble university, which trained programmers from the mid-1950s and soon became a
leading research center in languages, compilers and systems.
- The Société d’Electronique et d’Automatisme (SEA), a startup company created in 1948 to
develop electronic computers. To give only a short timeline of its evolution regarding programs:
  1949: SEA’ founder, Raymond, wrote initial reflections toward the design and « coding » of a von
Neuman machine, following closely the von Neuman reports;
  1952-1955: A young mathematician, Lepage, reflected on what programming should be for this
machine and for further, more advanced, computers; his visions of the machine evolved from a
« calculator » to an « automaton which processes information », from a technical artefact to an
abstract mathematical concept;
  1965 : SEA thoughts evolved toward stack machines inspired by the Algol effort and the Burroughs
B-5000 experience ; then toward non-von Neumann architectures.

2) **What can we learn from the controversies about the patentability of computer programs ?**
Outside of the usual focus of historians and philosophers of science, I suggest to look how lawyers
have considered computer programs, since the 1960s when software started to raise legal issues.
Basically, lawyers have considered computer programs and algorithms in reference two two classes
of objects:
- mathematical theorems,
- elements of machines, elements which are an intrinsic part of the machine to make it work.
Comparing the US and European controversies about the patentability of computer programs should
shed light on « what a program is » in different professional contexts.

**Giuseppe Primiero, Identity and copy of programs.**

In this lecture I will introduce the general lines of research behind some current work on the notions
of identity and copy of programs. Part of this work is done in collaboration with Nicola Angius at
the University of Sassari (Italy). The philosophical problem of identity has a long history, for which
a short summary (limited to our purposes) should at least include the following three crucial
conceptual axes:

1. [Frege, 1892, Quine, 1948, Wright, 1983]: an object is anything that is associated with
   interpretations of terms and variables and it requires
   a definition of identity;
2. [Quine, 1980]: No Entity without Identity
3. [Martin-Löf, 1993]: No entity without type and No type without semantical identity
In computer science, the introduction of the notion of identity is usually referred back to the work of Milner (1971), aiming at making precise a sense in which two programs may be said to be realizations of the same algorithm, namely that the programs do the same ‘important’ computations in the same sequence, even though they differ in other ways. Since then, identity for algorithms can be formally understood as the identification of an equivalence class over all programs written in different languages. The other approach is represented by defining a notion of simulation for labelled transition systems.

One issue in trying to join these two research lines (philosophical and computational), is to determine which of the identity criteria discussed in the philosophical literature [Lowe, 1998, Wiggins, 2001, Quine, 1990] can be said to hold over simulation relations for identity (and their weakenings for the notion of copy). In recent work (Angius, Primiero (submitted)) we have

1. Distinguished several types of identity relations between programs:
   - Perfect Identity
   - Exact Copies
   - Inexact Copies
   - Approximate Copies
2. Associated to each a formal representation in terms of simulation and associated operations
3. Investigated which identity criteria hold for each such relation.

The next steps of this research will follow several different lines:

1. History: metaphysics and proofs-as-programs analysis
2. Complexity: analysis for operations weaker than bi-simulation
3. Epistemology: analysis of second-order properties preserved by copies
4. Ethics: the problem of software copyright

Franck Varenne, *Some ideas to begin with…*

**Research area**: history and epistemology of formal models and computer simulations in contemporary sciences; philosophy of applied computer science

**Some personal results and ideas:**

- There are many different epistemic functions for formal models in contemporary sciences.
- The computerization of science has increased their number.
- A CS is not always dedicated to the calculability problems posed by an untractable formal model. There are at least 3 different types of computer simulations (CS) with different epistemic functions: numerical, rule-based and object-based.
- The programming language, the program and their expressivity are very important for the complete validation of a given CS. They are crucial if one wants to assess the epistemic function of this CS.
- To consider the precise nature, form and running of the program is all the more important if we are to evaluate some complex integrative simulations highly depending on high-level programming as are, for instance, complex simulations based on multiformalisms, multiprocessus and/or multiscale systems of models

**There are specific relations between some of my questions and the «ANR PROGRAM» questions. There are at least two parallel questionings, to begin with:**
I. I think that questioning the **similitude** between a **CS** and an **experiment** (= questioning the epistemic function of a **CS**) sooner or later leads to questioning the mutual **modelling relationships** between the algorithm, the source code and the stored executable program. I recalled and evoked rapidly this questioning about programs in *Qu’est-ce que l’informatique?*, Vrin, 2009, pp. 46-47:

If we look at the already quite old Hoare/Fetzer debate, for instance, we can see that they used arguments that are identical with - or very similar to - those used today by philosophers of simulations when they are debating about the empirical/non-empirical dimension of computer simulations. See on these recent debates e.g.: Margaret Morrison, *Reconstructing Reality – Models, Mathematic, and Simulations*, 2015, OUP, pp. 240-247 and its bibliography. The question of the presence/absence of materiality and of the presence/absence of causal connection is put at the forefront and discussed. In particular many of the arguments in presence are turning around the absence or presence of **causal connection** in the computer simulation as it was the case when the debate was concerning the nature of a computer program, along with this specific question about the presence or absence of causal connection in the high level program (code ?) Vs. the executable program (Fetzer, 2000).

II. I think we also can see similarities between the **problem of program verification** related with the importance of **hierarchies** in high level programming language and in complex programs on the one hand (*Qu’est-ce que l’informatique?*, pp. 49-50), and the problem of the validation of multiprocessus and integrative computer simulations on the other.

I.e.: I also suggest seeing parallel questioning and argumentations between the problem of program **verification** and the problem of complex simulations **validation**.

As noted by Graham White (2004), multiple typifications and hierarchies are problematic when we’re trying to use a **formal semantics** to verify a complex program.

Don’t we face the same difficulties for almost the same reason (multiple typifications and hierarchies) for the problem of anchoring and validating multiprocess and multiscale simulations? On this, see my *Working paper Chains of Reference in Computer Simulations*

**References:**

**Maarten Bullynck, What is a program? A set of questions inspired by history**

This text develops a number of related questions on the notion of ‘program’ taking its cue from some recent research I did on computing in the 1940s, 1950s and 1960s. The questions are developed more or less in chronological order and build up gradually. They are meant both as testimony to historical viewpoints on ‘program’ and as challenges for the current, contemporary debate.
1. A program is a plan
In the early days of digital computing, say the 1940s and 1950s, a distinction was routinely made between ‘program’ and ‘code’. The program was the plan, the solution of a problem, whereas the code was the sequence of instructions that the machine should execute to implement this plan. Goldstine and von Neumann’s Planning and coding of problems has this distinction in its title, and it seems that near the late 1950s it had become common parlour (for a short while). The distinction made its way into the ACM Glossary of 1954:

- Code (verb): to prepare problems in computer code or in pseudocode for a specific computer. Coding: the list, in computer code or in pseudo-code, of the successive computer operations required to solve a given problem.
- Program (verb): to plan a computation or process from the asking of a question to the delivery of results, including the integration of the operation into an existing system. Thus programming consists of planning and coding, including numerical analysis, systems analysis, specification of printing formats and any other functions necessary to the integration of a computer in a system.

or in IBM’s definitions around 1960:
- Code: translating and writing information in an abbreviated or shorthand form, for example, writing machine instructions or symbolic instructions from a statement of the problem. Program: (1) The plan for the solution of problem including data gathering, processing and reporting. (2) A group of related routines which solve a given problem.

As is clear from the definitions, coding is a sequence of instructions prepared for the machine, whereas programming is the overarching category, planning a solution to a problem. Notoriously, von Neumann considered that finding the (mathematical) solution to a problem was the most difficult part, and that planning and especially coding the solution were easy tasks. Programmers after him would strongly disagree, but in one respect he was right, the ‘coding’-part would over time become increasingly automated. Macro-assemblers, interpreting systems, operating systems, compilers and programming languages would be developed (under the general heading ‘automatic programming systems’) that would make the ‘coding’ part easier indeed. The ‘planning’ was (and is) evidently harder to automate, but second-generation languages such as LISP, Simula, APL and others were developed to aid in this process. But, these languages were conceived of as interactive languages of some sort and needed input from the outside (the programmer, the interactive user, a database of heuristics etc.). Indeed, McCarthy became the spokesperson for time-sharing, because he needed it to implement his LISP that could only thrive if it had access to a large memory and interactive debugging facilities.

2. Programs not Program
To ease programming, not only automatic programming was studied, but importantly, libraries of subroutines were created. At first, the catalogue of such routine libraries resembled the content of books of numerical analysis, but soon, new categories of programs altogether would appear such as bootstrap routines, post mortem routines, report routines, error routines, conversion routines, input and output routines, and, also, supervisory or executive routines. Writing a program was as much patching other programs together as it was spelling out the sequence of instructions (see also the second definition of ‘program’ in the IBM manual). Therefore, the question is not so much ‘what is a program’ but ‘what are programs’ and how do they work together. In this process of complexification, the concept of a ‘large’ program became meaningful. The problem of writing large programs was first addressed by people involved with the Whirlwind computer at MIT, but the problem would be recurrent in the years to come, though the phenomenological user-category ‘large’ program would be substituted for ‘system program’, software project etc. (more in tune with the software engineering style). Another evolution of the late 1950s was the ‘operating’ or control program, that tried to automate parts of the operator’s (and programmer’s) job, bundling a number
of conversion, input/output, error and report routines and monitor routines in one system, that, much later, would conceptually crystallize as the ‘operating system’. In many cases, the operating system become the dominant interface for the programmer/user/operator and the computer, automatically handling numerous tasks through an even greater number of programs. Operating systems would also become, e.g. OS/360 and Multics, the classic examples of ‘large’ programs in software engineering.

3. Programs, machine and users
In the early 1960s the developers of Burroughs’s AOSP wrote there had been ”a change of perspective”, viz. ”computers do not run programs, [...] programs control computers.” Indeed, whereas the first digital computer were seen by many to be big calculating machines where each program did a computation, or a data processing machine that automatically handled batches of tasks, this situation would inverse itself. Whether multifunctional computers such as the RW-400, Burroughs D825 or the IBM/360, or time-sharing installations, the programs would now provide the framework for the users/programmers, shaping the polymorph machine into a special-purpose system for the problem at hand. In many cases, the operating system would be the interface through which this happened. This did not only affect the reciprocal relationship between program and machine, but also between the users and the workfloor. The program imposes a division of labour for man and machine, but also between men (as already evident in the Goldstine/von Neumann coding and planning). ¹ This goes to show that programs cannot be analysed unidirectionally, from programmer to machine and back, but only within a network of relationships.

Marie-José Durand Richard, *Diagrams as preliminary to computer programs*

I could contribute to this project from the point of view of an historian of mathematics, working on mathematical machines of the 19th and 20th centuries since several years. Until now, I was not specifically interesting by the issue of what a program is, but I met elements discussed for the organisation of these machines, which are nowadays included in computer programs. So I am interested in updating these elements, so as to understand how these elements have been combined to obtain a result named a program; My general view in history of mathematics is not at all a recurring or a retrospective one. I am not at all looking backwards to see what was similar to our present way of thinking. On the contrary, I try to be very cautious about the context in which mathematical concepts were thought, so as to be able to examine what needed to be deconstructed, and how other ideas have to be associated, so as to constitute something new.

Form these assumptions, it is clear that my contribution does not intend to give an answer to the question: « what a program is? ». In fact, I would like to present three examples, in which the transition from a mathematical computation to its execution by a machine was represented by a diagram which indicated the relations between elementary units corresponding to elementary operations. From these diagrams, it seems possible to raise the following questions:
- how computations on mathematical machines were thought,
- how the writing of procedures evolved,
- what were the continuities and disruptions from one type of machine to the other,
- and how the notion of program progressively merged from these approaches.

As I already told, I have previously examine these mathematical machines, but without a view to examine directly what was related to the notion of program. So my presentation corresponds to a

¹ This is very explicit in numerically controlled systems where the program is dictated by management, implemented by engineers or programmers, but executed by a machinist (see Noble’s book). The classic ‘this will reduce human errors’ is management speak for ‘this will reduce human workforce’, though this would come back with a vengeance, once they need better and more educated workers either as programmers or as operators.
work to begin rather than a work in progress. Nevertheless I can already identify some important steps.

My 3 examples will be the following:

1. The approach, by both Charles Babbage (1791-1871), Luigi Menabrea (1809-96) and Ada Lovelace (1815-52) of the way to describe computations on the Analytical Engine\(^2\), which is considered nowadays as an automatic mechanical calculator with an external program

\[\text{Image of the Analytical Engine}\]

2. The approach of Douglas Hartree (1897-1958) to describe how to set up the differential analyser, of which he realized two items, one in Manchester and one in Cambridge, before World War II

\[\text{Image of the Differential Analyser}\]

3. How the same Hartree, as well as his colleagues in Cambridge, began to write programs for the British EDSAC (Electronical Delay Storage Automatic Calculator), which was built and began to work in Cambridge in 1949.

\[\text{Image of the EDSAC}\]

---

https://www.bibnum.education.fr/calcul-informatique/calcul/notions-sur-la-machine-analytique-de-m-charles-babbage

1. In my view, the first step is important in order to break some false ideas about Lovelace contribution on The Analytical Engine, as she is traditionally presented as “the first programmer”. Instead, I think it is important to notice that Babbage brought drawings and explanations in Italy when he went to Turin in 1840, and that Menabrea, in his review of Babbage’s machine in 1842, already presented some examples of diagrams to describe how the Analytical Engine worked. I name them static diagrams, as they give only the state of the machine when data have been introduced, and the columns with the partial computations. This is Menabrea static diagram for the resolution of a system of two equations of the first degree:

\[
\begin{align*}
mx + ny &= d \\
m'x + n'y &= d'
\end{align*}
\]

What was really new in Lovelace’s work, and which was really HER work, was to introduce what I name dynamical diagrams, where she indicated every step of the computations.
To this end, she carefully distinguished between different uses of the perforated cards for operations:
- supplying cards,
- receiving cards,
- working cards
She introduced a specific notation for the working cards, and she examined very precisely how to minimize the number of cards during the computation.
She also explained precisely the process named « backing », which asked the cards to go back to repeat the execution of a partial part of a computation.

She gave a very complicated example with the computation of Bernoulli numbers, which has only a little error on a sign, but which works on computers nowadays if this error is corrected.

2. Douglas Hartree introduced the differential analyser in England, from his visit to Vannevar Bush (1890-1974) in 1933, and conducted the building of two items of this machine, one in Manchester and one in Cambridge, before the World War II.
The differential analyser was an analog machine.
Nevertheless, Hartree had to draw diagrams to represent the setting of the machine to resolve differential equations. To this end, he started from Bush's notation of the elementary operations of the machine.
Then he drew diagrams to represent how they were connected together.

He isolated some partial settings, named sub-routines, and established a library of these sub-routines to be used in more complicated settings.

3. Thanks to his experience, at the end of the war, Hartree was invited in the United States to establish special settings of the ENIAC. And returning to England, he went on working on the differential analyzer, but was also consulted for the building of the EDSAC. In 1952 he published a book where he collected all the algorithmic methods used to approximate differential equations, and introduced a chapter on programming with diagrams such as this one, for the same equation as the previous one for the differential analyser:
Here are indicated the number of times each elementary operation has to be carried on.

So it will be interesting to distinguish what was similar and what was different between the different ways to represent the processes of computations.

I would like to examine particularly the following book:

with a first edition in 1950, establishing only from programming on the EDSAC, and a new edition in 1957 completed with programming on other machines.

It will be interesting to look at the introduction of the term “program” in the vocabulary of these authors, even if it probably came from the United States.

In these diagrams, it seems to me that the work is going from mathematics to machines, which were conceived, since Babbage, as embodiments of the calculus of operations, without bringing anything from their own. But progressively, the authors began to perceive that the execution of these processes could get some self-governance, and it will be interesting to follow the development of such ideas.

**Felice Cardone, Programs and their meaning(s)**

I would like to make some remarks about programs and their meanings. If you recall that what this project looks for is the meanings of the word "program", you may think that this is a pun, but it is not. Not completely, at least. I believe that the many styles of interpretation that can be given to programs contribute significantly to our understanding of what kind of thing a program is. The idea that programs have interpretations is not completely obvious, because it defines programs as semiotic objects.

And indeed in the beginning there was semiotics, in the backround of the early reflections on program semantics. The hero of this phase was Saul Gorn, who drew from the semiotics of Peirce and Morris through the filter of formal logic. Recall that Carnap had the first chapter of his Introduction to semantics (1941) devoted to semiotics and its three parts, along the lines of Morris: syntax, semantics and pragmatics. Other pioneers were Heinz Zemanek and Alfonso Caracciolo di Forino: they all thought of programming languages as languages. We can see a trace of their ideas in the discussions that took place, in particular, at the 1964 IFIP working conference on formal
language description languages. (I understand that Troy Astarte has a paper devoted to an analysis of this conference.) It is in this early phase of the reflection on programming languages that logic enters the picture and starts contributing the foundations of what we now regard as formal semantics. One important name in this connection is that of Curry, with his investigations of formal systems that were central in his analysis of logic, and the related techniques. Structural induction is probably the best known of these. Curry and Gorn were acquainted with each other, and it is possible to find in the works of Gorn traces of Curry's work on formal systems. Curry was also deeply concerned with formal languages from a semiotic viewpoint. My contention is that an investigation of this period is interesting, not only in itself, but also in relation to the formation of a few "dogmas" of program semantics. I do not mean that these dogmas are altogether wrong. But giving meaning to programs assumes implicitly what a program is, and I would like to just outline a few points where the assumptions of formal semantics seem to give programs a restricted ontology which hinders the pluralistic view of programs that underlies this project. The first point is compositionality. This is one great achievement of logic inherited by denotational semantics in the Scott-Strachey tradition. There are several ways of reading this assumption, and Wilfrid Hodges has several papers on this. In general it states that the meaning of a program phrase is a function of the meanings of its components. More generally this allows for the decomposition of a system into smaller parts relying on the possibility of composing local behaviors to obtain a unique global system behavior. This is also related to constituent structure of a programming language, with recursive rules allowing to generate abstract syntax trees over which recursively synthesize meanings (or behaviors), as McCarthy first did in his early papers on computing. A general dogma underlying these points is the implicit assumption of the possibility of passing from local to global in a uniform way. Furthermore, in a vague sense that should be made precise, the program control is localized in a pointlike entity which neglects the physical distribution of the program text across the page, its 2-dimensionality. The self that executes the program is unextended, like a monad. Brian Rotman wrote several papers on this a few years ago, for example Going parallel or Becoming beside oneself, where he points out how computer and technology in general tend to make the traditional Self more plural, or parallel, leading to a lesser role for writing than visualization. At the same time, from the technical side, it is by now widely accepted that parallelism (in the widest sense) is best represented by higher-dimensional formalisms (higher categories, higher order automata and so on).

But where are the program and its control in, say, a configuration of Conway's Game of Life? Or in a global snapshot protocol, whose aim is precisely to reconstruct a global state of a distributed system out of the local states of its components and those of the communication channels? The fact that the notion of global system state is operationally meaningless in a distributed context can be seen to follow from general principles about locality of action in computing (or physics, for that matter), yet global states are assumed tacitly in most models of programs. Compositionality is also closely related to extensionality, which "means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value" (Strachey, Fundamental concepts in programming languages, 1971): this may be read as stating that the value of an expression is a function of the values of its subexpressions --- another way of expressing compositionality. It has been extremely important to see the values of program phrases as set-theoretical entities like functions and sets, but throughout the history of semantics more intensional notions have always been in the background, even in the eminently extensional context of denotational semantics. The concrete data structures of Gerard (Gustave?) Berry and Pierre-Louis Curien are a prominent example together with the game models arising from work on Girard's linear logic by Abramsly, Hyland and their coworkers. Another common dogma of semantics is that meaning is independent of representation, which is suggested by the denotational approaches to program semantics. If we turn to algorithms as the meanings of programs, we see instead that representations are quite relevant. This is because algorithms rely upon representation in their formulation: representations are data structures, and often these latter are reflected in the
control structures that use them --- think of the various notions of recursion, like recursion on notations, in arithmetic. In the foundations of mathematics, ultrafinitists are very uneasy about this dogma. In computer science, complexity theorists are uneasy too. (However, I recall Robert Tarjan leaving open the possibility of a denotational approach to complexity: "One possibility is to develop a denotational definition of complexity, as has been done for program semantics [...], but since this is a current research topic we shall proceed in the usual way and define complexity operationally"). In conclusion, I suggest that an exegesis of these dogmas underlying the semantics of programs may contribute to a better understanding of what programs themselves are.

Selmer Bringsjord, Programs and logic programming
I endeavored to place on the proverbial table a very simple fact: viz.,

An answer to the question (Q) "What is a (computer) program?" that can be sensibly provided by a devotee of "pure" logic programming might be regarded by some as rather striking.

For instance, imagine a program that consists of all and only the single formula (pos (and \zeta (not \zeta))), which is an S-expression (with LaTeX for pretty printing) for an obvious formula in propositional modal logic. Now let us imagine that at the prompt we can ask the following query: (and (Tall alvin) (not (Tall alvin))). This is an S-expression for a quantifier-free first-order formula that is clear to all. Assuming that the inference schemata in the background for both propositional modal logic and first-order logic are standard, we will receive back the answer yes. For most people asking Q, it will seem odd that we have here a program, and that upon execution we receive back this result. But we certainly do have here a program, and the execution need only follow trivial deduction (as search over the inference schemata) to yield the output of yes.

In my presentation, I ended with a simple puzzle from the game META-FORMS. The point of that example was two-fold. On the one hand, the example shows that answers to Q are typically given so firmly solely from the perspectives of functional and procedural programming that most people find it hard to grasp that solving the puzzle in question is to hand-simulate a pure logic program (with suitable background inference schemata). Secondly, the example shows that there is available a paradigm for programming even at the early ages based on pure logic programming, in which to program is to list formulae that represent propositions about some environment, and to execute is to enable the machine to, relative to background inference schemata, search for a theorem entailed by these formulae.

Edgar Daylight, Strachey's halting problem
The term 'Halting Problem' arguably refers to computer science's most celebrated impossibility result. Software engineers often ignore this theoretical outcome however. In retrospect, this is not very surprising given that several researchers follow Christopher Strachey's 1965 alleged proof of his Halting Problem (which is about executable programs) rather than Martin Davis's correct 1958 version or his 1994 account (each of which is solely about mathematical objects). For the sake of conceptual clarity, I will scrutinize Strachey's 1965 line of reasoning which is widespread today both from a charitable, historical angle and from a critical, engineering perspective.

Elisabetta Mori, Validity & Correctness in LEO I and II
What is a (correct) computer program? The presentation is an historical reconstruction of the procedures to ensure validity and program correctness in the early examples of business computers: the focus of the talk is the hardware testing, data validation and program correctness techniques designed for LEO I and II in the UK during the 1950s.
As opposite to mathematical and scientific work, which typically requires a small number of highly complex calculations to be performed, in business computing a large number of simple calculations had to be accomplished in the shortest possible time.

Hardware tests would follow the identification of a fault, but more in general were performed daily, prior to operational work, to check if every physical component of the computer was well-functioning and to check the reliability of the results achieved, in order to minimise the risk of faults. Maintenance tests were of two kinds: preventive and curative. They included thermionic valves testing and marginal testing, also in conjunction with program tests for hardware.

Data Validation procedures were divided into general input checks, manual and automatic checksums and I/O test programs.

In LEO, a systematic approach to fault diagnosis was set up at early stages, together with the development of complex equipment and laborious procedures to process high amount of data. Following a well-known distinction already in place for the EDSAC, programs to locate errors were of two kinds: post-mortem routines and checking routines. These practices were aided by the useful procedure of photographing the cathode ray tube image of the store. The process of ensuring program correctness consisted in designing the logic of programs in stages, also through the flowcharting technique. Complete programs or parts of them were then checked on the computer with trial data, prepared to test every possible requirement. A non systematised program correctness check had been in place since the early stages of LEO. The computer was equipped with a loudspeaker: each application had its own noise rhythm and experienced operators would note something had gone wrong.

Validity and correctness preoccupations originate at the very beginning of computing. Methodology and practices developed at LEO to approach validity and correctness show a modern attitude towards these notions, though many of their protocols were rooted in much older accounting and auditing practices. The duality between correctness and reliability, between validity and efficiency, was also the result of the essential impossibility of delegating at least partial control of such issues to an operating system. This would change significantly with the advent of LEO III and its Master Routine.