Tout fichier est un programme... et non l'inverse

Baptiste Mélès

Tout fichier est un programme… et non l'inverse

• B. Mélès

La notion de programme étant, contrairement à celle de fichier, informelle, des croyances imprécises ou fausses peuvent circuler à son égard. Nous en contestons deux ici. Nous montrons d'abord, à partir d'exemples historiques, qu'il existe des programmes qui ne sont pas des fichiers. Nous soutenons ensuite, plus étonnamment, que n'importe quel fichier peut être vu comme un programme, puisque tous les critères qui caractérisent les programmes en tant que tels s'appliquent aussi bien aux fichiers. Nous en concluons plus particulièrement qu'il n'y a pas de raison technique de distinguer les couples fichier/programme et programme/interprète.

1. Introduction : concepts techniques et concepts informels

Le discours scientifique ne s'appuyant pas exclusivement sur des concepts techniques, il laisse certaines questions à la philosophie [1].

Par concepts techniques, nous entendrons ceux qui possèdent une définition précise permettant leur mobilisation exacte dans le cadre interne de la science, c'est-à-dire une attribution certaine à certains objets puis un usage dans des procédures de justification – démonstration, observation, expérimentation – et enfin dans des énoncés scientifiques ; telles sont les notions de nombre entier, de dérivée, d'électron, de mammifère, de jeu à somme nulle etc. La définition des concepts techniques est toujours précise en ce qu'elle s'accompagne de méthodes permettant de vérifier leur application ou non à des objets.

Il en va tout autrement de certains autres concepts, plus informels, qui, à la frontière de la science et de l'expérience commune, s'accommodent parfaitement d'usages approximatifs ; telles sont les notions de nombre, de calcul, de vivant, etc. Faute de définition exacte, ceux-ci sont généralement trop approximatifs ou trop glissants pour un usage scientifique précis. Quand l'on prétend munir ces concepts informels d'une définition rigoureuse, celle-ci reste ouverte à la discussion.

Il arrive certes que des scientifiques proposent une définition rigoureuse d'un concept jusqu'ici informel ; mais savoir si la définition proposée traduit correctement et complètement l'intuition commune reste une question ouverte, qui ne relève pas de la science mais de la philosophie. Ainsi, lorsque Gentzen appelle « déduction naturelle » son système formel de déduction logique, lorsque Turing prétend que ses machines permettent de calculer tout et seulement ce que l'on appelle ordinairement « calculable », tous deux élaborent des concepts techniques en s'appuyant sur l'analyse préalable de concepts informels [2] : une analyse non scientifique, mais philosophique. Sur cette base peuvent ensuite se développer des théories scientifiques de la démonstration et de la calculabilité ; on n'en pourra pas moins continuer à débattre philosophiquement sur le point de départ, à savoir la définition de ce que c'est que déduire naturellement et de ce que c'est que calculer.

L'étude des concepts scientifiques informels dans leur lien avec les concepts techniques appelle

1. Ceci n'exclut naturellement pas que la philosophie ait *aussi* son mot à dire sur les concepts techniques.
2. Gerhard Gentzen, « Untersuchungen über das logische Schließen. I », *Mathematische Zeitschrift*, vol. 39, n° 2, 1935, p. 176-210 ; Gerhard Gentzen, « Untersuchungen über das logische Schließen. II », *Mathematische Zeitschrift*, vol. 39, n° 3, 1935, p. 405-431. Alan Mathison Turing, « On computable numbers, with an application to the Entscheidungsproblem », *Proceedings of the London Mathematical Society*, vol. 42, n° 2, 1936, p. 230-265.

ainsi des méthodes proprement philosophiques telles que l'analyse conceptuelle et la discussion dialectique, si l'on entend par ce dernier terme l'examen contradictoire visant à connaître des choses par-delà leur première apparence. C'est ainsi, par exemple, que, dans un article célèbre, James H. Moor avait montré que trois oppositions conceptuelles courantes dans le discours informatique – logiciel et matériel, numérique et analogique, modèle et théorie – ne correspondaient en réalité pas tant à des entités distinctes qu'à des façons différentes de considérer les mêmes choses [3]. Les concepts informels du discours scientifique appellent des méthodes philosophiques.

C'est à cette frontière de la science et de la philosophie que nous étudierons la relation entre deux notions informatiques : celles de fichier et de programme.

2. Ce que l'on croit savoir sur les programmes et les fichiers

Définissons d'abord les notions de fichier et de programme avant de résumer ce que l'on croit ordinairement savoir à leur sujet.

La notion de fichier informatique est une notion technique, susceptible d'une définition précise. Un fichier est une suite de chiffres binaires (bits) dénotée dans un système informatique par un identifiant – typiquement un nom ou un numéro [4]. On parlera ainsi de fichiers en texte brut (portant souvent par convention l'extension `.txt`...), de documents de traitement de texte (`.odt`, `.doc`...), de document tableur (`.odc`, `.xls`...), de fichiers audio (`.ogg`, `.mp3`...) et vidéo (`.mpg`...), mais aussi de répertoires et même de « fichiers spéciaux » qui sous les systèmes de la famille Unix dénotent l'imprimante, l'écran, le scanner, la température du processeur, etc. [5] Cette définition précise rend techniquement facile de déterminer si un objet donné est un fichier ou non.

Le concept de programme, quelque fondamental qu'il puisse paraître en informatique, est quant à lui une notion informelle. On pourra évidemment en proposer des interprétations formelles, typiquement dépendantes d'un modèle de calcul arbitrairement choisi : un programme sera tantôt défini comme un terme du λ-calcul, tantôt comme l'ensemble des transitions d'une machine de Turing à partir d'un état initial, etc. Mais ces notions ne caractérisent pas la notion de programme : elles attestent simplement que l'on peut voir les λ-termes ou les machines de Turing comme des programmes, et que tout programme pourrait dans l'absolu être vu comme l'analogue d'un λ-terme ou d'une machine de Turing... *à traduction près*. Mais lorsque l'on appelle « programme » un code source en langage C, on ne pense pas aux λ-termes et l'on est convaincu d'être en présence immédiate d'un programme.

Qu'entend-on alors ordinairement par programme ? Nous pensons nous tenir relativement près du sens commun en définissant le programme comme *disposition de signes supposée déterminer le comportement d'une machine* [6].

Comme toute définition réelle, la définition mérite quelques explications. Nous disons d'abord que le programme est une « disposition de signes » en général, car il peut s'agir d'un schéma, d'un diagramme ou d'une carte perforée aussi bien que d'un texte composé de lettres et d'autres signes typographiques. La « machine » peut ensuite elle-même être située dans un environnement particulier constitué d'acteurs humains ainsi que d'objets physiques et d'autres machines. Le programme n'est enfin que

3. James H. Moor, « Three Myths of Computer Science », *The British Journal for the Philosophy of Science*, Vol. 29, N° 3 (Sep., 1978), p. 213-222.

4. Les concepteurs de Multics définissent le fichier de la façon suivante : « 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 » (Daley R. C. et 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*, New York, New York, USA, ACM, 1965, p. 213-229, en ligne : `http://doi.acm.org/10.1145/1463891.1463915` ; consulté le 19 mai 2018). Andrew Tanenbaum propose la définition suivante : « Un fichier Unix est une suite éventuellement vide d'octets contenant une information quelconque » (Tanenbaum Andrew, *Systèmes d'exploitation*, Paris, France, Pearson Education, 2003, p. 773). Mais la définition on ne peut plus technique de la notion de fichier est tout simplement celle que contient le code source, dans la définition de la structure `inode` du code source d'Unix (Lions John, *Lion's commentary on Unix 6th edition with source code : Unix operating system source code level six and commentary on the Unix operating system*, Poway (Californie), Annabook, 1996, lignes 5659-5675).

5. Dennis M. Ritchie et Ken Thompson, « The UNIX time-sharing system », *Communications of the ACM, vol. 17, n° 7, 1er juillet 1974, p. 365-375, ici p. 367. Voir également Daudel Olivier,* /proc et /sys, *Paris, France, O'Reilly, 2006. Pour une analyse philosophique de la notion de fichier dans UNIX, cf. Baptiste Mélès, « Unix selon l'ordre des raisons : la philosophie de la pratique informatique », Philosophia Scientiae, n°17-3, octobre 2013, p. 181-198*

6. Une intuition similaire est par exemple exprimée dans 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. »

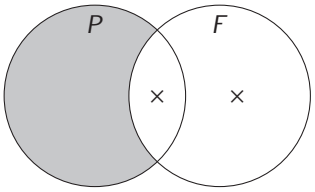
« supposé » déterminer le comportement attendu car, comme l'indique l'étymologie, le programme est *écrit avant* son exécution et il peut également être bogué sans pour autant cesser d'être un programme. Nous n'utilisons pas la notion de langage de programmation car il serait parfaitement tautologique de définir le programme comme ce qui est écrit dans un langage d'écriture de programmes [7]. Notre objectif présent serait atteint si la définition n'était pas jugée trop contraire au sens commun.

Voici maintenant ce que l'on croit aller de soi sur les programmes. Ils seraient des textes écrits dans un langage « de programmation » ; ils consisteraient de façon intrinsèque en une suite d'instructions, ce qui permet leur exécution ; en tant que composés d'« instructions », ils seraient des objets intrinsèquement actifs. En tout ceci, ils s'opposeraient aux *données*, qui sont des objets inertes, qui ne contiennent pas d'instructions et qui ne jouent de rôle dans un processus actif qu'une fois passés en argument à un programme ; elles seraient donc intrinsèquement passives. Un cas particulier de données est celui des *fichiers*, qui peuvent être passés comme entrée à un programme, mais qui dans la plupart des cas, ne sont pas des programmes, s'ils ne contiennent pas des instructions permettant leur exécution par un système. Deux thèses seront donc assez largement partagées :

1. si l'on en juge par les programmes qui peuplent nos disques durs, *tous les programmes sont des fichiers* ;

2. quelques exemples aussi triviaux que les fichiers en texte brut ou les fichiers audio attestent que la réciproque est fausse ; en d'autres termes, *tous les fichiers ne sont pas des programmes*.

Figure 1 – La relation entre fichier et programme selon le sens commun



Nous montrerons que ces deux thèses de sens commun ne résistent pas à l'examen technique, ce qui nous mènera aux deux thèses suivantes :

1. tous les programmes ne sont pas des fichiers ;
2. tous les fichiers sont des programmes.

3. Tous les programmes ne sont pas des fichiers

Dans notre culture et notre pratique informatiques actuelles, l'on pourrait être tenté de soutenir la première thèse du sens commun, selon laquelle *tout programme est un fichier*. À l'appui de cette thèse, on peut invoquer l'*argument intuitif* suivant : il suffit de regarder dans nos disques durs et nous verrons que tous les programmes que nous pourrons y trouver sont des fichiers, qu'il s'agisse de fichiers binaires ou de scripts, c'est-à-dire de fichiers de programme en mode texte destinés à une interprétation mécanique [8].

Cet argument ne suffit pourtant pas à démontrer cette thèse, et ce pour deux raisons.

1. L'argument repose sur une généralisation abusive. Il s'appuie en effet sur l'expérience, qui permet toujours de forger des thèses existentielles mais jamais – sauf dans le cas fini – des thèses universelles. Comme on ne peut

7. C'est ce que fait l'ISO, qui définit le programme comme une « 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 »… avant de définir le « langage de programmation » comme un « artificial language for expressing *programs* » (nous soulignons). Ajoutons que cette définition omet la notion de machine, qui nous semble essentielle (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:v1:en`, consulté le 15 juin 2022.

8. Il est, de manière générale, difficile d'attester par une référence des thèses de sens commun : les thèses réputées aller de soi n'ont pour ainsi dire pas d'auteur. Un exemple entre mille, nullement plus représentatif qu'un autre, est, dans un texte signé sous le pseudonyme StackLima, intitulé « Différence entre programme et fichier » et daté du 5 juillet 2022 (en ligne : `https://stacklima.com/difference-entre-programme-et-fichier/` ; consulté le 28 octobre 2022), la définition suivante du programme : « Les programmes, comme leur nom l'indique, sont de simples *fichiers* exécutables contenant un ensemble ou une collection d'instructions utilisées par l'ordinateur pour exécuter ou effectuer des tâches particulières ainsi que pour produire les résultats souhaités » (nous soulignons). Précisons bien qu'il ne s'agit pas d'un texte de recherche, ce qui explique et même pardonne ses nombreuses imprécisions.

conclure de l'existentiel affirmatif à l'universel affirmatif, nous pouvons tirer de l'expérience l'énoncé « certains programmes sont des fichiers », mais non « tous les programmes sont des fichiers ».

2. L'argument est circulaire. L'expérience en question suppose implicitement le contexte des programmes que l'on trouve dans nos ordinateurs modernes, où les programmes se trouvent être enregistrés dans une mémoire structurée par un système… de fichiers [9]. Rendons explicite ce contexte dans notre formule et l'on obtiendra la tautologie suivante : « Tout *fichier* de programme est un *fichier* ».

Allons plus loin en réfutant non seulement l'argument intuitif, mais la première thèse du sens commun elle-même : l'histoire atteste l'existence de programmes qui n'ont pas été des fichiers. Les systèmes de fichiers ne sont en effet apparus que dans les années 1960 avec les systèmes d'exploitation CTSS et Multics [10], alors qu'il existait déjà des programmes : une carte perforée de métier Jacquard, d'orgue de Barbarie ou plus récemment de machine à traitement par lots est un programme mais n'est pas un fichier – pas même rétrospectivement, puisqu'elle ne s'inscrit pas dans un système de fichiers. Les programmes que Turing fournit à sa machine universelle sous forme de « description standard » – par exemple le programme

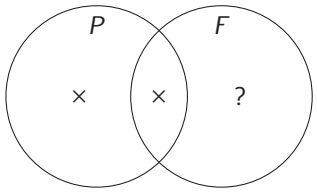
```
;DADDCRDAA;DAADDRDAAA;DAAADDCCRDAAAA;DAAAADDRDA;
```

qui affiche une alternance de 0 et de 1 – ne sont pas des fichiers non plus. Nous pouvons donc, contre la première thèse du sens commun, affirmer que *tous les programmes ne sont pas des fichiers*.

4. Tous les fichiers sont des programmes

Venons-en maintenant à la seconde thèse du sens commun, selon laquelle *tous les fichiers ne sont pas des programmes*, autrement dit *certains fichiers ne sont pas des programmes*. Comme la précédente, cette thèse peut s'appuyer sur un argument intuitif : les vidéos, fichiers texte, etc. sont indiscutablement des fichiers mais ne sont pas des programmes [11]. Cette fois, l'argument empirique est à l'abri de toute généralisation abusive, puisqu'il suffit d'un seul exemple pour démontrer une thèse existentielle.

FIGURE 2 – Tous les programmes ne sont pas des fichiers



Nous allons pourtant montrer, contre cet argument intuitif, que tout fichier est un programme. Nous procéderons à cette fin par analyse de types de fichiers successifs, du cas le plus évident au plus général : nous étudierons d'abord le cas trivial des *fichiers de programme* afin de mettre au jour selon quels critères nous jugeons qu'un fichier est un programme ; puis nous étudierons les *fichiers de données* pour montrer que ces critères s'appliquent tout aussi bien à eux. Autrement dit, si les premiers sont des programmes, les seconds devraient l'être aussi [12].

9. Un système de fichiers (par exemple FAT32, NTFS, ext4, etc.) est un dispositif de structuration d'une mémoire informatique en fichiers, c'est-à-dire en séquences de signes. Ce dispositif spécifie le mode de segmentation de la mémoire, les structures de données permettant le référencement des fichiers, une topologie d'ensemble (typiquement une arborescence), des fonctions de manipulation de fichiers (création, suppression, modification, datation…) ainsi que diverses autres fonctionnalités (journalisation, gestion des droits, chiffrement, adaptation aux réseaux, etc.).

10. Daley R. C. et 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*, New York, New York, USA, ACM, 1965, p. 213-229, en ligne : `http://doi.acm.org/10.1145/1463891.1463915` ; consulté le 19 mai 2018.

11. Le texte citée plus haut, « Différence entre programme et fichier », distingue sur deux colonnes les caractéristiques supposées distinctes des programmes et des fichiers. Sur l'une des lignes, on lit les deux phrases suivantes, implicitement présentées comme une opposition : « Les types de programmes comprennent le programme d'application, le programme système tel que les traitements de texte, le système d'exploitation, le système de base de données, etc. / Les types de fichiers informatiques peuvent être JPEG, PNG, GIF, PDF, MP4, etc. ». Par cette référence, nous entendons simplement illustrer une opposition informelle entre programmes et fichiers qui nous semble être assez répandue.

12. Il ne s'agit donc pas de nier une différence de nature en arguant d'une différence de degré : nous aurions tout aussi bien pu passer directement au dernier cas, qui est le plus général, mais nous avons choisi à titre pédagogique une démarche plus progressive.

4.1 – Pourquoi les fichiers de programme sont-ils des programmes ?

Pourquoi perdre son temps à démontrer une thèse aussi triviale que *les fichiers de programme sont des programmes* ? Parce que ce qui nous intéressera ici ne sera pas simplement de mettre au jour *que* cette thèse est vraie, mais bien plutôt *ce qui fait que* cette thèse est vraie : nous verrons plus loin que ces critères, une fois admis, s'appliquent en réalité bien au-delà de ces cas évidents.

Nous examinerons successivement trois types de fichiers de programme, du plus simple au moins évident : a) binaires, b) à compiler, c) à interpréter.

Parce qu'ils sont exécutables par la machine

Le cas le plus évident de programme est le fichier de programme binaire.

Un fichier de programme binaire est une suite d'instructions en langage machine, c'est-à-dire directement interprétable et exécutable par la machine sans intermédiaire d'aucune sorte. Il est composé de suites de 0 et de 1 et n'est généralement pas affichable sous forme de texte : si l'on essaie de l'afficher, on obtiendra généralement une succession sans queue ni tête de symboles étranges – symboles graphiques, émoticônes, signes de ponctuation, caractères de tous alphabets et même signes non visualisables puisque sans équivalent dans les codages ASCII ou Unicode.

De toute évidence, tout fichier de programme binaire satisfait trivialement ce qui nous semble être la définition du sens commun : « disposition de signes supposée déterminer le comportement d'une machine ». La machine exécute en effet directement ce programme, qui est écrit dans le langage même de son processeur.

On en tire une première conclusion : tout ce qui est exécutable directement par la machine est un programme.

Parce qu'ils sont traductibles en exécutables

Tout programme n'est pourtant pas exécutable directement par la machine. En quoi s'agit-il néanmoins de programmes ?

Analysons le cas des fichiers de programme à compiler. Un fichier de programme binaire est souvent produit à partir d'un fichier texte écrit dans un langage dit « langage de programmation » tel que C ou Java. Pour reprendre un exemple classique, voici un court exemple classique de fichier de programme à compiler écrit en C :

```
#include <stdio.h>

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

Les fichiers de programme à compiler sont-ils des programmes au même titre que les fichiers de programme binaires ? L'intuition commune ne le met pas en doute : il n'est pas rare que l'on dise « mon programme » en désignant ce qui n'est en réalité que le *code source* d'un programme binaire. Le code source satisfait en effet la définition du sens commun : il s'agit bien d'une « disposition de signes supposée déterminer le comportement d'une machine », la seule différence avec les fichiers de programmes binaires étant que les fichiers de programme à compiler ne modifient pas directement le comportement de la machine, mais seulement après une étape de traduction, appelée compilation.

On en retient la thèse suivante : un fichier peut être un programme *même s'il n'est pas directement exécutable et que son exécution suppose une étape extrinsèque préalable*.

Cette concession est d'importance, car elle montre qu'un programme n'est pas toujours un objet *intrinsèquement* actif. Un programme compilable n'est qu'un fichier de texte qui pour devenir actif doit être fourni à un autre programme. La conception commune exposée plus haut a donc tort d'opposer de façon intrinsèque programmes et données : certaines données ne s'avèrent des programmes que pourvu qu'il existe un traducteur idoine. Au moins dans certains cas, c'est ainsi un fait extrinsèque au fichier de programme lui-même – l'existence du compilateur – qui fait de lui un programme. *L'opposition entre programmes et données n'est donc pas intrinsèque*.

On objectera peut-être qu'*après compilation*, le fichier est transformé en un programme binaire et qu'il devient alors *directement* exécutable par la machine. Le fichier compilable est donc « indirectement directement exécutable ».

Parce qu'ils sont interprétables comme des exécutables

Les fichiers de programmes ne se résument pourtant pas aux binaires et aux compilables : on y range tout aussi communément les fichiers interprétables.

Un fichier de programme interprétable est un programme destiné à être exécuté au fur et à mesure de sa lecture par un programme spécifique. Les scripts shell sont interprétés par un shell, les fichiers batch par le MS-DOS, les scripts Perl, Python etc. par les interprètes respectifs de ces langages, etc. Voici un exemple élémentaire de programme shell :

```
#!/bin/zsh

for neveu in Riri Fifi Loulou
do
    echo "Bonjour $neveu !"
done
```

Les fichiers de programme à interpréter sont-ils des programmes au même titre que les fichiers de programme binaires ou à compiler ? Contrairement aux précédents, ils ne modifient pas directement le comportement de la machine, fût-ce après une étape intermédiaire : ils modifient le comportement d'un programme intermédiaire en cours d'exécution, à savoir l'interprète. Mais celui-ci étant lui-même en cours d'exécution, toute modification de son comportement modifie le comportement de la machine. Les fichiers de programme à interpréter, que l'intuition commune appelle programmes, respectent donc toujours notre définition du programme comme « disposition de signes supposée déterminer le comportement d'une machine ».

On en retient la thèse suivante : un fichier peut être un programme *même si pour modifier le comportement de la machine il nécessite l'exécution en cours d'un programme intermédiaire*.

Les fichiers de cette classe ne sont des programmes que de façon totalement indirecte, puisqu'à aucun moment ils ne sont transformés en fichiers directement exécutables. Leur statut de programme leur est – et reste – purement extrinsèque, puisqu'il dépend de l'existence et de l'exécution du programme spécifique qu'est l'interprète.

Conclusion : le coût élevé d'une thèse triviale

L'analyse de ces trois cas triviaux – programmes binaires, compilables et interprétables – nous mène à la conclusion sans surprise selon laquelle *les fichiers de programme sont des programmes*. Mais cette affirmation ne va pas sans concession, puisque l'on doit admettre qu'un programme peut ne pas être directement exécutable par la machine – comme l'attestent les compilables – et qu'il peut même ne jamais être transformé en un fichier exécutable par la machine – comme l'attestent les interprétables. Admettre cette thèse apparemment triviale ne va donc pas sans un certain coût.

Nous allons montrer par la suite que cette thèse d'une apparente banalité confondante ouvre la boîte de Pandore : tout fichier peut être vu comme un programme.

4.2 – Pourquoi les fichiers de données ne seraient-ils pas des programmes ?

S'il est aisé d'admettre que les fichiers de programmes sont des programmes, cette thèse semble dans le cas des fichiers de données non seulement moins évidente, mais tout simplement fausse. Nous allons pourtant montrer que tout ce qui fait que les fichiers de programme sont des programmes s'applique aussi bien aux fichiers de données.

Nous procéderons encore en trois temps, du plus simple au moins évident, en examinant successivement a) les fichiers de document à compiler, b) les fichiers de données structurées et c) les fichiers quelconques.

Nous montrerons d'abord que toute compilation ne produit pas un exécutable ; puis que toute donnée structurée peut aussi être vue comme une instruction ; enfin, que toute donnée peut être vue comme structurée, autrement dit comme une instruction.

Toute compilation ne produit pas un exécutable

Un fichier de document à compiler est un fichier écrit dans un langage permettant sa traduction automatique en un fichier d'un format donné, qui n'est généralement pas reconnu intuitivement comme un programme. Tel est le cas, par exemple, d'un fichier TEX – non LATEX – comme celui-ci, à partir duquel on

pourra obtenir un fichier visualisable ou imprimable aux formats DVI, PS ou PDF :

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

Le fichier PDF qui en résulte ne sera généralement pas considéré comme un programme. Qu'en est-il du fichier TEX dont il provient ?

Un document à compiler comme celui-ci contient essentiellement du texte, ici « Hello » et « world », mais également des instructions, comme la commande d'italiques `\it`, qui se traduira graphiquement par des italiques dans le document compilé, ou la commande `\bye`, qui indique de finaliser le document. On voit donc immédiatement que les documents à compiler contiennent au moins en partie des instructions – au même titre que les fichiers de programmes.

Peut-être alors dira-t-on que ces fichiers « mélangent » des instructions et de simples données ? Ce sera se méprendre. Les instructions signalées en TEX par une barre inversée ne sont que la partie émergée de l'iceberg : les documents à compiler ne contiennent *que* des instructions. Comme le soulignait Donald Knuth dans le manuel du langage TEX, la moindre lettre est déjà une commande [13]. Ayant en effet expliqué que l'on peut remplacer dans un document TEX toute occurrence de la lettre *b* par la commande `\char98`, Knuth dévoile qu'il s'agit là en réalité du mécanisme interne de TEX : il interprète la lettre *b* comme un alias de cette commande. TEX ne lit donc un document que comme une suite d'instructions. Les mots « Hello » et « world » sont donc eux aussi des suites d'instructions. TEX est d'ailleurs un langage particulièrement puissant puisqu'il est complet au sens de Turing : tout ce qui peut être écrit dans un langage de programmation quelconque peut être écrit en TEX – ce qui ne veut pas dire que ce serait une bonne idée.

Les documents à compiler sont-ils donc des programmes ? En tant que suite d'instructions, ils déterminent la machine, via la compilation, à produire un certain fichier. Ce qui nous conduit à admettre qu'un document à compiler est un programme *même s'il n'est pas écrit dans un langage réputé « de programmation » et que son résultat n'est généralement pas considéré comme un programme*.

Toute donnée structurée est une instruction

Venons-en au cas des fichiers de document à interpréter. Nous entendons par là les documents structurés, c'est-à-dire les fichiers écrits dans un langage de présentation de documents tel que HTML, roff, Markdown ou SPIP. Voici un exemple minimal de fichier HTML :

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

  <body>
    <p>Ceci est une page HTML.</p>
  </body>
</html>
```

Un document structuré contient non seulement du texte, mais également des instructions telles que `<body>` et `<strong>`. Ces commandes sont interprétées par le navigateur, de la même façon que LATEX compile les instructions `\it` et que le shell interprète la commande `echo`. Même les parties « purement textuelles » comme les mots `Ceci est une page` HTML sont des instructions pour le programme chargé de l'affichage. De ce fait, si les *programmes interprétés* (par exemple les scripts shell) et les *documents compilés* (par exemple les fichiers TEX) sont des programmes, alors les *documents interprétés* sont des programmes à double titre. Un fichier HTML est un programme pour navigateurs, un fichier PS est un programme pour imprimantes, et même des fichiers PDF, vidéo etc. sont des programmes pour visionneuses de diverses sortes.

Cette thèse mérite d'être soulignée, car c'est probablement ici que le sens commun opposera le plus de résistance. Selon l'intuition ordinaire, il existe des données, des programmes et des interprètes, les premières ne sont généralement pas les deuxièmes et les deuxièmes ne sont généralement les troisièmes. Ce que nous montre au contraire le cas des fichiers structurés – qui par définition obéissent toujours à une syntaxe nullement moins rigoureuse que celle des fichiers de programmes – est qu'*un fichier structuré fourni à un programme ne se distingue pas techniquement d'un programme fourni à un interprète*.

Nous pouvons donc conclure que non seulement les textes écrits dans un langage dit de programmation sont des programmes, mais que cela est vrai de tout texte écrit dans une syntaxe donnée,

13. Donald Knuth, *The TeXbook*, Addison-Wesley, 1984, 1996, p. 43.

fût-elle celle d'un langage « de structuration de documents » plutôt que « de programmation ».

Toute donnée est une instruction

Abordons maintenant le cas des documents en texte brut, c'est-à-dire des fichiers codant une simple suite de caractères, comme le ferait par exemple un fichier `aeneides.txt` :

> Arma virumque canō, Trōiae quī prīmus ab ōrīs
> Ītaliam, fātō profugus, Lāvīniaque vēnit
> lītora, multum ille et terrīs iactātus et altō
> vī superum saevae memorem lūnōnis ob īram ;
> multa quoque et bellō passus, dum conderet urbem,
> inferretque deōs Latiō, genus unde Latīnum,
> Albānīque patrēs, atque altae moenia Rōmae.

Tout document en texte brut, aussitôt ouvert dans un programme quelconque, fût-ce un simple éditeur de texte, est un programme pour cet interprète. Ce sont des données structurées – même s'il s'agit du degré zéro de la structure, réduit par exemple à son encodage – et à ce titre des instructions à part entière. Par exemple, le fichier `aeneides.txt` est pour un éditeur de texte, ou pour tout autre programme supposé le manipuler (affichage avec la commande `cat`, recherche avec `grep`, etc.), la suite d'instructions suivante : affiche « `A` », affiche « `r` », affiche « `m` »… exactement comme un fichier HTML sans italiques est un programme interprété par le navigateur. Tout fichier en texte brut, *pour peu qu'il soit ouvrable par un programme quelconque*, est donc un programme interprété, donc un programme à part entière.

Il est maintenant facile de passer au cas général des fichiers quelconques : n'importe quelle suite de bits, qu'elle soit lisible en mode texte ou non, se comporte comme un programme pour peu qu'elle soit fournie comme entrée à un programme donné, qui pour elle se comportera comme un interprète. Tout fichier respecte donc la définition intuitive du programme comme « disposition de signes supposée déterminer le comportement d'une machine ». Il suffit qu'il existe au moins un mécanisme d'utilisation – qu'il s'agisse d'une machine ou d'un programme en cours d'exécution adaptant son comportement au contenu de ce fichier. Avant ouverture, un fichier n'est certes un programme qu'en puissance ; mais c'est aussi le cas des interprétables. Pour raisonner à la limite, seul un fichier qui ne serait destiné à être ouvert d'aucune façon pourrait ne pas être un programme. Nous oserons poser – toutefois sans démonstration – qu'il n'en existe pas.

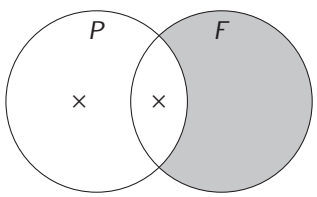
5. Conclusion : caractérisation intrinsèque des programmes

Contrairement à l'intuition commune, selon laquelle tous les programmes sont des fichiers mais certains fichiers ne sont pas des programmes, nous avons soutenu que certains programmes ne sont pas des fichiers mais que tous les fichiers peuvent indifféremment être vus comme des programmes.

Nous avons en effet montré qu'un programme n'était pas nécessairement exécutable directement par la machine, qu'il ne permettait même pas toujours de produire un fichier exécutable directement par la machine et qu'il dépendait parfois de l'exécution sous-jacente d'un autre programme. Il en résulte que tout fichier de données fourni en entrée à un programme peut être vu comme un programme fourni à un interprète, sans qu'il existe de moyen technique de distinguer les deux situations.

Le programme ne se définit donc pas de façon intrinsèque : ce qui le caractérise comme tel est un contexte d'exécution et d'utilisation, qui lui est extrinsèque [14].

Figure 3 – La relation entre fichier et programme défendue ici



14. Cette conclusion a pour corollaire qu'un même fichier peut être plusieurs programmes à la fois. Un fichier vide, c'est-à-dire de longueur 0, est par exemple à la fois un programme pour éditeur de texte et pour des interprètes shell, Perl, Python etc. Un code source en C provoque de son côté des comportements différents de la machine selon qu'il est soumis à un compilateur – qui produira un fichier de programme binaire – ou à un éditeur de texte – qui produira un affichage de texte.

6. Discussion

Peut-être ces conclusions paraîtront-elles contre-intuitives. Il faudra alors probablement élaborer une définition de la notion de programme plus restrictive que la nôtre.

Une option pourrait être d'exiger qu'un programme soit écrit dans un « langage de programmation », notion qui devra être définie à son tour – évidemment sans en appeler de façon circulaire à la notion de programme – pour ne pas rouvrir la boîte de Pandore.

a) Définira-t-on un langage de programmation comme un langage complet au sens de Turing, c'est-à-dire capable d'exprimer tout algorithme calculable dans une machine de Turing ? Il faudra justifier l'exclusion hétérodoxe de Coq, langage délibérément incomplet puisqu'il n'admet que le sous-ensemble strict de la classe des fonctions qui terminent.

b) Appellera-t-on plutôt langage de programmation tout langage tolérant la récursivité ? Il faudra alors assumer l'inclusion de tous les fichiers TEX, le langage TEX offrant toutes les structures de contrôle usuelles des langages de programmation. Inversement, il faudra exclure le langage Catala, élaboré par Denis Merigoux pour formaliser le Code des impôts français, qui ne possède pas de structure de récursion générale [15].

Dans l'attente d'une meilleure définition, il faudra se résoudre à admettre qu'il n'existe pas de critère technique pour distinguer le couple fichier-programme du couple programme-interprète et que leur distinction dépend uniquement du contexte d'utilisation et des finalités de l'agent qui les manipule [16].

Baptiste Mélès

CNRS, université de Lorraine, archives Henri-Poincaré

baptiste.meles@normalesup.org

http://baptiste.meles.free.fr/

Baptiste Mélès est chargé de recherche. Son travail porte entre autres sur la philosophie mathématique et sur l'étude linguistique des codes sources.

15. Denis Merigoux, Nicolas Chataing et Jonathan Protzenko. « Catala : a programming language for the law ». *Proceedings of the ACM on Programming Languages*, vol. 5 (18 août 2021), article 77, p. 1-29, ici p. 11. Que le code juridique français n'ait pas besoin de récursion infinie est indéniablement une bonne nouvelle.

16. Ce texte a été rédigé dans le contexte du programme ANR « Qu'est-ce qu'un programme ? Perspectives historiques et philosophiques » (PROGRAMME, ANR JCJC, ANR-17-CE38-0003-01, 2018-2022). Il doit beaucoup aux discussions avec la porteuse du projet, Liesbeth De Mol, et Henri Stéphanou, ainsi qu'aux remarques des rapportrices ou rapporteurs de la *Gazette de la Société Mathématique de France*.