

---

# Approche philologique des langages de programmation

**Baptiste Mèlès**

LHSP – Archives Henri-Poincaré  
91 avenue de la Libération  
54000 Nancy  
France  
[baptiste.meles@univ-lorraine.fr](mailto:baptiste.meles@univ-lorraine.fr)

---

**RÉSUMÉ.** On appelle souvent « langages de programmation » aussi bien des langages concrets que des modèles abstraits de calcul. Pourtant, les premiers possèdent bien des propriétés que les seconds s'efforcent d'éviter, et qui les rapprochent des langues naturelles : des irrégularités syntaxiques, des traces historiques, des symboles explétifs ou redondants, un apprentissage par la pratique. On peut dès lors appliquer aux langages de programmation des méthodes issues de la linguistique générale et de la philologie comme les analyses comparatives, synchroniques et diachroniques, étymologiques, phylogénétiques, stylistiques et littéraires.

**ABSTRACT.** By “programming languages” are meant both concrete languages and abstract computation models. However, the formers have many properties which the latters try to avoid, and which are close to those of natural languages: syntactic irregularities, historical residues, expletive or redundant symbols, learning through practice. Hence, one could apply to programming languages methods coming from general linguistics and philology such as comparative, synchronic and diachronic, etymological, phylogenetic, stylistic and literary analyses.

**MOTS-CLÉS :** langages de programmation, linguistique, philologie.

**KEYWORDS:** programming languages, linguistics, philology.

---

DOI:10.3166/TSI.35.237-254 © 2016 Lavoisier

## 1. Introduction

### 1.1. Langages théoriques et langages concrets

Sous son unité apparente, la notion de langage de programmation semble recouvrir deux réalités différentes : d'une part des langages « concrets » comme l'assembleur, le C, Perl, SQL, Lisp, Prolog et Javascript, de l'autre des modèles abstraits de calcul, ou langages « théoriques », comme les tables de machines de Turing, le  $\lambda$ -calcul et les catégories cartésiennes closes.

Technique et science informatiques – n° 2/2016, 237-254

Les premiers sont mis en œuvre sur des ordinateurs au moyen de compilateurs ou d'interprètes tandis que les seconds sont définis dans des articles et pourraient sans dommage rester sur le papier. Les premiers permettent aux développeurs d'écrire des programmes dotés d'une certaine utilité pratique — systèmes d'exploitation, interprètes de commande, navigateurs, jeux — tandis que les seconds servent essentiellement à faciliter la démonstration de théorèmes de logique ou d'informatique théorique. Quoique confondues sous une même appellation, les deux sortes de langages semblent ainsi ne pas moins différer par le support que par les fins. Que les auteurs d'un livre d'initiation à Haskell aient pu choisir pour titre *Real World Haskell* montre à quel point la distinction entre langages théoriques et langages concrets est profondément ancrée dans les esprits et les pratiques de ceux mêmes qui cherchent à y remédier : certains acteurs pensent en termes « de monoïdes, de foncteurs et d'hylémorphismes » tandis que d'autres ne parlent que de programmes « exécutés sur un parc de matériel doté de ressources limitées »<sup>1</sup>.

Une façon d'expliquer cet écart entre langages théoriques et langages concrets serait de présenter les seconds comme de simples « applications » des premiers ; la pratique serait un corollaire de la théorie. Naturellement, on peut parler d'application sans imposer un ordre chronologique entre le modèle abstrait et le langage concret : Lisp fut créé après le  $\lambda$ -calcul mais avant les catégories cartésiennes closes. Tantôt le langage concret est une mise en œuvre de la théorie, tantôt la théorie s'efforce de capturer rétrospectivement le contenu conceptuel immanent à une famille de langages existants. Tel est l'esprit qui semble avoir guidé Gilles Dowek et Jean-Jacques Lévy, auteurs d'une excellente *Introduction à la théorie des langages de programmation* : « de même que la zoologie ne consiste pas à étudier successivement toutes les espèces animales, l'étude des langages de programmation ne consiste pas à étudier successivement tous les langages, mais s'organise autour des fonctionnalités que l'on retrouve dans les différents langages »<sup>2</sup>.

Déterminer l'éventuel manque auquel conduit cette réduction sera l'objet du présent travail : les langages de programmation théoriques épuisent-ils les propriétés des langages de programmation concrets, ou ceux-ci présentent-ils quelque apport propre, irréductible à leur structure conceptuelle ? Nous entendons montrer ici que la notion intuitive d'« expressivité » des langages de programmation serait mieux capturée par un appel à des outils philologiques et plus généralement linguistiques que par des outils formels.

1. O'Sullivan *et al.* (2008, p. 561). On pourrait contester cette distinction en exhibant quelques langages au statut intermédiaire, tels que Coq et Haskell, tout à la fois outils théoriques et outils de programmation concrets. Mais comme l'observe Georges Canguilhem, qui traite de la distinction entre santé et maladie en s'appuyant sur la théorie hegelienne de la mesure, tout écart quantitatif suffisamment flagrant engendre des différences qualitatives (Canguilhem, 1966/2013, p. 86–87).

2. Dowek et Lévy (2006, p. 100).

### 1.2. Calculabilité et compilation

Écartons d'entrée de jeu les objections qui prétendraient résoudre le problème en s'épargnant de le poser. Les langages de programmation sont presque tous complets au sens de Turing, c'est-à-dire qu'ils permettent de calculer tout ce qui est récursivement calculable, pour ne pas dire calculable en général<sup>3</sup> ; dans la mesure où ces langages possèdent à peu près la même puissance calculatoire, et où du reste les langages concrets se laissent tous réduire en dernière instance, par une chaîne de compilations, à quelque langage « primitif » tel que le langage machine, toute distinction entre eux n'est-elle pas illusoire ? Ainsi convergent l'argument de la calculabilité et celui de la compilation.

Mais l'argument de la calculabilité oublie qu'il n'est pas d'équivalence absolue. Comme tout jugement d'équivalence, l'équivalence au sens de Turing dépend d'une relation de congruence, dont toute personne soucieuse d'éviter la pétition de principe doit évaluer la pertinence *avant* d'en faire usage. Aussi Matthias Felleisen n'envisage-t-il pas cet argument sans objecter aussitôt qu'il mobilise un critère trop grossier pour rendre compte de la notion informelle d'expressivité d'un langage : « Il est vain de comparer l'ensemble des fonctions calculables qu'un langage peut représenter parce que les langages en question sont généralement universels »<sup>4</sup>. Pour comparer langages théoriques et langages concrets, il faut un grain plus fin que la calculabilité.

En appeler à l'argument de la compilation revient également à prévenir la question plutôt que la résoudre, la compilation ayant précisément pour mission d'aplatir les idiomes qui s'expriment dans les codes sources pour les réduire aux expressions d'un autre langage, généralement de plus bas niveau. Ici comme dans l'argument de la calculabilité, le critère lui-même préjuge la réponse, décidant à l'avance de négliger les différences qu'il est justement question d'examiner.

Aussi proposons-nous d'adopter le parti de comparer les langages de programmation tels qu'ils sont mobilisés, avant toute compilation, dans les codes sources. C'est dans ces textes que l'on a le plus de chances de voir s'exprimer une « pensée informatique » analogue à l'expérience mathématique que voulait décrire Jean Cavaillès. La sémantique à laquelle nous en appelons n'est donc pas celle du compilateur mais du programmeur.

## 2. Les langages de programmation théoriques

Il pourrait sembler trivial d'attendre d'un langage de programmation théorique que ses propriétés essentielles dérivent tout entières du contenu scientifique et de lui seul,

3. Du reste, les langages de programmation qui ne sont pas complets au sens de Turing sont soit dans le cas du HTML, pour lequel la complétude serait absurde et inutile, soit dans le cas de Coq, pour qui elle est indésirable : Coq y perdrait la propriété de normalisation forte, qui garantit la terminaison de tous ses calculs.

4. Felleisen (1991, p. 35).

évitant scrupuleusement tout ce qui serait de nature à brouiller le contenu conceptuel. Cette attente engendre à elle seule plusieurs exigences.

Une première exigence est ainsi celle de la pureté conceptuelle, qui implique en premier lieu d'éviter toute irrégularité syntaxique ; par exemple, toutes les fonctions y seraient notées selon une syntaxe uniforme. En second lieu, un langage de ce genre serait dénué de résidus historiques passivement reçus. Chaque langage, n'étant jugé que relativement au contenu conceptuel qu'il exprime, pourrait être considéré comme conçu *ex nihilo*, rien ne lui interdisant par exemple de réutiliser en un sens nouveau des signes ayant possédé autrefois une autre signification. Ainsi Per Martin-Löf attribue-t-il des règles nouvelles aux symboles  $\Sigma$  et  $\Pi$ , qu'Alonzo Church employait dans sa première version du  $\lambda$ -calcul, sans que cela engendre le moindre conflit conceptuel : autre langage, autres concepts<sup>5</sup>.

Une deuxième exigence serait que le langage soit en bijection avec les concepts qu'il doit exprimer, ce qui implique deux contraintes. La première est l'élimination de toute signe explétif, c'est-à-dire dénué de signification, comme l'est la négation dans l'expression française « Je crains qu'il *ne* parte »<sup>6</sup>. Un langage théorique doit être économique, c'est-à-dire n'exprimer que ce qui est conceptuellement pertinent ; Russell et Whitehead revendiquent ainsi dans la préface des *Principia Mathematica* que « nul symbole n'a été introduit sur un autre motif que son utilité pratique pour les objectifs immédiats de notre raisonnement »<sup>7</sup>. La seconde contrainte est l'élimination de toute redondance, principe que Schönfinkel formule explicitement :

*Il est conforme à la nature de la méthode axiomatique telle qu'elle est aujourd'hui reconnue avant tout à travers les travaux de Hilbert, non seulement de tendre en ce qui concerne le nombre et le contenu des axiomes à une limitation la plus étroite possible, mais aussi de tenter de réduire au maximum le nombre des notions de base non définies, en recherchant des notions qui soient de préférence appropriées pour construire toutes les autres notions du domaine de connaissance en question*<sup>8</sup>.

En vertu de ces deux contraintes, rejet de l'explétivité et élimination des redondances, chaque signe du langage doit recevoir une signification, et une seule.

Ces contraintes ont des conséquences pédagogiques. N'étant jugé qu'à l'aune des concepts qu'il exprime, le langage de programmation théorique peut n'être appris que par simples concepts, au moyen de sa définition abstraite. La présentation du langage PCF, dans l'ouvrage de Gilles Dowek et Jean-Jacques Lévy évoqué plus haut, est à cet égard tout à fait caractéristique : après quelques remarques générales sur la

5. Church (1932) ; Martin-Löf (1984).

6. Grevisse et Goosse (2011, §375, p. 490 : « Le mot explétif est un terme qui ne joue pas le rôle qu'il a l'air de jouer ; il est, logiquement, inutile, quoiqu'on ne puisse pas toujours le supprimer dans certains emplois figés »).

7. Whitehead et Russell (1910, Préface, p. VIII).

8. Schönfinkel (1924, p. 12).

syntaxe et la sémantique des langages de programmation, les auteurs en arrivent très tôt à la présentation inductive dans le langage BNF de la syntaxe du langage PCF<sup>9</sup> :

```
t = x
  | fun x -> t
  | t t
  | n
  | t + t | t - t | t * t | t / t
  | ifz t then t else t
  | fix x t
  | let x = t in t
```

L'objectif est pour les auteurs d'entrer aussi vite que possible dans le cœur du sujet : l'exposition des concepts et la démonstration des théorèmes.

Les propriétés que l'on peut attendre d'un langage de programmation théorique sont donc la pureté conceptuelle (régularité syntaxique et absence de résidu historique), la bijection avec les concepts exprimés (absence d'explétivité et de redondance) et l'apprentissage purement conceptuel — autant de caractéristiques qui les distinguent des langues naturelles, riches de toutes ces aspérités dont les langues théoriques cherchent précisément à se délivrer.

### 3. Les langages de programmation concrets

Les propriétés que nous avons identifiées sont-elles satisfaites par les langages de programmation concrets ?

#### 3.1. Irrégularités syntaxiques et traces historiques

Les langages de programmation concrets présentent bien des irrégularités syntaxiques, qui rappellent bien souvent leurs origines historiques.

Tel est le cas des déclarations de type en langage C. Depuis la normalisation du C publiée par l'ANSI en 1989, la définition d'une fonction doit contenir explicitement son type et ceux de ses arguments. La fonction `atoi`, qui convertit en un nombre entier une chaîne de caractères, est ainsi annoncée dans la bibliothèque standard<sup>10</sup> du langage C de la façon suivante :

```
int atoi(const char *nptr);
```

Ce prototype indique que la fonction correspondante prend pour argument un tableau de caractères — ou plus précisément un pointeur sur un caractère (`char *`) — et renvoie un nombre entier (`int`). C'est de cette façon que les types d'entrée et de sortie de toute fonction doivent être déclarés en C depuis la normalisation ANSI. Et pourtant, dans la seconde édition (1988) du manuel historique *Le Langage C*, qui tient compte

9. Dowek et Lévy (2006, p. 23–24).

10. *ISO C Standard 1999* (1999, section 7.20.1.2).

de la norme ANSI, Brian Kernighan et Dennis Ritchie définissent la fonction `main` d'un convertisseur de températures en ne stipulant ni le type de ses arguments ni son type de retour<sup>11</sup> :

```
#include <stdio.h>

main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

L'entrée et la sortie de cette fonction sont toutes deux nulles : l'exécution ne dépend d'aucun argument et ne produit rien d'autre que des effets de bord, en l'occurrence l'affichage d'un texte sur la sortie standard, qui peut être l'écran. Si la syntaxe avait été régulière, les auteurs auraient dû, en toute rigueur, écrire le code C

```
void main(void)
```

en utilisant le mot-clef `void`, que la norme ANSI du C emprunta au jeune langage C++ créé par Bjarne Stroustrup au début des années 1980. Mais `void` n'existant pas dans la première version du langage, celle du manuel de référence de 1974 et de la première édition du *Langage C*<sup>12</sup>, la syntaxe obsolète est restée autorisée dans les versions ultérieures du langage :

pour assurer la compatibilité avec les anciens programmes en C, les compilateurs qui suivent la norme considèrent une liste d'arguments vides comme une déclaration sous l'ancienne forme, et dans ce cas, ils ne vérifient pas la nature des arguments ; pour indiquer explicitement une liste vide, il faut se servir du mot `void`<sup>13</sup>.

Comme grevé par son histoire, le C de la norme ANSI tolère ainsi une irrégularité dans la syntaxe. Ce type d'irrégularités ne devrait pas advenir dans des langages théoriques, car en rompant la généralité du style elles brouilleraient l'exposition des concepts.

Ce n'est pas la seule trace que le langage C actuel garde de son histoire. Il existe une liste de mots « réservés », que l'utilisateur ne peut utiliser librement lorsqu'il souhaite par exemple nommer ses variables ou ses fonctions<sup>14</sup>. Presque tous ces mots sont des mots du langage, à l'exception notable de `register`, aujourd'hui obsolète, qui permettait de ranger une variable dans un registre du processeur ; le procédé était particulièrement indiqué lorsque le programme avait besoin de consulter cette variable

11. B. W. Kernighan et Ritchie (2004, p. 13).

12. Ritchie (1974) ; B. Kernighan et Ritchie (1978).

13. B. W. Kernighan et Ritchie (2004, p. 33).

14. Ces mots sont `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile` et `while`.

rapidement ou fréquemment. Mais les compilateurs modernes sachant bien mieux que les programmeurs déterminer quelles variables il convient de garder dans les registres, ce mot-clef en est presque toujours ignoré et ne reste qu'à titre de survivance d'un état antérieur du langage. De même, Java possède des mots-clefs réservés `goto` et `const` qui n'ont jamais eu de signification dans le langage et n'en auront probablement jamais, mais en ont eu dans les langages dont s'inspire Java : le C et le C++.

On observe également des phénomènes d'« exaptation<sup>15</sup> » : certains signes du langage, devenus obsolètes, y sont restés à l'état dormant, se rendant dès lors disponibles pour recevoir ultérieurement de nouvelles significations. Un cas particulièrement frappant est celui du mot `static` en Java, héritier d'une histoire complexe :

*L'historique du terme « static » est curieux. Le mot clé static a été introduit d'abord en C pour indiquer des variables locales qui ne disparaissaient pas lors de la sortie d'un bloc. Dans ce contexte, le terme « static » est logique : la variable reste et elle est toujours là lors d'une nouvelle entrée dans le bloc. Puis static a eu une autre signification en C, il désignait des variables et fonctions globales non accessibles à partir d'autres fichiers. Le mot clé static a été simplement réutilisé pour éviter d'en introduire un nouveau. Enfin, C++ a repris le mot clé avec une troisième interprétation, sans aucun rapport, pour indiquer des variables et fonctions appartenant à une classe, mais pas à un objet particulier de la classe. C'est cette même signification qu'a ce terme en Java<sup>16</sup>.*

Les langages de programmation concrets ont ainsi leur étymologie, riche en accidents et en contingences.

On n'attendrait pas d'un langage théorique de telles irrégularités syntaxiques ni tant de résidus historiques. Ce sont autant de traces archéologiques qui érigent les langages de programmation en authentiques lieux de mémoire. On ne peut comprendre toute leur syntaxe sans apprendre également des pans de leur histoire. Il n'en va guère autrement lorsque l'on cherche à comprendre la composition, rarement logique, des caractères chinois : il faut sans cesse invoquer les déformations accidentelles et les prononciations antiques qui justifient la ressemblance actuelle de deux caractères. En chinois comme en C, c'est bien souvent l'histoire qui rend raison de la structure.

### 3.2. Explétivité et redondances

Les langages de programmation concrets rompent également la bijection attendue entre les signes du langage et les concepts qu'ils mobilisent.

On y trouve d'abord des formes d'explétivité. Le signe le plus évident en est la commande vide, comme l'instruction `NOP` (*no operation*) des processeurs x86. De

15. Gould et Vrba (1982).

16. Hortsman et Cornell (2003, p. 158).

même, la seule fonction de l’instruction `Proof` en Coq est de montrer explicitement le passage du mode commande au mode démonstration, mais elle est facultative :

```
Theorem elim_conj :
  forall P Q: Prop, P /\ Q -> P.
Proof.
  intros P Q PQ. destruct PQ. assumption.
Qed.
```

Le choix des noms de variables et de fonctions doit également faire l’objet du plus grand soin, alors même que le compilateur n’en a cure.

Les commandes explétives les plus fréquentes sont les commentaires, les lignes blanches, les indentations et les espaces vides : autant de signes du code source qui, dans un grand nombre de langages, sont techniquement « inutiles ». Purement et simplement ignorés par le compilateur, ces signes jouent un rôle crucial dans la lisibilité du code source par des êtres humains. Ils aèrent le texte et créent des alignements qui permettent d’appréhender d’un seul coup d’œil un bloc entier de code, comme le montre l’incipit du fichier `param.h` de la version 6 d’Unix<sup>17</sup>, écrit en C :

```
0100 /* fundamental constants: do not change */
0101
0102
0103 #define USIZE    16
0104 #define NULL     0
0105 #define NODEV    (-1)
0106 #define ROOTINO  1
0107 #define DIRSIZ   14
0108
0109
0110 /* signals: do not change */
0111
0112
0113 #define NSIG      20
0114 #define SIGHUP    1      /* hangup */
0115 #define SIGINT    2      /* interrupt (rubout) */
0116 #define SIGQUIT   3      /* quit (FS) */
0117 #define SIGINS    4      /* illegal instruction */
...
```

La partie explétive d’un code source occupe parfois une grande partie du fichier. Sur les 451 lignes du code source de la fonction racine carrée (`sqrt`) de la bibliothèque Java du compilateur GCC<sup>18</sup>, plus de 350 sont des commentaires détaillés sur l’algorithme utilisé ainsi que sur d’autres algorithmes possibles, proportion qui n’a

17. Lions (1996, p. 01).

18. Il s’agit du fichier `gcc-4.9.1/libjava/classpath/native/fdlibm/esqrt.c` (daté du 18 janvier 1995) du compilateur GCC (*GNU Compiler Collection 4.9.1*, s. d.). Cet algorithme repose sur la méthode de Newton, optimisée grâce au codage binaire des nombres.



d'ailleurs rien d'extraordinaire<sup>19</sup>. Ce trésor de pensée algorithmique fond à la compilation comme neige au soleil.

On pourrait presque, selon un procédé familier aux mathématiciens, transformer la propriété en axiome et définir les langages de programmation concrets comme étant précisément les langages possédant un signe de commentaires<sup>20</sup>. Si un langage n'en possède pas, c'est qu'il est conçu pour écrire des codes suffisamment courts pour être précédés et suivis de commentaires écrits dans le métalangage, typiquement la langue de l'article qui cite ces courts extraits de code. Si, à l'inverse, le langage possède un signe de commentaires, c'est qu'il est conçu pour des programmes suffisamment amples pour que le métadiscours doive être inséré à l'intérieur du code.

La programmation littéraire ou lettrée (*literate programming*) montre de la façon la plus radicale l'apport propre du code source par rapport au code compilé. Il s'y produit en effet une inversion : l'essentiel du texte à lire est le commentaire et le code réellement « efficace » devient pour ainsi dire le commentaire du commentaire. La programmation lettrée est pour ainsi dire le passage du code source à son dual. Initié par Donald Knuth<sup>21</sup>, ce style de programmation est par exemple pratiqué de manière systématique par Gérard Huet dans une grande partie de ses articles et jusque dans son cours de calculabilité, intégralement rédigé en Pidgin ML au lieu de l'habituel métalangage des mathématiques<sup>22</sup>. Le texte rédigé en langue naturelle et le texte formel s'y entremêlent harmonieusement, ce qui permet à la fois d'offrir une lecture fluide et de ne laisser échapper aucun détail technique.

Symétriquement à ces signes explétifs, les langages de programmation concrets offrent de nombreuses redondances. En Perl, par exemple, on peut exprimer une boucle en utilisant l'instruction de choix `if` avec l'instruction de saut `goto` :

```
my $i=0;
marque :
if ( $i < 10 ) {
    print ( "Bonjour\n" );
    $i=$i+1;
    goto marque ;
}
```

19. Par exemple, dans le code source de Linux 2.2.14, le fichier `arch/i386/kernel/time.c` consacre plus de 400 lignes sur 688 aux commentaires et espacements. La proportion est sensiblement la même dans l'ensemble du code source de Linux.

20. Nous ne résistons pas au plaisir de citer la boutade de Gilles Dowek, à qui l'auteur de ces lignes soumettait ce critère, et qui donna un exemple de commentaire en  $\lambda$ -calcul ; il suffit d'écrire en début de programme

$$(\lambda xy.y)(commentaire)...$$

21. Knuth (1984).

22. Huet (2011).

ou avec une structure de contrôle `while` :

```
my $i=0;
while ($i<10) {
    print (" $i Bonjour\n");
    $i=$i+1;
}
```

ou avec une structure de contrôle `until` :

```
my $i=0;
do {
    print (" $i Bonjour\n");
    $i=$i+1;
} until ($i == 10);
```

ou encore avec une structure de contrôle `for` :

```
for (my $i=0; $i<10; $i++) {
    print (" $i Bonjour\n");
}
```

Coq est également riche en redondances ; il existe par exemple diverses façons, presque équivalentes, d'appliquer la règle logique d'élimination de l'implication (`apply`, `specialize`, `assert`, `cut`...), ce qui laisse à l'utilisateur la possibilité de choisir finement sa stratégie démonstrative. Il existe une synonymie parfaite entre les instructions `Theorem`, `Lemma`, `Example`, `Fact` et `Remark`, dont les nuances ne sont que psychologiques : par exemple, le « théorème » de Pythagore est certes un théorème quand on l'apprend pour lui-même, mais seulement un lemme quand on l'utilise pour démontrer que la diagonale du carré est incommensurable à son côté, et un simple exemple quand on le dérive du théorème d'Al-Kashi, dont il n'est qu'un cas particulier. Il n'y a pas en Coq de différence technique entre un théorème, un lemme, un exemple, un fait et une remarque : c'est au programmeur qu'il appartient de faire la distinction, selon le rôle que joue la proposition dans sa propre démarche démonstrative.

Les redondances, souvent désignées sous le nom de « sucre syntaxique », facilitent la tâche du programmeur en lui offrant tantôt les expressions les plus concises, tantôt au contraire les plus explicites pour exprimer sa pensée. Dans un langage théorique, la présence de ces redondances nuirait à la pureté conceptuelle : comment reconnaîtrait-on le concept de boucle s'il se présente différemment en chacune de ses occurrences ?

### 3.3. Apprentissage par la pratique

Enfin, on apprend rarement un langage de programmation concret par la définition inductive de sa grammaire. On commence généralement par recopier un programme affichant les seuls mots « *Hello world!* », selon une tradition inaugurée par Kernighan et Ritchie dans la première édition du *Langage C* (1978) :

```
#include <stdio.h>

main()
{
    printf("Hello world!\n");
}
```

L'exemple est devenu un passage obligé<sup>23</sup>. Le débutant recopie les programmes d'exemple, compile ou exécute chaque programme, observe le résultat ; ensuite seulement, l'auteur du manuel lui explique un par un les termes du programme<sup>24</sup>. La définition inductive, proposée sous la forme de Backus-Naur, ne figure généralement que dans le manuel de référence ou en annexe<sup>25</sup>, et s'adresse prioritairement aux auteurs de compilateurs futurs. On apprend donc les langages concrets comme les langages naturels : par essais, erreurs et corrections.

#### 4. Langages concrets et langues naturelles

##### 4.1. La naturalisation des langages formels

Les langages de programmation concrets possèdent ainsi des propriétés que l'on n'attendrait pas des langages théoriques supposés les représenter : l'irrégularité syntaxique et la présence de traces historiques, l'explétivité et les redondances, l'apprentissage par la pratique. Ce sont autant de propriétés que ces langages partagent avec les langues naturelles. Dans les langues naturelles, ces propriétés sont même reconnues comme positives, capables de donner naissance à des styles : les programmeurs ne sont pas des abeilles.

La ressemblance des objets justifiant parfois celle des méthodes, on peut se demander s'il est possible de transposer à l'étude des langages de programmation des méthodes initialement destinées aux langues naturelles. Plutôt que de formaliser une langue naturelle comme le fait Chomsky avec un fragment de l'anglais<sup>26</sup>, il s'agirait ici de naturaliser certains langages formels.

Telle entreprise ne saurait être confondue avec celles de Felleisen et de Mitchell, qui ont proposé des critères formels pour définir et comparer l'expressivité des langages de programmation<sup>27</sup> en s'appuyant sur des morphismes entre langages :

*Étant donnés deux langages de programmation universels ne différant l'un de l'autre que par un ensemble de structures de programmation  $\{c_1, \dots, c_n\}$ , les deux langages entretiennent cette relation si les structures que possède le langage le plus gros le rendent plus expressif que*

23. Stroustrup (2000, p. 50) ; Wall (2000, p. 3) ; Schwartz *et al.* (2011, p. 13).

24. B. W. Kernighan et Ritchie (2004, p. 6–7).

25. B. W. Kernighan et Ritchie (2004, annexe A13, p. 237–243) ; Stroustrup (2000, annexe A).

26. Chomsky (1957, notamment aux ch. 4 et 12).

27. Felleisen (1991) ; Mitchell (1991).

*le plus petit. Par « plus expressif », on veut dire ici que pour traduire dans le plus petit des deux langages un programme contenant des occurrences de l'une des structures  $c_i$ , il faudrait réorganiser le programme tout entier<sup>28</sup>.*

L'outil formel proposé par Felleisen ne permet pas de rendre compte de formes d'expressivité telles que l'usage de commentaires et d'espaces vides.

#### 4.2. *Le choix du langage*

Il ne s'agira pas non plus de juger les langages, comme on le fait parfois, à l'aune de leur seul « niveau d'abstraction », selon le point de vue d'inspiration évolutionniste qui voudrait faire coïncider l'ontogenèse et la phylogenèse des langages de programmation en les représentant par des degrés d'abstraction successifs : langages machine et assembleur, langages structurés, langages orientés objet. L'intelligibilité d'un langage pour un programmeur ne dépend pas tout entière de son niveau d'abstraction. C'est ce que nous verrons en examinant selon quels critères un programmeur choisit le langage qui lui permettra de réaliser une tâche donnée.

Le premier critère est celui des familles de langages : programmation structurée, programmation logique, langage de requêtes, programmation web, etc. Les autres critères sont autrement moins triviaux.

Le second critère est bien celui du niveau d'abstraction du langage, qu'il ne faudrait pourtant pas interpréter comme unilatéral. Il résulte en effet d'un équilibre entre deux tendances opposées : l'une, ascendante, porte le programmeur vers des langages épousant autant que possible la structure dans laquelle il tend à organiser ses représentations ; l'autre, descendante, le porte vers des langages décrivant aussi précisément que possible comment la machine exécutera les instructions. Il n'y a donc pas d'évolution naturelle vers les langages de haut niveau : si Windows est écrit en C++, Linux est resté fidèle au langage C d'Unix, plus léger, et les parties sensibles des systèmes d'exploitation et des pilotes restent aujourd'hui encore en partie écrites en assembleur<sup>29</sup>.

Un troisième critère, sociologique, est celui des traditions disciplinaires. Le FORTRAN, qui passe pour obsolète auprès des jeunes programmeurs, reste plebiscité par les physiciens, non tant pour ses propriétés intrinsèques que par atavisme : leurs prédécesseurs y ont codé tant de bibliothèques de fonctions qu'un changement de lan-

28. Felleisen (1991, p. 36).

29. À l'époque de MS-DOS 1.1 (1981) et 2.0 (1984), dont les codes sources ont été publiés début 2014 (<http://www.computerhistory.org/atchm/microsoft-ms-dos-early-source-code/>), le code source tout entier d'un système d'exploitation était écrit en assembleur. Aujourd'hui, le code source de Linux contient, dans le répertoire `arch/`, près de 400 000 lignes de code en assembleur spécifiques aux différentes machines (Linux 3.17.1, s. d.). Certains systèmes d'exploitation modernes, comme AcidOS et KolibriOS, sont encore tout entiers écrits en assembleur. Cette famille de langages n'a donc rien d'obsolète.

gage s'accompagnerait d'un fastidieux travail de réécriture. Ada bénéficie de la même popularité dans les milieux aéronautiques.

Un quatrième et dernier critère relève de considérations psychologiques individuelles. Perl et Python sont des langages globalement équivalents dans le sens où ils possèdent le même niveau d'abstraction, les mêmes structures de données, les mêmes structures de contrôle, et permettent tous deux de programmer dans une grande variété de styles — procédural, fonctionnel, orienté objet, etc. La principale différence entre les deux langages est esthétique. Python donne aux indentations une signification structurante dans le langage, alors qu'en Perl — comme en C, en C++ et en Java — elles ne servent qu'à rendre le code source plus lisible. Les indentations en Python jouent le rôle des accolades en Perl : elles servent à délimiter des blocs de commandes imbriqués. Selon le critère d'expressivité de Felleisen, les deux langages seraient équivalents, la traduction d'un programme de factorielle de Python en Perl ne produisant pas un programme significativement plus verbeux. Le code en Python contiendrait en effet les instructions

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

et le code Perl lui serait comparable presque ligne à ligne :

```
sub fact ($) {
    my ($x) = @_;

    if ($x < 2)
    { return 1; }
    else
    { return ($x * fact($x - 1)); }
}
```

Mais pour le programmeur au travail, le choix entre Python et Perl relève d'une évidence qui n'a pas de meilleure justification que le jeu des circonstances ou une mauvaise foi dont il est le premier à s'amuser.

#### 4.3. Aspects philologiques et stylistiques

Ni la notion formelle d'expressivité ni la pyramide des niveaux d'abstraction ne suffit à rendre compte de l'écart entre les langages théoriques et les langages concrets. Échappant à la pure technique, bon nombre des aspects des langages concrets relève plus généralement de la linguistique.

Les langages de programmation concrets sont sujets à une philologie aussi bien synchronique que diachronique. D'un point de vue diachronique, certains langages dérivent de langages antérieurs comme le français du latin ; ainsi une partie non négligeable de la syntaxe du C — notamment certains mots-clefs et l'utilisation d'accolades

pour délimiter les blocs — a-t-elle été reprise par C++, Perl, Java, JavaScript et bien d'autres langages. Bien des langages naissent ainsi d'autres langages, feignant d'en être des améliorations ou des évolutions alors qu'ils pourraient assumer, jusque dans leur syntaxe, de faire du passé table rase. Faciliter la migration n'est pas le moins important des effets recherchés par cette fiction.

On constate également des phénomènes d'importation dans l'évolution d'une langue. Comme l'observe Leo Spitzer au sujet de Spinoza, certains auteurs du XVII<sup>e</sup> siècle ont emprunté le *tó* grec pour pallier l'absence d'article défini en Latin, écrivant par exemple « *tó ens* » pour distinguer « l'étant » de « *un étant* »<sup>30</sup>. De même les « classes » de Simula, caractéristiques de la programmation orientée objet, ont-elles été intégrées dans le langage « *C with classes* » qui deviendra le C++, et jusque dans certaines versions récentes de Lisp, langage pourtant traditionnellement fonctionnel.

L'importation n'est d'ailleurs pas toujours unilatérale : deux langues contemporaines peuvent également s'enrichir mutuellement. Le C++, originellement dérivé du C, l'a ensuite influencé en retour avec la norme C99. Celle-ci reprend des fonctionnalités de C++ comme les commentaires en fin de ligne, les fonctions `inline`, la reconnaissance des déclarations comme instructions à part entière, etc. Ce qui est particulièrement remarquable est le fait que le langage ainsi enrichi ne soit pas conçu comme un « nouveau » langage, comme l'était le C++, mais comme un simple enrichissement du C. Les langages C et C99 ne sont pas reconnus comme deux langages différents, mais comme deux instantanés d'un « même » langage, qui évolue avec son temps, c'est-à-dire avec l'évolution du matériel et des pratiques de programmation comme la langue de Ronsard s'est adaptée à l'évolution des techniques et des formes de vie des quatre derniers siècles.

Il existe également, en programmation comme dans les langues naturelles, des langues vivantes et des langues mortes. Perl a ainsi connu de nombreuses versions qui témoignent de sa vivacité — Perl 1.0 en 1987, Perl 2.0 en 1988, Perl 3 en 1989, Perl 5 en 1994, Perl 6 depuis 2000 — tandis que d'autres langues, parmi lesquelles les langages B et BCPL, ancêtres du C, n'ont guère plus de locuteurs et ont cessé toute évolution<sup>31</sup>. Dans ses différents états, qui forment autant de langues différentes, on reconnaît toujours une seule et même langue, quelque nombreuses que soient les différences entre le C du « K&R » — le premier manuel de Kernighan et Ritchie — et le C de la norme ANSI<sup>32</sup>. On peut ainsi transposer à l'étude des langages de

30. (Spitzer, 2007, p. 172–173).

31. On peut consulter, entre autres mesures de popularité des langages de programmation, l'indice TIOBE (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>), qui classe les langages selon le critère — d'ailleurs contestable — du nombre de requêtes tapées dans les moteurs de recherche. En octobre 2014, B et BCPL ne sont pas même classés.

32. Quiconque lit aujourd'hui le code source d'Unix v6 ne peut manquer d'être surpris par les légers archaïsmes de son style : les variables du registre sont déclarées implicitement comme entières par des instructions telles que `register c` au lieu du « pédant » `register int c` (Lions, 1996, p. 3-2); les variables ne sont jamais initialisées explicitement à 0 ; les fonctions de type `int` ou `void` ne sont jamais déclarées comme telles ; etc.

programmation concrets et de leur histoire des méthodes empruntées à la philologie historique.

L'usage des langages de programmation concrets donne aussi lieu à une approche stylistique. On reconnaît ainsi la pratique d'une grammaire normative dans la façon dont des programmeurs ont condamné le « code spaghetti » et débattu de la pertinence des instructions de saut<sup>33</sup>. D'autres publient des livres entiers de conseils aux programmeurs, où ils recommandent le beau style : utiliser des commentaires éclairants, donner des noms judicieux aux variables et aux fonctions, espacer et indenter le code ; dans leur livre *Practice of Programming*, Brian Kernighan et Rob Pike font explicitement le lien entre l'écriture d'un beau code et l'écriture d'un anglais élégant, renvoyant à la lecture des *Elements of Style* publiés par William Strunk et Elwyn Brooks White en 1918<sup>34</sup>. Le programmeur exigeant en C a l'habitude de compiler ses programmes non seulement avec l'option « `-Wall` » du compilateur `gcc`, qui affiche tous les avertissements, mais également avec l'option « `-pedantic` », qui sanctionne tout impureté stylistique ; le programmeur Perl possède des instructions équivalentes, `use warnings` et `use strict`. Ces commandes montrent que le programmeur ne se soucie pas seulement d'efficacité technique, mais aussi occasionnellement de pureté du style. La programmation a ses Vaugelas et ses Grevisse.

Le prolongement naturel de ces considérations serait d'examiner si, pour les langages de programmation comme pour les langues naturelles, la langue ne s'épanouit pas dans une littérature. Leo Spitzer écrivait en effet :

*Le meilleur document pour l'âme d'une nation c'est sa littérature ; or celle-ci n'est rien d'autre que sa langue telle qu'elle est écrite par des locuteurs privilégiés. Ne peut-on pas alors saisir l'esprit de cette nation dans ses œuvres littéraires les plus importantes*<sup>35</sup> ?

Le « beau code » a ses « Lagarde et Michard »<sup>36</sup>. S'il fallait trouver à la poésie épique, genre noble par excellence dans les langues anciennes, un équivalent en programmation, nul doute que l'on s'arrêterait sur les systèmes d'exploitation. Ces œuvres de l'esprit sont d'immenses textes définissant par le menu, en s'appuyant sur les éléments disponibles dans le boîtier, toute l'ontologie de la machine avec laquelle l'utilisateur interagira : les notions de processus, de fichier, de noyau... La popularité du C ne vient pas seulement de ses propriétés intrinsèques, mais également du grand œuvre qu'il a permis d'écrire, Unix, dans la lecture duquel une génération de programmeurs apprit le C comme on apprit le grec en lisant Homère<sup>37</sup>. Bien plus que dans les exercices scolaires, c'est dans les grandes œuvres que s'exprime la beauté d'une langue<sup>38</sup>.

33. Dijkstra (1968) ; Knuth (1979).

34. B. Kernighan et Pike (1999, p. 28) ; Strunk et White (1979).

35. Spitzer (1970 (1948), p. 53).

36. Oram et Wilson (2007).

37. Lions (1996).

38. Nous nous permettons de mentionner à ce sujet la création en janvier 2015 du séminaire « Codes sources » par l'auteur de ces lignes ainsi que les informaticiens Raphaël Fournier-S'niehotta (CNAM) et

## 5. Conclusion

On n’attend pas les mêmes propriétés d’un langage de programmation théorique et d’un langage concret. Ces derniers possèdent des vertus, chez eux positives et fertiles, dont on ne voudrait pas nécessairement dans les premiers, et qui permettent la transposition de méthodes philologiques et stylistiques, mais aussi phylogénétiques et étymologiques, qui ont fait leurs preuves pour les langues naturelles. Pour être mené de manière systématique, ce rapprochement exigerait de son auteur une connaissance fine des concepts et méthodes de la linguistique comparée et un talent de polyglotte aussi bien en programmation que dans les langues naturelles.

Pour jeune qu’elle soit, la philosophie de l’informatique peut se garder de certaines erreurs de jeunesse en s’inspirant de ses aînées. Du point de vue de la méthode, la lecture d’un code source paraîtra probablement bientôt aussi naturelle pour un philosophe de l’informatique que l’est celle d’un traité pour un philosophe des mathématiques. Du point de vue de l’objet, la philosophie de l’informatique peut s’enrichir d’un certain mouvement récent en philosophie des mathématiques en décrivant l’écart qui apparaît parfois entre les fondements et les pratiques. La théorie n’y perdra pas : peut-être découvrira-t-on rétrospectivement des propriétés latentes et d’abord insoupçonnées des outils théoriques.

### Remerciements

*L’auteur remercie le lecteur anonyme de la revue et Valérie Schafer pour leurs précieuses remarques.*

## Bibliographie

- Auerbach E. (1946/1968). *Mimésis. La représentation de la réalité dans la littérature occidentale*. Paris, Gallimard.
- Backus J. (1978). Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, vol. 21, n° 8, p. 613–641.
- Canguilhem G. (1966/2013). Essai sur quelques problèmes concernant le normal et le pathologique. In *Le Normal et le pathologique*. Paris, Presses universitaires de France.
- Chomsky N. (1957). *Syntactic Structures*. The Hague, Mouton & Co.
- Church A. (1932). A Set of Postulates for the Foundation of Logic Part I. *Annals of Mathematics*, vol. 33, n° 2, p. 346–366.
- Church A. (1933). A Set of Postulates for the Foundation of Logic Part II. *Annals of Mathematics*, vol. 34, n° 2, p. 839–864.
- Dijkstra E. W. (1968). Go To statement considered harmful. *Comm. ACM*, vol. 11, n° 3, p. 147–148.

---

Lionel Tabourier (LIP6). À chaque séance, un intervenant vient y présenter un code source de son choix et en propose un commentaire algorithmique, stylistique, historique, littéraire ou philosophique. Le code est ensuite discuté avec le public.



- Dowek G., Lévy J. (2006). *Introduction à la théorie des langages de programmation*. Les Éditions de l'École polytechnique.
- Felleisen M. (1991, décembre). On the expressive power of programming languages. *Science of Computer Programming*, vol. 17, n° 1–3, p. 35–75.
- GNU Compiler Collection 4.9.1*. (s. d.). Consulté sur <ftp://ftp.irisa.fr/pub/mirrors/gcc.gnu.org/gcc/releases/gcc-4.9.1/gcc-4.9.1.tar.gz> ((16 juillet 2014))
- Gould S., Vrba E. (1982). Exaptation - a missing term in the science of form. *Paleobiology*, vol. 8, p. 4–15.
- Grevisse M., Goosse A. (2011). *Le Bon Usage*. Bruxelles, De Boeck. (15<sup>e</sup> édition)
- Hortsmann C. S., Cornell G. (2003). *Au cœur de Java 2. Notions fondamentales*. Paris, CampusPress.
- Huet G. (2011). *Constructive Computation Theory*. Consulté sur <http://pauillac.inria.fr/~huet/PUBLIC/CCT.pdf>
- ISO C Standard 1999*. Rapport technique. (1999). Consulté sur <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>
- Kernighan B., Pike R. (1999). *The Practice of Programming*. Boston, Addison-Wesley.
- Kernighan B., Ritchie D. (1978). *The C Programming Language*. Upper Saddle River, Prentice-Hall.
- Kernighan B. W., Ritchie D. M. (2004). *Le Langage C*. Paris, Dunod.
- Knuth D. E. (1979). Structured programming with go to statements. In *Classics in software engineering*, p. 257–321. Upper Saddle River, NJ, USA, Yourdon Press.
- Knuth D. E. (1984). Literate programming. *The Computer Journal*, vol. 27, p. 97–111.
- Linux 3.17.1*. (s. d.). Consulté sur <https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.17.1.tar.xz> ((15 octobre 2014))
- Lions J. (1996). *Lions' Commentary on UNIX 6th Edition, with Source Code*. San Jose, USA, Peer-to-Peer Communications.
- Martin-Löf P. (1984). *Intuitionistic Type Theory*. Bibliopolis.
- Mitchell J. C. (1991). On Abstraction and the Expressive Power of Programming Languages. In T. Ito, A. R. Meyer (Eds.), *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*, vol. 526, p. 290-310. Springer.
- Oram A., Wilson G. (Eds.). (2007). *Beautiful Code*. Sebastopol, O'Reilly.
- O'Sullivan B., Goerzen J., Stewart D. (2008). *Real World Haskell*. Sebastopol, O'Reilly.
- Rechenberg P. (1990). Programming Languages as Thought Models. *Structured Programming*, vol. 11, p. 105–115.
- Ritchie D. (1974). *C Reference Manual*. Consulté sur <http://cm.bell-labs.com/cm/cs/who/dmr/cman74.pdf> (15 janvier 1974)
- Schönfinkel M. (1924). Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, vol. 92, p. 305-316. (trad. fr. par Geneviève Vandeveld, *Mathématiques et sciences humaines*, tome 112 (1990))

- Schwartz R. L., Foy B. D., Phoenix T. (2011). *Learning Perl* (6<sup>e</sup> éd.). O'Reilly.
- Spitzer L. (1970 (1948)). Art du langage et linguistique. In *Études de style*, p. 45–78. Paris, Gallimard.
- Spitzer L. (2007, printemps). Milieu et ambiance. *Conférence*, n° 24, p. 113–189.
- Stroustrup B. (2000). *Le Langage C++*. Paris, Pearson.
- Strunk W., Jr., White E. B. (1979). *The Elements of Style* (Third éd.). Macmillan.
- Wall L. (2000). *Programming Perl*. Sebastopol, O'Reilly.
- Whitehead A. N., Russell B. (1910). *Principia Mathematica*. Cambridge University Press.