Every file is a program, but not the reverse

Baptiste Mélès

Every file is a program, but not the reverse

English translation of the article entitled "Tout fichier est un programme… et non l'inverse" and published in *La Gazette des Mathématiciens* **176** (April 2023)

Baptiste Mélès

The concept of a program, unlike that of a file, is an informal one, giving rise to inaccurate or even false assumptions. In this article, we will contest two of them. First, using historical examples, we will show that there are programs that are not files. Then, more surprisingly, we will argue that any file can be seen as a program, since all criteria characterizing programs hold true for files. In particular, we will conclude that there is no technical reason to distinguish between file/program and program/interpreter pairs.

1 Introduction: technical vs. informal concepts

Since scientific discourse does not rely exclusively on technical concepts, it leaves certain questions to philosophy.[1]

By technical concepts, we mean those that have a precise definition enabling their exact usage within the internal framework of science, i.e., a definite attribution to certain objects, their according usage in justification procedures – demonstration, observation, experimentation – and finally in scientific statements; this is the case for the concepts of integer, derivative, electron, mammal, zero-sum game, etc. The definitions of technical concepts are always precise, backed up by methods for verifying whether they can be applied to objects.

The same cannot be said of certain more informal concepts which, at the frontier between science and common experience, are suited to approximate usage, such as the concepts of number, calculation, living, etc. In the absence of an exact definition, these are generally too approximate or too slippery for scientific use. Whenever an attempt is made to provide these informal concepts with a rigorous definition, it will always be open to discussion.

It is true that scientists sometimes propose a rigorous definition of a hitherto informal concept; but whether the proposed definition correctly and completely translates common intuition remains an open question, belonging to philosophy rather than science. So, when Gentzen calls his formal system of logical deduction a "natural deduction," when Turing claims that his machines can compute everything and not only what is ordinarily called "computable," both are developing technical concepts based on the prior analysis of informal concepts [3, 4, 12]: a philosophical approach, not a scientific one. Scientific theories of proof and computability can then be developed on this basis, but the philosophical debate will continue on the starting point, namely the definition of what it means to deduce naturally and what it means to compute.

Indeed, the study of informal scientific concepts in their relation to technical concepts calls for properly philosophical methods such as conceptual analysis and dialectical discussion, if by the latter we mean the contradictory examination aimed at gaining knowledge of things beyond their first appearance. Thus, for example, in a famous article, James H. Moor showed that three conceptual oppositions commonly found in computer discourse – software and hardware, digital and analogue, model and theory – did not in fact correspond so much to distinct entities but rather to different ways of looking at the same thing [9]. The informal concepts used in scientific discourse call for philosophical methods.

It is precisely at this frontier between sciences and philosophy that we will study the relationship between two central concepts of computer science: file and program.

2 What you may have heard about programs and files

Let us first define the terms "file" and "program," before summarizing what one commonly assumes one knows about them.

The term "computer file" is a technical term, and as such, is susceptible to precise definition. A file is a sequence of binary digits (bits) denoted in a computer system by an identifier – typically a name or a number.[2] One distinguishes between plain text files

[1] Of course, this does not mean that philosophy cannot *also* have its say on technical concepts.

[2] The Multics designers define a file as follows: "A file is simply an ordered sequence of elements, where an element could be a machine word, a character, or a bit, depending upon the implementation" [1]. Andrew Tanenbaum suggests the following definition: "A Unix file is a sequence of 0 or more bytes containing arbitrary information" [11]. However, a more technical definition of "file" is simply that it contains source code, according to the Unix source code `inode` structure [6, lines 5659–5675].

(conventionally given the extension `.txt`, …), word-processing documents (`.odt`, `.doc`, …), spreadsheet documents (`.odc`, `.xls`, …), audio files (`.ogg`, `.mp3`, …) and video files (`.mpg`, …), as well as directories and some "special files," which under Unix family systems denote printer, screen, scanner, processor temperature and so on [10, p. 367] and [2].[3] This precise definition makes it technically easy to determine whether a given object is a file.

The concept of a program, however basic it may seem in computer science, is an informal one. We can, of course, propose formal interpretations of it, typically dependent on an arbitrarily chosen computational model: a program will sometimes be defined as a term of the λ-calculus, sometimes as the set of transitions of a Turing machine starting from an initial state, and so on. But these concepts do not characterize what a program is: they simply attest that λ-terms or Turing machines can be seen as programs, and that any program could in absolute terms be seen as the analogue of a λ-term or a Turing machine… *albeit with some translation*. But when one calls a C-language source code a "program," one is not thinking about λ-terms, and one is convinced that we are in the immediate presence of a program.

So what does one usually think of as a program? We consider that a common-sense definition of a program is *a layout of symbols that is supposed to determine the behaviour of a machine*?[4]

Like any real definition, this definition requires a few explanations. First, we will say that a program is a "layout of symbols" in general, since it can be a chart, a diagram or a punched card, as well as a text made up of letters and other typographical symbols. The "machine" itself can then be situated in a particular environment made up of human actors as well as physical objects and other machines. Finally, the program is only "aimed" at determining the expected behaviour because, as the etymology indicates, the program is *written before* it is executed, and it can also be faulty yet still be a program. We will not use the notion of programming language, as it would be perfectly tautological to define a program as something written in a program-writing language.[5] Our current aim would be achieved if our definition were not deemed too contrary to common sense.

Now let us look at what one generally assumes to be obvious about programs. They would be texts written in a "programming" language; they would intrinsically consist of a sequence of instructions, which enables them to be executed; being composed of "instructions," they would be intrinsically active objects. In all this, they would be opposed to *data*, which are inert objects, containing no instructions and only playing a role in an active process when passed as an argument to a program; they would therefore be intrinsically passive. A special case of data are the *files*, which can be passed as input to a program, but which in most cases are not programs, given that they do not contain instructions enabling them to be executed by a system.

The following two theses will be widely shared:

1. Judging by the programs that populate our hard disks, *all programs are files*.
2. A few trivial examples, such as plain text files and audio files, demonstrate that the reverse is not true; in other words, *not all files are programs*.

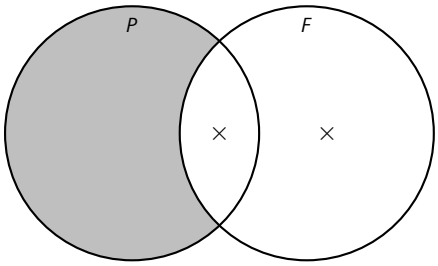

Figure 1. The relation between file and program according to common sense.

We will show that these two common-sense assertions do not stand up to technical scrutiny, which will lead us to the following two theses:

1. Not all programs are files.
2. All files are programs.

3 Not all programs are files

In today's computer practice and culture, we might be tempted to support the first common-sense thesis, according to which *every program is a file*. In support of this thesis, we can invoke the following *intuitive argument*: all we have to do is look in our hard

[3] For a philosophical analysis of what is a file in UNIX, see [8].

[4] A similar intuition is, e.g., expressed in Clarisse Herrenschmidt, *Les Trois écritures : langue, nombre, code*, Paris, France, Gallimard, 2007, p. 404: "Qu'est-ce qu'un programme ? Un programme est un texte où sont consignées les instructions données à un ordinateur, un texte écrit dans un langage." (What is a program? A program is a text in which the instructions given to a computer are recorded, a text written in a language. [Our translation])

[5] That is what the ISO does by defining a program as a "syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem"… before defining "programming language" as an "artificial language for expressing *programs*?" [our emphasis]. It is also worth noting that this definition omits the concept of the machine, which seems essential to us (International Organization for Standardization, 2015: Information technology – Vocabulary. (ISO/?IEC Standard 2382:2015(en)), https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v2:en, accessed 06/15/2022).

disks, and we'll see that all the programs we can find there are files, whether binary files or scripts, i.e., text-mode program files intended for mechanical interpretation.[6]

However, this argument is not sufficient to prove this thesis, for two reasons.

1. The argument is based on an improper generalization. It is based on experience, which always allows us to form existential theses, but never – except in the finite case – universal theses. Since one cannot conclude from the existential affirmative to the universal affirmative, one can derive from experience the statement "some programs are files," but not "all programs are files."
2. The argument is circular. The experiment in question implicitly assumes the context of programs found in our modern computers, where programs happen to be stored in a memory structured by… a file system.[7] If we make this context explicit in the above formula, we obtain the following tautology: "Every program *file* is a *file*."

Let us go a step further and refute not only the intuitive argument, but the first common-sense thesis itself: history attests to the existence of programs that were not files. File systems did not appear until the 1960s, with the CTSS and Multics operating systems [1], even though programs already existed: a punched card from a Jacquard loom, a barrel organ or, more recently, a batch-processing machine is a program, but not a file – not even in retrospect, since it is not part of a file system. The programs that Turing supplies to his universal machine in the form of a "standard description" – e.g., the program

```
;DADDCRDAA;DAADDRDAAA;DAAADDCCRDAAAA;DAAAADDRDA;
```

which alternates between 0 and 1 – are not files either. So, against the first common-sense thesis, we can assert that *not all programs are files*.

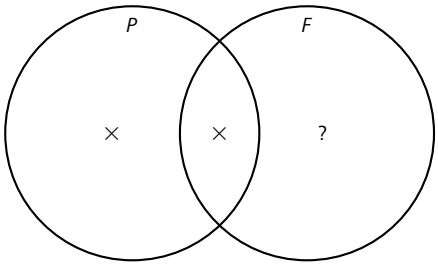

Figure 2. Not all programs are files.

4 All files are programs

Let us turn now to the second common-sense thesis, according to which *not all files are programs*, or in other words, *some files are not programs*. Like the previous one, this thesis can be based on an intuitive argument: videos, text files, etc. are indisputably files, but they are not programs.[8] This time, the empirical argument is safe from any improper generalization, since a single example suffices to demonstrate an existential thesis.

However, against this intuitive argument, we are going to show that every file is a program. We will do this by analysing successive file types, from the most obvious to the most general case: first, we will study the trivial case of *program files*, in order to reveal by which criteria we judge a file to be a program; then we will take a closer look at *data files* to show that these criteria apply just as well to them. In other words, if the former are programs, then the latter should be too.[9]

4.1 Why are program files programs?

Why wasting time demonstrating such a trivial thesis as *program files are programs*? Because what we are interested in here is not simply revealing *that* this thesis is true, but rather *what makes* this thesis true: we will see later that these criteria, once admitted, actually apply far beyond these obvious cases.

[6] Generally speaking, it is difficult to provide references for common-sense theses: theses that are taken for granted have virtually no author. One example among a thousand, no more representative than any other, is a text signed under the pseudonym StackLima, entitled "Différence entre programme et fichier" (Difference between program and file [our translation]) and dated 07/05/2022 (URL: https://stacklima.com/difference-entre-programme-et-fichier/; accessed 10/28/2022), stating the following definition of programs [our translation, our emphasis]: "Programs, as their name implies, are simple executable *files* containing a set or collection of instructions used by the computer to execute or perform particular tasks and to produce desired results." It must be stressed that we are not dealing here with a scientific text, which explains and even excuses its many inaccuracies.

[7] A file system (e.g. FAT32, NTFS, ext4, etc.) is a device for structuring computer memory into files, i.e., sequences of characters. This device specifies the memory segmentation mode, data structures for referencing files, an overall topology (typically a tree structure), file manipulation functions (creation, deletion, modification, dating, etc.) and various other functions (logging, rights management, encryption, network adaptation, etc.).

[8] The text on the difference between program and file quoted above distinguishes in two columns the supposedly distinct characteristics of programs and files. On one of the lines, we read the following two sentences, implicitly presented as an opposition [our translation]: "Program types include application programs, system programs such as word processors, operating systems, database systems, etc. File types can be JPEG, PNG, GIF, PDF, MP4, etc." With this reference, we simply intend to illustrate an informal opposition between programs and files that seems to us to be fairly widespread.

[9] We are not denying a difference in nature by arguing a difference in degree: we could just as easily have gone straight to the last case, which is the most general; however, for pedagogical purposes, we have opted for a more progressive approach.

We will treat three types of program files, from the simplest to the less obvious: (a) binary files, (b) files to be compiled, (c) files to be interpreted.

Because they are machine-executable

The most obvious type of program is the binary program file.

A binary program file is a sequence of instructions in machine language, i.e., directly interpretable and executable by the machine without any intermediary. It is made up of sequences of 0s and 1s, and is generally not displayable in text form: if you try to display it, you will generally get a mindless succession of strange symbols – graphic symbols, emoticons, punctuation marks, characters of all alphabets and even signs that cannot be viewed because they have no equivalent in ASCII or Unicode coding.

Clearly, any binary program file trivially satisfies what we consider as the common-sense definition: "a layout of symbols aimed at determining the behaviour of a machine." The machine executes this program directly, and it is written in the very language of its processor.

We have got a first conclusion to draw from this: Anything that can be directly executed by the machine is a program.

Because they are translatable into executables

However, not all programs can be directly executed by the machine. What makes them programs nonetheless?

Let us look at the case of program files to be compiled. A binary program file is often produced from a text file written in a so-called "programming language" such as C or Java. Here is a short classic example of a program file to be compiled, written in C:

```
#include <stdio.h>

int main()
{
    printf("Hello world.\n");
    return 0;
}
```

Can we say that program files to be compiled are programs in the same sense as binary program files? Common intuition does not put this in doubt: it is not uncommon to say "my program" when referring to what is in fact only the *source code* of a binary program. The only difference with binary program files is that the program files to be compiled do not directly modify the behaviour of the machine: they do it only after a translation stage, called compilation.

This leads us to the following thesis: a file can be a program, *even if it is not directly executable and its execution presupposes a prior extrinsic step*.

This concession is important, as it shows that a program is not always an *intrinsically* active object. A compilable program is merely a text file which, in order to become active, must be supplied to another program. The above-mentioned common conception of programs and data, as opposed to one another, proves thus to be wrong: some data turn out to be programs if there is a suitable translator. In at least some cases, it is a condition extrinsic to the program file itself – the existence of the compiler – that turns the file into a program. *The opposition between programs and data is therefore not intrinsic*.

It may be objected that, *once compiled*, the file is transformed into a binary program, making it "directly executable" by the machine. A compilable file is, in this sense, "indirectly directly executable."

Because they can be interpreted as executables

However, program files are not restricted to binaries and compilables: one also commonly groups interpretable files in this category.

An interpretable program file is a program that is intended to be executed as it is read by a specific program. Shell scripts are interpreted by a shell, batch files by MS-DOS, Perl, Python etc. scripts by the respective interpreters of these languages, and so on. Here is an elementary example of a shell program:

```
#!/bin/zsh

for nephew in Huey Dewey Louie
do
    echo "Hello $nephew!"
done
```

Can we say that interpretable program files are programs in the same way as binary or compilable program files are? Unlike binary or compilable program files, interpreter program files do not directly modify the behaviour of the machine, even after an intermediate step: they modify the behaviour of an intermediate program that is currently running, namely the interpreter. But since the interpreter is itself running, any change in its behaviour modifies the behaviour of the machine. The program files to be interpreted, which common intuition calls programs, therefore always respect our definition of a program as "a layout of symbols aimed at determining the behaviour of a machine."

The following thesis can be drawn from this: a file can be a program *even if it requires the execution of an intermediate program to modify the behaviour of the machine*.

The files in this class qualify as programs only in an entirely indirect way, since at no point are they transformed into directly executable files. Their program status is – and remains – purely extrinsic, since it depends on the existence and execution of the specific program, namely the interpreter.

Conclusion: the high price of a trivial thesis

Analysis of these three trivial cases – binary, compilable and interpretable programs – leads us to the unsurprising conclusion

that *program files are programs*. But this assertion is not without concessions, since we have to admit that a program may not be directly executable by the machine – as compilables prove – and may even never be transformed into a machine-executable file – as interpretables prove. There is a certain price to pay for adhering to this seemingly trivial thesis.

We will now show that this apparently obvious thesis opens a Pandora's box: Any file can be seen as a program.

4.2 Why not consider data files as programs, too?

While it is easy to admit that program files are programs, in the case of data files this thesis seems not only less obvious, but downright false. Yet we are going to demonstrate that everything that classifies program files as programs also applies to data files.

Again, we will proceed in three stages, from the simplest to the least obvious, successively examining (a) document files to be compiled, (b) structured data files and (c) any files.

We will first show that not every compilation produces an executable; then that any structured data can also be seen as an instruction; finally, that any data can be seen as structured, in other words as an instruction.

Not every compilation produces an executable

A document file to be compiled is a file written in a language that allows it to be automatically translated into a file of a given format, which is generally not intuitively recognized as a program. This is the case, for example, with a TEX – not LATEX – file as the following one, from which one can obtain a viewable or printable file in DVI, PS or PDF format:

```
Hello, {\it world}!
\bye
```

The resulting PDF file is generally not considered a program. But what about the TEX file it comes from?

A document to be compiled like this one contains mainly text – in this case "Hello" and "world" – but also instructions, such as the italics command `\it`, which will be graphically translated into italics in the compiled document, or the `\bye` command, which indicates to finalize the document. So it is immediately obvious that compilable documents contain at least some instructions – just like program files.

Perhaps someone will argue that these files "mix" instructions with simple data? That would be a mistake. The instructions indicated in TEX by a backslash are only one small part of the iceberg: the documents to be compiled are *entirely* made up of instructions. As Donald Knuth pointed out in the TEX language manual, even the smallest letter is a command [5, p. 267]. After explaining that any occurrence of the letter *b* in a TEX document can be replaced by the command `\char98`, Knuth reveals that this is actually TEX's underlying mechanism: it interprets the letter *b* as an alias for this command. TEX simply reads a document as a sequence of instructions. The words "Hello" and "world" are likewise sequences of instructions. TEX is in fact a particularly powerful language, since it is Turing-complete: Anything that can be written in any programming language can be written in TEX – which does not mean it would be a good idea to do so.

So, can we say that compilable documents are programs? As a sequence of instructions, they determine the machine, via compilation, to produce a certain file. This leads us to admit that a document to be compiled is a program *even if it is not written in a so-called "programming" language, and its output is not generally considered to be a program*.

All structured data is an instruction

Let us now consider the case of interpretable document files. We refer to structured documents, i.e., files written in a document presentation language such as HTML, roff, Markdown or SPIP. Here is a very basic example of an HTML file:

```
<html>
    <head>
        <title>HTML page</title>
    </head>

    <body>
        <p>This is an HTML page.</p>
    </body>
</html>
```

A structured document contains not only text, but also instructions such as `<body>` and `<strong>`. These commands are interpreted by the browser, just as LATEX compiles the `\it` instructions and the shell interprets the `echo` command. Even "purely textual" parts such as the words `This is an HTML page` are instructions for the program responsible for displaying it. So if *interpreted programs* (e.g., shell scripts) and *compiled documents* (e.g., TEX files) are programs, then *interpreted documents* are programs in two respects. An HTML file is a program for browsers, a PS file is a program for printers, and even PDF, video, etc. files are programs for viewers of various kinds.

This thesis is worth emphasizing, as it is probably here that common sense will put up the most resistance. According to ordinary intuition, there are data, programs and interpreters; the first are generally not the second, and the second are generally not the third. Yet what we see from the case of structured files – which by definition always obey a syntax no less rigorous than that of program files – is that *a structured file supplied to a program does not technically differ from a program supplied to an interpreter*.

We can thus conclude that not only the texts written in a so-called programming language are programs, but also any text

written in a given syntax, even if it is a "document structuring" rather than a "programming" language.

All data is an instruction

Let us turn now to plain text documents, i.e., files that encode a simple sequence of characters, as for example the following text file named `aeneides.txt`:

> Arma virumque cano, Troiae qui primus ab oris
> Italiam, fato profugus, Laviniaque venit
> litora, multum ille et terris iactatus et alto
> vi superum saevae memorem Iunonis ob iram;
> multa quoque et bello passus, dum conderet urbem,
> inferretque deos Latio, genus unde Latinum,
> Albanique patres, atque altae moenia Romae.

Any plain-text document opened in any program, even a simple text editor, qualifies as a program for this interpreter. It is structured data – even if it is at the zero level of structure, reduced, for example, to its encoding – and as such it is an instruction in its own right. For example, the file `aeneides.txt` represents the following sequence of instructions for a text editor, or for any other program that is supposed to manipulate it (display with the `cat` command, search with `grep`, etc.): display "`A`," display "`r`," display "`m`"… just as an HTML file without italics is a program interpreted by the browser. Any plain text file, *provided it can be opened by any program*, is consequently an interpreted program in its own right.

It is now easy to move on to the general case of any file: Any sequence of bits, whether readable in text mode, behaves like a program as long as it is supplied as input to a given program, which in turn behaves like an interpreter. Every file therefore respects the intuitive definition of a program as "a layout of symbols aimed at determining the behaviour of a machine." All that is needed is at least one user mechanism – whether a machine or a running program – which adapts its behaviour to the contents of the file.

Before being opened, a file is only potentially a program, but this is also the case for interpretable files. To take such reasoning to the limit, only a file that is not intended to be opened at all might not be a program. We dare to claim – albeit without proof – that there is no such case.

5 Conclusion: An extrinsic characterization of programs

Contrary to the common intuition that all programs are files, but some files are not programs, we have argued that some programs are not files, but that all files can indifferently be considered as programs.

Indeed, we have shown that a program is not necessarily directly executable by the machine, that it does not even always enable the production of a directly machine-executable file, and that it sometimes depends on the underlying execution of another program. The result is that any data file supplied as input to a program can be seen as a program supplied to an interpreter, without there being any technical way of distinguishing the two situations.

A program is therefore not defined intrinsically: What characterizes it as such is a context of execution and use, which is extrinsic to it.[10]

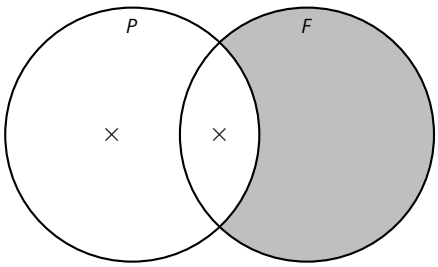

Figure 3. The connection between file and program as defended here.

6 Discussion

Perhaps these conclusions will seem to be counter-intuitive. It will then probably be necessary to develop a more restrictive definition of the concept of program than the one we used.

One option might be to require that a program be written in a "programming language," a concept that will have to be defined in its turn – obviously without circularly appealing to the concept of program – so as not to reopen Pandora's box.

(a) Will we define a programming language as a complete language in the Turing sense, i.e., capable of expressing any algorithm computable in a Turing machine? We will have to justify the heterodox exclusion of Coq, a deliberately incomplete language that admits only the strict subset of the class of terminating functions.

(b) Should we call any language that tolerates recursion a programming language? In that case, we would have to accept the inclusion of all TEX files, since the TEX language offers all the control structures common to programming languages. Conversely, we would have to exclude the Catala language, developed by Denis Merigoux to formalize the French Tax Code, which does not have a general recursion structure [7, p. 11].

[10] A corollary of this conclusion is that the same file can be used for several programs at the same time. An empty file, i.e., of length 0, for example, is both a program for a text editor and for interpreters such as Shell, Perl, Python, etc. A C source code, for its part, causes different machine behaviour depending on whether it is submitted to a compiler – which will produce a binary program file – or to a text editor – which will produce a text display.

Pending a better definition, we will have to accept that there is no technical criterion for distinguishing the file-program pair from the program-interpreter pair, and that their distinction depends solely on the context of use and the purposes of the agent who manipulates them.[11]

Acknowledgements. The *Magazine of the EMS* thanks *La Gazette des Mathématiciens* for authorisation to republish this text, which is an English translation of the paper entitled "Tout fichier est un programme… et non l'inverse" and published in *La Gazette des Mathématiciens* **176** (April 2023). The author thanks J.-B. Bru and M. Gellrich Pedra for the English translation.

References

[1] R. C. Daley and P. G. Neumann, A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, pp. 213–229, ACM, New York, USA (1965)

[2] O. Daudel, */proc et /sys*. O'Reilly, Paris, France (2006)

[3] G. Gentzen, Untersuchungen über das logische Schließen. I. *Math. Z.* **39**, 176–210 (1935)

[4] G. Gentzen, Untersuchungen über das logische Schließen. II. *Math. Z.* **39**, 405–431 (1935)

[5] D. Knuth, *The TEXbook*. Addison–Wesley, Reading, MA, USA (1984, 1996)

[6] J. Lions, *Lions' commentary on Unix, 6th edition with source code*. Annabook, Poway, CA, USA (1996)

[7] D. Merigoux, N. Chataing and J. Protzenko, Catala: a programming language for the law. In *Proc. ACM Program. Lang.*, vol. 5, article no. 77 (2021)

[8] B. Mélès, Unix selon l'ordre des raisons : la philosophie de la pratique informatique. *Philos. Sci.* **17**, 181–198 (2013)

[9] J. H. Moor, Three myths of computer science. *Brit. J. Phil. Sci* **29**, 213–222 (1978)

[10] D. M. Ritchie and K. Thompson, The UNIX time-sharing system. *Comm. ACM* **17**, 365–375 (1974)

[11] A. S. Tanenbaum, *Modern operating systems*. (2nd. ed.), Prentice Hall, Upper Saddle River, NJ, USA (2001)

[12] A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. (2)* **42**, 230–265 (1936)

[11] This text was written in the context of the ANR program *What is a program? Historical and philosophical perspectives* (PROGRAMME, ANR JCJC, ANR-17-CE38-0003-01, 2018–2022). It was greatly enriched by discussions with the project leader, Liesbeth De Mol, and Henri Stéphanou, as well as by comments from the rapporteurs of the *Gazette des Mathématiciens*.

Baptiste Mélès is a researcher at the CNRS in the Archives Henri-Poincaré, Université de Lorraine, University of Strasbourg.

baptiste.meles@univ-lorraine.fr