Technique & Pratique
Collection dirigée par Emmanuel Cornet et Alexandre Hérault

Comment choisir un langage de programmation

Thomas Pornin

Consultant en cryptologie
Docteur en informatique
Ancien élève de l'École Normale Supérieure

Avant-propos

À la surprise générale et sans avertissement préalable, même de la part des auteurs de science-fiction les plus inventifs, l'ordinateur a déboulé brutalement dans nos sociétés humaines, au point d'envahir les foyers familiaux, depuis la fin des années 1970. La machine, désormais consommable et achetable en promotion au supermarché, fait maintenant partie de notre environnement quotidien.

Malgré cette apparente normalisation de la situation, l'engin conserve une part de mystère, à cause de son maniement souvent difficile, parfois ésotérique. Ceux qui maîtrisent la machine sont appelés des programmeurs, caste privilégiée de chamans qui ont percé les secrets du fonctionnement de l'ordinateur. Apprendre à programmer, c'est entrer dans cette société dont l'art est de plus en plus indispensable ; il n'est désormais plus envisageable d'être un jeune ingénieur sans savoir bredouiller quelques lignes de programme quand le besoin s'en fait sentir.

C'est là que les choses se compliquent : l'ordinateur se programme selon un certain langage, qui comporte des instructions. Il existe une myriade de langages différents. Face à son clavier, le programmeur se demande fatalement lequel il doit choisir, pour telle ou telle tâche. Et ce choix n'a rien d'évident. Le foisonnement est important, chaque concepteur de langage de programmation fait sa promotion selon des termes techniques obscurs, usant d'arguments dont l'importance pour une situation donnée est rarement explicitée. Chaque programmeur a son idée sur la question, ses langages favoris et ceux qu'il apprécie beaucoup moins. Le programmeur étant aussi un humain, il est autant prompt à l'éloge qu'à l'anathème, et les querelles sont légion.

Le but du présent ouvrage est de guider le lecteur dans cette jungle. Nous allons essayer de déblayer le terrain suffisamment pour que le choix puisse être fait sereinement en toutes circonstances, en gardant l'esprit ouvert, afin que l'informatique cesse d'être un obstacle et devienne ce qu'elle est censée être, à savoir un outil. Tout au long de ce livre, j'ai essayé d'être clair et didactique, et d'aller à l'essentiel. Il existe des milliers de langages de programmation, et si tous devaient être cités dans les 128 pages de cet ouvrage, il n'y aurait même pas une ligne par langage. Aussi, la concision a dû l'emporter sur l'exhaustivité.

Les informaticiens utilisent une pléthore de termes techniques. J'ai tenté d'expliquer ceux que j'utilise au fil du texte. En cas de doute sur un mot, dans ce livre ou dans un autre, je vous invite à consulter le glossaire qui occupe les dernières pages de cet ouvrage.

J'espère que vous aurez autant de plaisir à lire ce livre que j'en ai eu à l'écrire ; j'implore votre pardon pour mes pathétiques prétentions à l'envolée lyrique. Malgré tout, je pense que cet ouvrage saura vous être utile. Vos critiques comme vos éloges m'aideront à l'améliorer encore : vous pouvez en faire part à l'éditeur, à l'adresse

`contact@H-K.fr`

Si vous rencontrez ce que vous estimez être une erreur ou une imprécision gênante dans l'ouvrage, nous vous serions reconnaissants de nous en faire part également.

Bonne lecture !

Thomas Pornin

Table des matières

Chapitre 1

Introduction

L'ordinateur est un outil, un exécutant pointilleux, méticuleux, rapide, et totalement dénué d'imagination et d'initiative. La machine ne fait rien par elle-même ; elle ne fait qu'appliquer, à la lettre, des ordres simples et précis. Une suite de tels ordres est un *programme*. « Programmer » est l'activité du « programmeur » : il s'agit de mettre en forme une telle suite d'ordres.

Un « langage de programmation » est un dialecte dans lequel on peut exprimer des programmes. C'est un langage commun entre l'ordinateur et le programmeur, entre la machine et l'humain. L'ordinateur n'est pas capable de comprendre une langue humaine comme le français[1]. Il est trop stupide pour ça. Aussi est-ce le programmeur qui doit faire l'effort de parler d'une façon que la machine comprend. Il doit *apprendre* à converser avec l'ordinateur.

Il existe de nombreux langages de programmation ; plusieurs milliers au moins. Quand un programmeur veut faire son office, il doit *choisir* un langage parmi ceux qui existent et que sa machine comprend. Ce choix est épineux, et l'objet de ce livre est de vous guider dans cette aventure.

1.1 Ce qu'est un langage de programmation

L'ordinateur, en interne, a une « mémoire » où il stocke à la fois les données à traiter et les instructions qui indiquent comment faire ce traitement. Chaque instruction est en elle-même très simple ; les plus complexes ressemblent à « multiplie deux nombres ». La mémoire est constituée de cases

1. Il y a des gens qui y travaillent, mais ça n'est pas encore au point.

élémentaires pouvant contenir un 0 ou un 1 : on appelle cela un « bit ». Une suite de quelques bits représente, conventionnellement, une donnée (un nombre, un caractère, un point d'une image, etc.), ou une instruction pour l'ordinateur.

L'activité de programmation sur les premiers ordinateurs (il y a environ soixante ans) était très peu enthousiasmante : il fallait rentrer un par un les bits constituant les instructions pour la machine. Au départ, cela se pratiquait avec des boutons, puis avec des cartes perforées (un trou indiquant un 0, une absence de trou spécifiant un 1). La technologie progressant, on a fini par inventer le clavier, qui permet de taper des lettres et autres caractères.

Depuis lors, on programme avec du texte. Pas des phrases comme dans un roman (l'ordinateur ne saurait pas quoi en faire), mais des suites de caractères qui ont un sens précis, selon le langage de programmation. Voici par exemple un tout petit programme, écrit dans un langage de programmation appelé le « langage C » :

```
#include <stdio.h>

int
main(void)
{
        puts("Hello World !");
        return 0;
}
```

Ce programme, quand il est exécuté par l'ordinateur, affiche à l'écran le texte « `Hello World !` ». La plupart des caractères constituant ce programme sont importants : ne vous avisez pas de remplacer une accolade par une parenthèse, ou d'oublier un point-virgule ; l'ordinateur vous le reprocherait vertement dans un message d'erreur plus ou moins clair (*syntax error*, habituellement) et s'en tiendrait là. Notamment, la machine ne serait plus en mesure de comprendre le programme, et donc ne l'exécuterait pas. En revanche, il y a quelques points de détail qui n'importent pas à l'ordinateur :

par exemple, j'ai ajouté quelques espaces avant le mot `puts` (il est décalé vers la droite) ; ces espaces supplémentaires n'ont pas d'incidence sur le comportement de la machine, elle les ignore[2]. Ces espaces servent à rendre le programme plus lisible pour le programmeur.

C'est le langage de programmation qui définit précisément ce que l'ordinateur va comprendre de cette suite de caractères. On dit que le langage définit la *syntaxe* et la *sémantique* des programmes : la syntaxe indique quelles sont les suites de caractères qui ont un sens dans le langage (et donc quelles suites sont absorbables par la machine), et la sémantique décrit ce que l'ordinateur doit faire pour chaque programme syntaxiquement valide. Dans l'exemple ci-dessus, la syntaxe dit que si `puts` est suivi d'une parenthèse ouvrante, puis d'un texte entre guillemets doubles, puis d'une parenthèse fermante et d'un point-virgule, alors ce morceau de programme est valide. La sémantique qui lui est associée est que le texte entre guillemets doit être affiché à l'écran.

À titre d'illustration, voici des programmes équivalents, dans deux autres langages de programmation. En « Java » :

```
public class Hello {
    public static void main(String[] args)
    {
        System.out.println("Hello World !");
    }
}
```

et en « Forth » :

```
: main ( -- )
    ." Hello World !" CR ;
main
```

Ces deux exemples montrent à la fois la grande diversité des langages de programmation, mais aussi leurs points

2. Cette absence de prise en compte des espaces entre les mots n'est pas universelle ; cela dépend du langage de programmation. La plupart des langages, dont le C, les ignorent.

communs. Au-delà des différences de syntaxe et de sémantique, on peut voir que tous ces langages s'expriment en utilisant une liste assez restreinte de caractères, typiquement les lettres latines, les chiffres et des symboles de ponctuation (pas d'idéogramme cunéiforme sumérien, par exemple). Quand des lettres forment un mot, le mot a une consonnance anglophone. La tradition veut qu'on représente les caractères façon « machine à écrire » : chaque caractère a la même largeur, ce qui permet de les aligner avec des espaces.

Ces caractéristiques relèvent de la Tradition. On peut imaginer des langages de programmation où les mots sont en français. On peut concevoir des systèmes où les mots ont une couleur, couleur qui a une signification pour la machine. On peut envisager l'usage de symboles divers comme ceux qu'on utilise couramment en mathématiques (lettres grecques, flèches, etc.), voire remettre en cause la structure linéaire du programme comme un texte lu de gauche à droite et de haut en bas. Tout cela est concevable, et a été tenté. Pour le moment, ces formes novatrices de programmation n'ont pas convaincu. Les symboles « spéciaux », et ceci inclut les caractères accentués comme le « é » français, posent des problèmes d'interopérabilité (les Américains auraient du mal à taper des accents sur leurs claviers qui ne comportent pas les touches correspondantes). La couleur est problématique pour les informaticiens daltoniens.

Aussi les langages de programmation que nous abordons ici suivent-ils tous le schéma traditionnel. Il convient de garder l'esprit ouvert et de faire attention aux innovations ; il est très possible que, dans l'avenir, une nouvelle forme de programmation supplante les programmes textuels simples. En attendant, de nombreux environnements de programmation utilisent des changements de couleur et de police de caractères pour faire ressortir, pour l'usage exclusif du programmeur, certains points de la structure du programme (le terme anglais est *syntax highlight*). Il s'agit alors de simples artifices de présentation, certes bien pratiques, mais qui ne font pas

partie du langage lui-même et qui n'ont pas d'incidence sur le sens des ordres envoyés à la machine. Il arrive d'ailleurs que ce mécanisme se trompe.

1.2 Pourquoi autant de langages ?

Une question qui se pose est alors : mais pourquoi existe-t-il plusieurs langages de programmation ? D'ailleurs, combien en existe-t-il ?

Ces deux questions ont une réponse commune : si tout le monde est d'accord pour dire que le langage brut de la machine – les instructions que comprend nativement l'ordinateur – est trop barbare pour être efficacement parlé par les programmeurs, personne n'est d'accord, au fond, sur la forme que devrait prendre le langage de programmation « idéal ». L'existence même d'un langage idéal, parfaitement adapté à toutes les situations, est remise en cause par beaucoup.

Aussi, depuis près de soixante ans, les informaticiens inventent, en moyenne, un nouveau langage de programmation par semaine. Inventer un langage dans ce contexte revient à :

- concevoir la syntaxe et la sémantique du langage ;
- décrire le langage dans une documentation technique ;
- implémenter les quelques outils nécessaires pour faire tourner les programmes écrits dans ce langage (c'est-à-dire les logiciels qui traduisent, d'une manière ou d'une autre, le langage en une suite d'instructions natives de la machine).

La conception d'un langage de programmation est une pulsion qui affecte à peu près tous les informaticiens. De plus, la démocratisation d'Internet permet à tout un chacun de publier ses œuvres, quels que soient leurs défauts et leurs qualités intrinsèques.

Un tel foisonnement est déjà déroutant pour le néophyte ; un effort de classification est nécessaire. Mais, plus gênant, de nombreux informaticiens ont des rapports passionnels et

passionnés avec les divers langages existants : l'objectivité s'amoindrit quand on parle de sa propre création, et la mauvaise foi est courante. Il y a des querelles de chapelles[3]. Aussi est-il difficile, pour le débutant, de faire la part des choses, de distinguer l'information technique de la propagande.

Dans le présent ouvrage, j'espère avoir évité les déballages trop flagrants de ma propre partialité.

1.3 Un langage, mais pour faire quoi ?

Puisqu'il n'y a pas de langage universellement idéal, il convient, à l'heure du choix, de se poser la question du but recherché : un langage de programmation, certes, mais pour faire quoi ? On peut distinguer principalement trois cas :

- choisir un premier langage, pour « apprendre à programmer » ;
- choisir un langage pour réaliser un projet précis, ou apprendre à réaliser un type de projet précis ;
- diversifier, de façon générale, ses compétences.

Débuter : l'activité de programmation repose sur une façon de penser un problème : quand on envisage de fabriquer un programme qui traite des données ou effectue une action, alors il faut concevoir intellectuellement la façon dont les choses vont se passer du point de vue de l'ordinateur. Ces schémas de pensée sont communs à tous les langages de programmation. Aussi peut-on parler « d'apprendre à programmer » en général, indépendamment du langage. C'est également ce qui fait qu'il est plus facile d'apprendre un langage particulier quand on en connaît déjà d'autres.

Néanmoins, pour apprendre à programmer, il faut bien se résoudre à choisir un premier langage qui servira de support pour les exemples et les essais. Programmer reste une activité

3. Tout cela n'a rien de spécifique à l'informatique : essayez donc de parler du PSG à un fan de l'OM.

« manuelle » : outre les connaissances théoriques, il faut acquérir des réflexes. Le premier problème qui se pose au débutant est donc le choix du langage à apprendre. Ce langage devra être disponible, si possible à un prix modique, pour le type d'ordinateur utilisé. Il devra être suffisamment simple pour ne pas noyer le débutant, mais suffisamment complexe pour qu'il soit possible de réaliser des miniprojets motivants. Le point le plus important est que le langage choisi ne devra pas enfermer l'apprenti-programmeur dans des schémas de pensée qui ne sont pas généraux. Acquérir des réflexes est bel et bon, mais cela ne doit pas se faire au détriment de la capacité à apprendre, par la suite, d'autres langages.

Réaliser : quand on commence à connaître un ou deux langages, on s'aperçoit assez vite que les aptitudes des différents langages à réaliser certaines tâches ne sont pas égales. Certes, théoriquement, elles sont équivalentes : tout ce qui est faisable dans un langage de programmation particulier l'est également dans n'importe quel autre langage[4]. Mais les modalités précises varient : certaines tâches s'expriment en trois lignes dans un langage donné, en trois cents dans un autre. L'effort investi par le programmeur peut ainsi considérablement changer selon le langage utilisé.

Nous nous plaçons ici dans le contexte suivant : un programmeur, maîtrisant un ou quelques langages, veut réaliser un projet particulier, ou apprendre à réaliser un type de projet. Par exemple, il veut apprendre à écrire des logiciels serveurs capables de réagir à des requêtes arrivant par réseau de façon efficace et robuste. Entre les langages qu'il connaît et ceux qu'il peut apprendre, le programmeur doit faire le tri.

Apprendre : l'envie d'en savoir plus est une composante essentielle de la psychologie humaine ; dans le cadre qui nous intéresse ici, cette curiosité peut pousser un programmeur à étudier d'autres langages. Dans ce cas, la question que se

4. On parle pédantesquement de « Turing-équivalence ».

pose le programmeur est de savoir lequel sera intéressant, lequel prendra place harmonieusement dans sa démarche intellectuelle. Il est certain que plus on connaît de langages différents, plus on est à même de discerner ce qui relève de la programmation en général de ce qui est spécifique à un langage donné. Par ailleurs, une vaste culture informatique est toujours une aide précieuse pour apprendre un nouveau langage : cela permet de progresser plus rapidement, en s'appuyant sur ce que l'on connaît déjà.

Mais l'extension de sa propre culture informatique n'est pas la seule motivation possible. On peut aussi vouloir apprendre un langage afin de se donner un avantage compétitif au sein d'une carrière. J'emploie ici ce terme au sens large : il désigne non seulement l'évolution de ses postes, tâches et responsabilités au sein du monde du travail, mais aussi la place que le programmeur aura dans le monde de la programmation en général. On peut par exemple vouloir apprendre un langage de programmation pour contribuer à des projets de logiciels libres ; cela constitue un choix de « carrière » même si la motivation n'est pas financière.

Le choix du langage est alors beaucoup plus influencé par l'environnement que par les qualités techniques des langages eux-mêmes. Les modes sont éphémères : un langage peut être très populaire à un instant donné, parce qu'il a une large communauté d'utilisateurs ou parce qu'il est activement soutenu par une grosse entreprise ; mais une telle situation n'est pas forcément éternelle. Il y a une douzaine d'années, beaucoup de programmeurs sur PC ne juraient que par le « Pascal » tel qu'édité par la firme Borland. L'engouement est depuis retombé. Si on programme en Pascal en 2005, on passe pour un dinosaure. En conséquence, si on choisit d'apprendre un nouveau langage pour améliorer sa carrière, il faut rester prêt à en apprendre d'autres, et à « abandonner » les langages dont la vie active est sur le point de s'achever[5].

5. Notons que certains langages truffés de défauts majeurs restent

1.4 Les pièges classiques

Apprendre et utiliser un langage de programmation est une activité intellectuelle qui, en retour, affecte l'intellect du programmeur. Qu'on le veuille ou non, la façon dont un problème est traité dans un langage de programmation conditionne la façon dont le programmeur envisage le problème. Ceci est particulièrement vrai pour le premier langage de programmation appris par le débutant. Idéalement, il ne faudrait pas apprendre *un* premier langage, mais au moins deux ou trois en même temps, de types variés. L'expérience montre que c'est pédagogiquement très difficile. Ainsi, ce qui se passe usuellement, c'est que le débutant apprend un langage, puis, à force de sueur, de larmes et de sang, il finit par se débarrasser des mauvais réflexes qu'il a acquis. C'est à ce prix que s'obtient la vraie *maîtrise*.

Les pièges dans lesquels le débutant et même le programmeur confirmé tombent souvent sont essentiellement les suivants :

- ramener tout langage à une sorte de variante abâtardie de ce qu'on connaît déjà ;
- défendre bec et ongles son langage favori quoi qu'il advienne, envers et contre toute logique ;
- remplacer la réflexion par la récitation de formules magiques ;
- essayer de prendre de haut un inconnu sur un forum de discussion publique (liste de diffusion, *newsgroup*, etc.).

Le premier piège revient à nier toute innovation externe. Si on a commencé à programmer en C, on continuera à faire « du C » dans n'importe quel autre langage. Cela revient à se restreindre au sous-ensemble commun des capacités de tous les langages abordés ; on appelle ça du nivellement par le bas, et cela conduit à l'inefficacité.

inexplicablement utilisés des décennies après leur date de péremption.

Le deuxième piège est un cas typique de passion excessive. Le programmeur tisse un lien affectif avec ses outils, et prend toute critique de ceux-ci comme une attaque personnelle. Il en résulte une stagnation intellectuelle, et des débats passionnés dont on dit qu'ils produisent beaucoup plus de chaleur que de lumière.

Le troisième piège est hélas un cas très répandu et favorisé par certaines institutions éducatives. Un cas typique est celui de la programmation « sûre » : afin d'éviter de laisser des trous de sécurité, c'est-à-dire des portes d'entrées pour un pirate qui passerait par là, on va bannir l'usage de certaines fonctionnalités qui, supposément, favorisent l'apparition de ces trous. Dans le cas du langage C, on dira : « N'utilisez pas `sprintf`, utilisez `snprintf`. » L'idée de base semble bonne : le programmeur s'impose l'utilisation de la fonction « moderne » (`snprintf`), où il doit exprimer explicitement les tailles des données qu'il manipule, ce qui permet au système de tronquer les données trop grosses plutôt que de les laisser déborder comme l'eau des pâtes sur un bec de gaz. Malheureusement, ce remplacement ne supprime pas le problème, il le cache sous le tapis : l'eau ne déborde plus, mais les pâtes ne seront pas cuites. Le véritable ennui est toujours présent, et le programmeur qui se contente de répéter le mantra appris par cœur l'ignorera en toute bonne foi.

Le quatrième piège est plus anecdotique, mais il arrive aussi : quand on intervient dans un forum public pour dire qu'un individu a tort, et que le langage A est de façon évidente supérieur au langage B, alors il se peut que l'individu dont on cherche à rabattre le caquet s'avère être l'inventeur du langage A *et* du langage B. S'il est d'humeur farceuse, les conséquences pour notre amour-propre peuvent être assez drastiques.

S'il fallait résumer l'état d'esprit à avoir quand on envisage l'apprentissage et l'utilisation d'un langage de programmation, ce serait : toujours savoir précisément ce que l'on fait, et garder l'esprit ouvert à d'autres méthodes.

Chapitre 2

Exprimer ses besoins

Lorsqu'on se pose la question du choix d'un langage de programmation, c'est qu'on se retrouve dans une situation qui requiert un travail de programmation, et où on a encore le choix du langage, au moins dans une certaine mesure. Ce terme de « situation » est très large : il recouvre aussi bien le cas d'un ingénieur qui doit faire tourner un logiciel spécifique pour un client[1] que celui d'un étudiant qui veut apprendre à programmer pour sa culture personnelle.

Cette grande diversité des situations a pour conséquence qu'il n'existe pas de langage universel. Chaque langage est bon à quelque chose, et plus ou moins adapté à une situation donnée. Il existe des milliers de langages de programmation ; pour s'y retrouver, une démarche possible est de faire une classification. Autrement dit, nous allons définir quelques critères qui permettent de caractériser chaque langage. Ceci permet, in fine, de faire passer chaque langage dans une grille d'évaluation qui devrait, si tout va bien, faire apparaître clairement le « bon » choix. C'est cette démarche que nous adoptons ici. Ce n'est pas la seule possible, mais dans la pratique elle fonctionne bien, et on peut décemment la qualifier de « scientifique ».

Nous allons dégager deux familles de critères :

- les critères « externes », qui correspondent aux besoins, à la situation présente ;
- les critères « internes », qui forment une classification des langages en fonction de ce qu'ils savent faire, de ce qu'ils apportent.

1. En général, les commerciaux vendent le logiciel avant même qu'il existe. Cela fixe des contraintes assez strictes sur les délais.

Chapitre 3

Typologie des langages

Dans le chapitre précédent, nous avons étudié une série de critères qui décrivent ce dont on a besoin, ce que le langage doit fournir. C'est une sorte de grille d'évaluation des capacités d'un langage.

Malheureusement, les langages de programmation sont rarement conçus et décrits en fonction de ces critères. La description typique d'un langage de programmation s'attarde sur la classification du langage selon des éléments de sa structure intrinsèque. Il existe ainsi une littérature considérable, en librairie et sur le Web, qui vante les mérites et fustige les défauts de divers langages de programmation sur la base de ces distinctions.

Je donne ici une description de quelques-uns de ces regroupements, en me concentrant sur ceux qui ont une influence tangible sur la façon de s'en servir. J'insiste sur le point suivant : aucune de ces classes de langages n'est intrinsèquement « meilleure » qu'une autre. Tout est une question de buts recherchés. Ce que l'on souhaite, c'est surtout savoir dans quelle mesure les détails du fonctionnement interne d'un langage influent sur ses capacités à remplir nos besoins.

3.1 Quelques notions

Pour bien comprendre les quelques sections qui suivent, il faut poser quelques notions.

Un ordinateur fonctionne fondamentalement par ordres. Il reçoit des instructions et les exécute, séquentiellement, sans rechigner et sans intelligence. La machine n'a aucune initiative, et elle ne comprend rien à ce qui se passe.

Chapitre 4

Les principaux langages

Dans ce chapitre, nous présentons quelques-uns des langages de programmation les plus courants, en commençant par les plus utilisés. Cette liste n'est bien évidemment pas exhaustive.

Pour chaque langage, nous décrivons succinctement ce à quoi il ressemble, d'où il vient et comment on peut se procurer les outils permettant de l'utiliser. Nous donnons également une évaluation du langage en fonction de ses capacités à remplir certains besoins, selon les dénominations du chapitre 2. Il y a bien évidemment une part de subjectivité dans cette évaluation ; mon jugement ne saurait être considéré comme absolu en la matière.

Ces évaluations sont reprises en une table synthétique à la fin du chapitre. Les capacités évaluées sont :

- disponibilité ;
- documentation (clarté et quantité) ;
- support (possibilités d'obtenir de l'aide) ;
- code existant ;
- liberté ;
- pérennité ;
- portabilité ;
- puissance d'expression ;
- performances ;
- temps d'apprentissage (pour devenir autonome) ;
- temps de maîtrise (pour « faire le tour » du langage).

Chaque « note » est `--`, `-`, ·, `+` ou `++`, par ordre croissant.

Chapitre 5

Le choix

Maintenant que nous avons vu dans le détail les critères qui doivent être pris en compte, ainsi que les fonctionnalités que peuvent offrir les divers langages de programmation, et succinctement résumé ce que les langages existants fournissent, il convient de réaliser le but de ce livre, c'est-à-dire choisir réellement un langage.

Nous avons choisi de présenter trois situations typiques :

- le débutant en programmation, qui cherche un premier langage ;
- l'amateur éclairé, qui veut réaliser un certain type de projet ;
- l'ingénieur qui souhaite améliorer ses compétences.

Pour chacune, nous présentons un arbre de décision, qui devrait permettre de trouver le « bon » choix en un coup d'œil. Ces arbres sont bien évidemment assez arbitraires et reflètent la façon dont, personnellement, je vois les choses. Il n'y a pas de choix universellement reconnu comme « bon » ; en revanche, il y a des choix ostensiblement médiocres.

Glossaire

Nous donnons ici des définitions succinctes de quelques termes techniques employés dans ce livre ou fréquemment utilisés dans les textes traitant des langages de programmation. L'usage de ce petit lexique devrait permettre de mieux comprendre certains jugements énoncés avec force sur des pages Web vantant tel ou tel langage de programmation.

Architecture : type de système matériel. L'architecture décrit essentiellement le type de processeur, et les grandes lignes de la façon dont le matériel est organisé. Si deux ordinateurs relèvent de la même architecture, alors ils pourront utiliser les mêmes systèmes d'exploitation et les mêmes logiciels.

Assembleur : transcription directe du langage natif de l'ordinateur. Chaque instruction (une suite de 0 et de 1) est transcrite en un mot dit « mnémonique » qui la représente. On dit aussi « langage d'assemblage ». On appelle également assembleur le logiciel qui convertit cette transcription vers le langage natif de la machine.

BASIC : langage « pour débutant » des temps anciens. Le BASIC était très populaire sur les micro-ordinateurs personnels, quand ces derniers ont commencé à se répandre, à la fin des années 1970. De nos jours, le BASIC est souvent cité comme exemple de langage archaïque et irrémédiablement mal conçu (rôle détenu auparavant par le COBOL, qui semble maintenant, Dieu merci, disparaître des mémoires).

Bibliothèque : ensemble de sous-programmes disponibles pour inclusion immédiate dans une application. L'accès à diverses ressources matérielles et logicielles passe par des biblio-

thèques spécifiques, dont le code fait le travail d'accès proprement dit. La conception d'un programme repose en premier lieu sur l'utilisation efficace de bibliothèques existantes. À noter : le mot anglais est *library* qui est souvent improprement traduit en « librairie ».

Binaire : système de numération en base 2, donc avec seulement des 0 et des 1. Par extension, on appelle « programme binaire » un programme écrit dans le langage natif de la machine, donc directement exécutable par elle.

Bytecode : nom donné généralement au langage « natif » d'une machine virtuelle, qui est elle-même émulée par un interpréteur ou un compilateur JIT. Le *bytecode* est appelé ainsi parce qu'il est souvent conçu pour être compact, avec des instructions tenant sur un ou deux octets[1]. On appelle parfois *wordcode* un *bytecode* qui cherche l'efficacité maximale de la machine virtuelle au détriment de la compacité, en utilisant des instructions larges.

Compilation : transformation du code source en une suite d'instructions compréhensibles par la machine. Cette transformation a lieu une fois pour toutes, avant le lancement du programme, et non en cours de route. La compilation est effectuée par un compilateur.

Concurrence : exécution simultanée de plusieurs morceaux de programmes sur la même machine, en se partageant des ressources communes. Quand le code source et la mémoire vive sont en commun, on appelle ces instances du programme des fils d'exécution, ou *threads*. La programmation concurrente consiste à organiser les *threads* pour qu'ils puissent travailler sans se marcher sur les pieds. On parle aussi de *multithreading*.

1. Le mot anglais « *byte* » désigne quasiment toujours un nombre binaire de huit chiffres, c'est-à-dire un « octet ».

CPU : *Central Processing Unit*. Dit « processeur » en français, c'est la pièce centrale de l'ordinateur, celle qui fait le vrai travail.

Développement : nom générique donné à l'activité de fabrication d'un logiciel. Le développement comporte la conception, la programmation, la validation et la documentation.

Encapsulation : paradigme de programmation consistant à isoler les données au sein d'un programme. Quelques procédures d'accès sont implémentées et systématiquement utilisées. L'encapsulation est une technique fondamentale de la POO.

Eval : nom usuel de la fonctionnalité de métaprogrammation de certains langages, consistant à fabriquer depuis le programme un nouveau morceau de programme et à l'exécuter immédiatement.

Fonctionnel : caractéristique d'un langage de programmation qui est fondé autour de la notion de fonction, qui travaille sur des arguments et renvoie une valeur. Exemple : OCaml.

Garbage collector : système de gestion automatique de la mémoire. Le GC décharge le programmeur d'une partie du travail de gestion des données en mémoire vive, en détectant automatiquement les données obsolètes (celles que le programme n'utilisera plus et qui donc occupent de la mémoire vive pour rien). Des traducteurs francophiles convaincus ont proposé « glaneur de cellules » et « ramasse-miettes » ; la traduction littérale serait « ramasseur de déchets » ou tout simplement « éboueur ».

Héritage : un des trois piliers de la POO. L'héritage est le concept de fabrication d'un type d'objet en étendant un type d'objet existant ; le nouvel objet peut remplacer l'ancien en toutes circonstances. Suivant les langages, l'héritage peut être simple (un objet est construit à partir d'un seul

ancêtre) ou multiple (un objet est construit à partir de plusieurs ancêtres). L'héritage multiple est plus puissant, mais plus complexe, à la fois à implémenter et à comprendre. Les avis sont partagés quant au type d'héritage qui est préférable. Java utilise l'héritage simple, C++ l'héritage multiple.

IDE : environnement de développement intégré. « IDE » est l'acronyme anglais (*Integrated Development Environment*) et est beaucoup plus utilisé que l'acronyme français « EDI ». Un IDE fournit sous une interface unifiée et cohérente tous les outils nécessaires à l'activité de programmation dans un langage donné.

Indentation : décalage typographique horizontal, ligne à ligne, du code source. Plus généralement, l'indentation désigne l'activité de modification d'un code source pour lui donner une forme plus harmonieuse et lisible pour les humains, sans changer le sens du programme du point de vue de la machine. Une indentation rigoureuse et systématique fait partie de la discipline que tout bon programmeur doit s'imposer. Notons que certains langages (par exemple Python) accordent une importance sémantique à l'indentation.

Impératif : caractéristique d'un langage de programmation dont la brique de base est la notion de séquence d'instructions, c'est-à-dire une liste d'ordres simples que l'ordinateur devra effectuer à la suite. Exemple : le langage C.

Interprétation : exécution d'un programme par un logiciel spécifique, dit « interpréteur », qui analyse le code source directement lors de l'exécution.

Introspection : capacité offerte par un langage, permettant à un programme de parcourir lui-même son propre code. Elle permet un style particulier de métaprogrammation. Exemple de langage introspectif : Java.

JIT : *Just In Time*. Ce sigle qualifie les interpréteurs qui travaillent par compilation rapide du code source au moment de l'exécution du programme.

Machine virtuelle : processeur fictif, émulé par un interpréteur ou un compilateur JIT. Les machines virtuelles permettent d'obtenir une grande portabilité des programmes, ainsi qu'une meilleure détection des erreurs, au prix de performances amoindries.

Mémoire vive : voir « RAM ».

Métaprogrammation : type de programmation où le programme génère un autre programme. Un langage offrant un bon support pour la métaprogrammation est dit extensible.

Multithreadé : type de programme utilisant la programmation concurrente avec plusieurs fils d'exécution (voir l'entrée « concurrence »). Ce terme est un ignoble mélange de français et d'anglais, comme les affectionnent les programmeurs.

Paradigme : cadre général de formulation et résolution des problèmes. Un paradigme de programmation est essentiellement une méthode d'organisation du programme et des données qu'il traite, indépendamment des détails de la syntaxe du langage utilisé. Par exemple, la programmation orientée objet est un paradigme.

Plate-forme : combinaison de l'architecture et du système d'exploitation. Un programme binaire est conçu pour une plate-forme donnée.

Polymorphisme : capacité de morceaux de programmes à traiter des données diverses de façon unifiée. Le polymorphisme est un des buts fondamentaux de la POO, mais il peut aussi exister indépendamment.

POO : programmation orientée objet. Il s'agit d'un paradigme de programmation qui consiste à modéliser les données

traitées sous la forme d'objets, chacun fournissant lui-même la façon de le traiter. La POO est praticable avec plus ou moins de bonheur suivant le langage de programmation. Tous les langages tendent à intégrer un support spécifique de la POO.

Procédure : suite d'instructions regroupées sous un seul nom et invoquées par ce nom depuis un programme. Le regroupement des instructions en procédures est la base de la programmation impérative structurée. Une procédure qui, en sortie, retourne une valeur au code qui a appelé la procédure, est aussi appelée une « fonction ».

Programme : suite d'instructions pour l'ordinateur, exprimées dans un langage de programmation.

RAD : développement rapide (de l'anglais *Rapid Application Development*). Le RAD est un concept de fabrication d'une application à partir de son interface utilisateur, avec des outils automatisés. Le RAD permet d'arriver très rapidement à un programme utilisable mais souvent de qualité médiocre.

RAM : mémoire de travail de l'ordinateur : toutes les données traitées y passent, au moins temporairement. Le contenu de la RAM (*Random Access Memory*) est perdu quand l'ordinateur est éteint. On l'appelle aussi « mémoire vive » par opposition à la ROM (*Read-Only Memory*), qui résiste aux coupures de courant, n'est pas modifiable, et est dite « mémoire morte ».

Script : petit programme monofichier à interprétation directe. Un script est écrit dans un langage de script. Ces langages sont en général conçus pour permettre de lier ensemble plusieurs applications et traiter simplement des données textuelles. Certains langages de programmation permettent à la fois l'écriture de scripts et d'applications de plus grande ampleur.

Sémantique : signification d'un programme, indépendamment de sa structure syntaxique. La syntaxe définit la forme

que peut prendre un programme, la sémantique décrit le sens de ce programme.

Source : nom donné à un programme tel qu'il est manipulé par le programmeur, donc dans le langage de programmation que le programmeur voit. On dit « code source » ou, par contraction, « le source ».

Spaghetti : type de code illisible. Le « code spaghetti » désigne un programme qui abuse des sauts (`goto`) au point de ressembler à un plat de spaghetti : les différents chemins d'exécution sont entremêlés au-delà de toute rédemption. Dans la pratique, cette expression est souvent utilisée pour jeter l'anathème, sans distinction, sur ce qui ressemble à un `goto` ; c'est un cas de mantra répété rituellement et remplaçant la réflexion.

Sucre syntaxique : construction d'ordre purement syntaxique, agréable pour le programmeur mais dénuée de portée théorique majeure. Il s'agit donc d'une fonctionnalité dont le programmeur pourrait se passer avec un peu plus de discipline. Cette expression est la traduction directe de *syntactic sugar*, qui correspond à une obsession anglo-saxonne pour les friandises sucrées vues comme archétypales du bonheur matériel. La littérature sur les langages fonctionnels utilise énormément cette expression, par tradition.

Syntaxe : ensemble des règles qui définissent ce qu'est un code source bien formé. C'est la syntaxe qui décrit quels éléments textuels (mots, symboles de ponctuation...) peuvent apparaître dans un langage donné, et dans quel ordre. D'un point de vue théorique, la syntaxe est sans importance ; mais le confort du programmeur, et donc son efficacité, en dépendent fortement.

Système d'exploitation : logiciel fondamental qui gère le matériel pour le compte des applications. Exemples de systèmes d'exploitation : Windows, Linux, Mac OS X.

Thread : fil d'exécution au sein d'un programme utilisant la programmation concurrente.

Typage : association d'un type à une valeur. Un « type » est ce qui dit que, par exemple, telle suite de 0 et de 1 représente un nombre, ou une chaîne de caractères. Le typage est utilisé pour détecter les erreurs de programmation, par exemple l'utilisation d'une valeur en mémoire pour ce qu'elle n'est pas. On parle de « typage statique » pour un mécanisme de détection qui a lieu entièrement lors de la compilation d'un programme ; le contraire (typage actif lors de l'exécution aussi) est dit « typage dynamique ».

UML : méthode de formalisation d'un problème selon une approche objet. UML est l'acronyme de *Unified Modeling Language*. UML s'appuie sur des diagrammes représentant les données manipulées dans un programme ; ces diagrammes sont dessinés dans un logiciel spécifique qui peut ensuite générer automatiquement un squelette du programme lui-même. UML est à la mode, et est donc enseigné quasiment systématiquement dans les écoles d'ingénieurs. UML *n'est pas* un langage de programmation.