Caractéristiques pour un langage pédagogique

Bruno Mermet [1]

Introduction

Langage « pédagogique » ?

Avant de déterminer les caractéristiques d'un langage « pédagogique », il faut que je précise ce que j'entends par ce qualificatif. Il s'agit d'un langage qui doit permettre à un novice d'assimiler rapidement les bases de la programmation, *sans dégoût*, et d'écrire assez rapidement des programmes présentant un minimum d'intérêt. Ce document, fondé sur de multiples retours d'expérience après plus de 25 ans à enseigner l'informatique, n'est bien sûr pas exempt de critères subjectifs.

Quelques mauvais exemples

Le langage d'assemblage [2] ne permet pas d'écrire rapidement des programmes présentant un minimum d'intérêt. Par ailleurs, le peu de contrôle sémantique possible, l'absence de structures de données de haut niveau, risque de provoquer du dégoût assez rapidement.

Le langage Java [3] n'est pas non plus un bon langage. Le fait qu'un simple *Hello World* s'écrive ainsi

1. Maitre de conférences, université Le Havre Normandie, secrétaire de la SIF.
2. https://fr.wikipedia.org/wiki/Assembleur.
3. https://fr.wikipedia.org/wiki/Java_(langage).

DOI:10.48556/SIF.1024.22.109 **1024** – Bulletin de la SIF, numéro 22, novembre 2023, p. 109–119

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

en rebutera plus d'un. Comprendre un tel code nécessite de parler de classe, de méthode de classe, etc.

Un langage comme C, qui ne permet pas de manipuler facilement les chaînes de caractères, un type basique pour l'utilisateur débutant, ne peut pas être non plus un bon langage.

©Luc Damas, https://luc-damas.fr/hop/quel-langage.

Et maintenant ?

Dans les parties qui suivent, je vais préciser certaines caractéristiques que je juge importantes pour un langage pédagogique. Bien sûr, il y aura quelques antagonismes, et il faudra faire des compromis ; j'essaierai donc dans une dernière partie de faire des propositions.

Caractéristiques requises

Gestion des erreurs du programmeur

Dans la conception d'interfaces homme-machine, la prise en compte des erreurs de l'humain est un critère fondamental (enfin... est censée l'être dans la théorie, car dans la pratique, c'est rarement le cas). Il faut normalement : limiter les risques d'erreur, détecter et expliquer les erreurs au plus tôt et proposer des solutions pour les corriger. Un langage pédagogique doit fournir ces trois caractéristiques.

Limiter les risques d'erreur

Pour limiter les risques d'erreur, il faut un langage défini par une grammaire stricte, et si possible, un outil de développement relativement contraignant. En ce qui concerne l'outil de développement, l'interface du langage Scratch [4], qui permet d'assembler des sortes de pièces de puzzle avec des connecteurs différents suivant

4. https://fr.wikipedia.org/wiki/Scratch_(langage).

ce dont il s'agit (instruction, expression, événement, etc.) est un très bon exemple. Un langage XML avec une DTD/un schéma et un éditeur XML est un autre bon exemple : la DTD ou le schéma permettent de contrôler très fortement ce qui peut être saisi.

Les langages similaires à C (Java, C++, C#, PHP, Javascript, etc.) constituent *a priori* un mauvais choix. Dans ces langages, le sens du programme suivant n'est pas fixé par la grammaire.

```
if (cond1)
        if (cond2)
                inst1 ;
        else
                inst2 ;
```

En effet, la grammaire seule ne permet pas de décider si l'instruction *else* se rapporte au premier ou au deuxième *if*. Sur ce plan, un langage comme Perl [5] a le mérite de lever cette ambiguïté, puisque les accolades sont imposées.

Un autre critère important est de disposer d'un langage fortement typé, avec le moins de conversions implicites possibles. Le typage peut être explicite (comme en Java), totalement inféré (comme en Haskell [6]) ou mixte (comme en Kotlin [7]).

Détecter et expliquer les erreurs au plus tôt

En ce qui concerne la détection des erreurs, un bon environnement de développement (IDE) est indispensable. Mais forcer le passage par un IDE dès le début de l'apprentissage n'est pas toujours aisé ni toujours souhaitable car l'IDE peut quelquefois masquer des caractéristiques importantes du langage. Il est donc indispensable que l'interpréteur ou le compilateur du langage soit efficient sur ce point là, afin que des messages d'erreur clairs soit proposés, y compris quand on ne passe pas par un IDE. Aussi, une analyse syntaxique et sémantique préalable d'un programme avant son exécution est indispensable. C'est en général le cas avec des langages compilés ou semi-compilés, mais cela pourrait aussi être envisagé avec des langages interprétés. Par exemple, le fait que le chargement du programme Python suivant ne provoque aucune erreur n'est clairement pas judicieux : la variable *m* de la fonction *g* n'est en effet pas définie.

```
def f(n):
        return n + 1
def g(n):
        return m + 1
print(f(4))
```

Outre la détection, l'explication des erreurs doit être accessible au novice. Sur ce plan là, Haskell est un exemple de ce qu'il ne faut pas faire. Par exemple, le chargement du programme suivant :

5. https://fr.wikipedia.org/wiki/Perl_(langage).
6. https://fr.wikipedia.org/wiki/Haskell.
7. https://fr.wikipedia.org/wiki/Kotlin_(langage).

```
f a b = a + f b
```

produit le message d'erreur suivant, qui a de quoi en rebuter plus d'un !

```
erreur.hs:1:1: error:
        Occurs check: cannot construct the infinite type: t ~ t -> t
                Expected type: t -> t
                  Actual type: t -> t -> t
        Relevant bindings include f :: t -> t (bound at erreur.hs:1:1)
  |
1 | f a b = a + f b
  | ^^^^^^^^^^^^^^^
Failed, no modules loaded.
```

Proposer des solutions pour corriger l'erreur

Sur ce point, les compilateurs ont fait de nombreux progrès. Prenons par exemple le fichier `Erreur.java` suivant.

```
public class Erreur2 {}
```

Lors de sa compilation, le message d'erreur produit est le suivant :

```
Erreur.java:1: error: class Erreur2 is public, should be declared in
a file named Erreur2.java
public class Erreur2 {
       ^
       1 error
```

Non seulement l'erreur est signalée, mais elle est expliquée (c'est parce que la classe est `public`), et une correction est proposée (changer le nom du fichier en `Erreur2.java`).

Niveau du langage

Le niveau du langage est également un critère qui peut être pris en compte. Un langage de trop bas niveau, s'il a le mérite de rendre explicite certains fonctionnements internes d'un ordinateur (ce qui pourra satisfaire les besoins de certains élèves) risque par contre d'en décourager d'autres par la complexité de mise en œuvre de traitements pourtant apparemment simples. Des langages intermédiaires, qui imposent une gestion manuelle de la mémoire, risquent également de nécessité l'explication de processus quelque peu compliqués au départ. Un langage de haut niveau devrait donc être privilégié. Cependant, il faut éviter les effets « magiques » qui risquent de décontenancer les élèves.

Éviter les `NullPointerException` et consorts

Les pointeurs nuls sont à la base de nombreux *bugs*. Certains langages ont, avec le temps, introduits des façons d'éviter ces erreurs, comme l'introduction du type `Optional<>` dans Java 8. Mais cela reste à la discrétion du développeur, et alourdit l'écriture. Les choix faits par des langages plus récents comme Kotlin et Rust [8], dont les versions standard des types interdisent ce genre de valeur, est de loin préférable. Par exemple, en Kotlin, écrire

```
var a: Int = null
```

est illégal. Une variable de type `Int` ne peut être affectée par `null`. Si on veut s'autoriser les valeurs nulles, il faudra écrire

```
var a: Int? = null
```

Les débordements d'indices d'un tableau doivent également être contrôlés. Le fait, comme en C, de pouvoir écrire hors de la zone mémoire allouée à la donnée que l'on croit modifier ne doit pas pouvoir être possible. C'est heureusement le cas pour la plupart des langages modernes.

Problème des conversions de type implicites

Cependant, un langage permettant trop de conversions implicites devient rapidement assez peu « sûr », et les erreurs ne sont révélées qu'assez tardivement.

Avantages

Les conversions implicites de type sont souvent vues comme des fonctionnalités très pratiques des langages. Prenons l'exemple d'un programme écrit Perl.

```
print "Veuillez entrer un nombre : ";
$nb = <STDIN>;
$nb2 = 2 * $nb;
chop($nb);
$chaine = "Le double de $nb est " . $nb2 . "\n";
print $chaine;
```

Dans ce programme, le contenu de `$nb` (ce qu'a saisi l'utilisateur) est une chaîne de caractères. Ce contenu peut être multiplié par 2 sans problème (conversion implicite d'une chaîne vers un nombre), le nombre résultant étant stocké dans la variable `$nb2`, et ce nombre peut être sans problème concaténé à une chaîne de caractères (conversion implicite d'un nombre vers une chaîne). En Python, des conversions explicites auraient été nécessaires dans les deux sens.

```
entree = input("Veuillez entrer un nombre : ")
nb = int(entree)
nb2 = 2 * nb
chaine = "Le double de " + str(nb) + " est " + str(nb2)
print(chaine)
```

8. https://fr.wikipedia.org/wiki/Rust_(langage).

Inconvénients

Un langage qui multiplie les conversions implicites présente cependant deux inconvénients majeurs. D'abord, on finit par ne plus trop savoir quelles conversions sont effectuées. Dans [1] par exemple, la description des conversions implicites du langage prend plus de 20 pages. Ensuite, on perd en sûreté, puisque le contrôle de type devient moins efficient.

Python est un langage qui autorise assez peu de conversions implicites, si ce n'est celles de base entre les types numériques. Malheureusement, en Python, le type `bool` est un type numérique et on a donc une conversion implicite des entiers et des réels vers les booléens. Cette conversion implicite, qui existait déjà en C, est à l'origine de nombreuses erreurs. De même, tout objet peut être interprété comme un booléen (`None` est converti en `False`, toute autre valeur en `True`), et cela peut également être source d'erreur.

Le cas particulier des nombres

Un langage n'offrant qu'un nombre restreint de conversions implicites, notamment entre les différents types numériques, peut vite s'avérer décourageant. Le cas suivant en Haskell, par exemple, laisse les débutants pantois.

```
Prelude> length [2, 3]
2
Prelude> 2 / 4
0.5
Prelude> (length [2, 3]) / 4
        No instance for (Fractional Int) arising from a use of '/'
        In the expression: (length [2, 3]) / 4
        In an equation for 'it': it = (length [2, 3]) / 4
```

En effet, comprendre que 2 et 2, ce n'est pas toujours la même chose, ce n'est pas tout à fait évident... Sur ce point, le choix de Python d'avoir deux opérateurs pour la division « à virgule » (/) et la division euclidienne (//) est plus judicieux que les langages similaires à C où on utilise le même opérateur, mais dont la signification dépend des types des opérandes.

Et la surcharge d'opérateur ?

Le fait qu'un langage autorise la surcharge d'opérateur peut de prime abord sembler très élégant. Si on définit une nouvelle classe `Matrice`, cela semble très naturel, étant donné deux instances $m1$ et $m2$ de cette classe de pouvoir écrire le produit matriciel $m1 \times m2$. Cependant, cela présente très vite l'inconvénient de ne plus savoir ce que l'on manipule. C'est déjà assez vrai en C++ [9], langage dont les variables sont typées, mais c'est encore plus vrai en Python, où les variables ne sont pas typées. Mais il est vrai qu'en Java, que l'on passe par une méthode d'instance demandant d'écrire `m1.prod(m2)`, ou par une méthode de classe demandant d'écrire `Matrice.prod(m1, m2)`, on obtient une écriture plus lourde et moins naturelle.

9. https://fr.wikipedia.org/wiki/C%2B%2B.

Sur ce point, une solution probablement plus élégante serait d'autoriser la surcharge d'opérateur moyennant l'implantation d'une interface spécifique. Mais je ne connais pas de langage fonctionnant ainsi.

Le choix du paradigme principal

On distingue en général trois paradigmes de programmation principaux, pouvant intégrer des paradigmes secondaires. Les trois paradigmes en question sont les paradigmes *impératif*, *fonctionnel* et *logique*. Quoique fort intéressant, le paradigme logique me paraît un peu trop spécifique pour être choisi comme « premier paradigme ». Les deux autres paradigmes présentent chacun leurs avantages et leurs inconvénients.

Le paradigme fonctionnel, en évitant la modification de données, permet d'éviter de nombreuses erreurs. Par ailleurs, il pourrait permettre de passer très rapidement de notions vues en mathématiques à un programme. Mais si ma phrase est au conditionnel, c'est parce que l'enseignement des mathématiques a été fortement remanié, et si, dans les années 90, la notion de fonction était étudiée dès le collège et était largement approfondie au lycée, ce n'est plus vraiment le cas aujourd'hui. Aussi, là où le paradigme fonctionnel pouvait être un bon choix de « premier paradigme » dans les années 2000, c'est moins le cas dorénavant. Il présente cependant un avantage de mettre à peu près tous les élèves à égalité — à savoir partir de zéro — si la découverte de la programmation est effectuée en primaire et en secondaire ; ce qui est moins le cas avec le paradigme impératif, que plusieurs élèves peuvent avoir déjà pratiqué.

Le paradigme impératif présente l'avantage non négligeable de se rapprocher de nombreuses situations de la vie courante (recettes de cuisine, notices de certains appareils, instructions d'un GPS...). De ce fait, c'est certainement, aujourd'hui, le paradigme à privilégier.

Le paradigme objet

Le paradigme *objet* est un paradigme secondaire. On peut ainsi trouver des langages fonctionnels orientés objet (OCaml [10]) et des langages impératifs orientés objet (Python par exemple).

Un langage pédagogique ne doit pas nécessairement être orienté objet. Cependant, certains concepts sont plus faciles à présenter sous la forme de classes et d'objets, et faire le choix d'un langage orienté objet peut donc être un plus non négligeable.

Cependant, un langage qui impose dès le départ le concept objet (comme Java ou Eiffel [11] par exemple), où tout code doit se situer à l'intérieur d'une classe, n'est pas forcément idéal pour débuter. Cela rajoute en effet dès le départ un concept de plus.

10. `https://fr.wikipedia.org/wiki/OCaml`.
11. `https://fr.wikipedia.org/wiki/Eiffel_(langage)`.

À ce titre, des langages comme Python ou Kotlin, dans lesquels du code peut être écrit en dehors de toute classe, est préférable.

Parmi les langages orientés objet, certains langages sont à base de prototypes (comme Javascript [12] par exemple), d'autres à base de classe (Java par exemple). Le système de prototype est plus souple, mais moins sûr et, à mon avis, moins clair. Aussi, les langages à base de classe sont, de mon point de vue, à privilégier.

Le premier intérêt du paradigme objet est d'apporter de la structuration et de la lisibilité dans le code. Il est fondamental que la structure d'une classe soit claire, et que la hiérarchie des classes soit transparente. Par conséquent, un langage pédagogique doit permettre de déclarer clairement les variables d'instance d'une classe, à l'instar de ce que l'on fait en Java, Eiffel, Kotlin, etc., et non « à la volée », par exemple au beau milieu du constructeur, comme cela est fait en Python. Expliquer une structure récursive et sa mise en œuvre est par exemple bien plus facile à faire en Java qu'en Python.

La hiérarchie des types doit également être claire. Par exemple, en Java, aussi bien la documentation que les méthodes d'introspection permettent de savoir qu'un tableau dynamique (`ArrayList`) est une liste (`List`) et qu'à ce titre, il est également itérable (`Iterable`). Cela est moins vrai dans un langage comme Python où, les notions de séquence et d'itérable ne correspondent qu'à des conventions. Ainsi, expliquer de manière un peu rigoureuse la boucle `for` en Python n'est possible qu'après une bonne pratique du langage, alors que la boucle `for` est souvent utilisée très tôt.

Un autre concept fort des langages à base de classe est la notion d'encapsulation. À partir du moment où un langage orienté objet est choisi comme langage pédagogique, il est indispensable que cette notion puisse être mise en œuvre de manière forte, comme cela est faisable en déclarant des attributs privés. Dans l'idéal, il serait même judicieux que l'encapsulation soit forcée, sans sacrifier à la lourdeur d'écriture. En Python, l'encapsulation n'est pas possible [13]. En Java, elle est possible, mais l'écriture reste lourde. En Kotlin, elle est possible avec une écriture légère.

Représentation de la structure de bloc et prise en compte de l'indentation

Les langages qui donnent une sémantique à l'indentation engendrent fréquemment des problèmes, surtout pour des débutants. Un premier problème vient du fait qu'il n'est pas évident de comprendre qu'une tabulation n'est pas équivalente à n espaces. Un autre problème lié à l'indentation est le risque d'erreur lors d'un copier-coller d'un niveau d'imbrication vers un autre. Toutefois, il est évident que lorsque sémantique et présentation s'accordent, le code est bien plus clair à lire. Mais pour ce faire, il vaut mieux un éditeur ou un IDE dédié qui assure l'indentation.

12. `https://fr.wikipedia.org/wiki/JavaScript`.

13. Le renommage automatique des variables d'instance dont le nom commence par « __ » ne constitue pas une réelle encapsulation.

Forcer la délimitation d'un bloc est plutôt un bon principe, que ce soit avec des accolades (comme en Perl) ou des mots-clefs comme `BEGIN`... `END` (comme en Pascal [14]). La possibilité offerte par les langages similaires à C de ne pas mettre de délimiteurs lorsqu'un bloc est restreint à une ligne est par contre fortement préjudiciable.

Verbosité ou concision ? Richesse ou pauvreté ?

Pendant quelques décennies, la mode a été aux langages à vocabulaire restreint. Cela a donné lieu à du code extrêmement verbeux, et à la nécessité de devoir rapidement maîtriser de nombreuses bibliothèques (c'est par exemple le cas de Java). Cela ne rend pas facile l'écriture de programmes un tant soit peu intéressants. La mode est en train de changer. Certains langages s'adaptent (Java s'enrichit avec une nouvelle boucle `for each`, des lambda-expressions, des enregistrements...), les nouveaux langages enrichissent leur vocabulaire (Kotlin est très riche en mots-clés par exemple) ou les types « natifs », comme Python, qui permet de manipuler directement des listes, des ensembles ou des dictionnaires. Ainsi, même si la bibliothèque de Java s'est fortement enrichie en méthodes facilitant l'utilisation des tableaux dynamiques, leur utilisation est bien plus aisée et lisible en Python.

Bibliothèques et documentation

Le fait qu'un langage dispose d'un ensemble de bibliothèques « officielles » riche est fondamental pour un langage pédagogique. En effet, c'est l'existence de telles bibliothèques qui va permettre de pouvoir développer rapidement des programmes intéressants. Le fait que ces bibliothèques soient plus ou moins officielles est également indispensable pour que les enseignants puissent échanger entre eux, pour que des ouvrages puissent être publiées... Il fut un temps, par exemple, où la classe `String` de C++ n'était pas standardisée ; inutile de dire comme cela compliquait les choses ! Sur ce point, des langages comme Python, Java ou Kotlin sont particulièrement bien dotés.

Un aspect fondamental pour un langage pédagogique est que celui-ci soit particulièrement bien documenté. À l'heure où, en informatique, quasiment plus personne n'achète de livres, souvent obsolètes avant même leur sortie, et où l'utilisation d'un moteur de recherche est devenue un réflexe primaire, il est indispensable qu'une documentation claire, structurée, bien indexée et centralisée soit disponible, afin d'éviter qu'une requête primaire dans un moteur de recherche mène vers une mauvaise réponse dans le premier forum venu. À ce titre, la documentation de PHP [15], par exemple, est un modèle de ce qu'il ne faut pas faire. La documentation de Python, par exemple, est bien mieux faite, même si elle présente de nombreux défauts. Par exemple, pour rassembler l'essentiel des informations

14. `https://fr.wikipedia.org/wiki/Pascal_(langage)`.
15. `https://fr.wikipedia.org/wiki/PHP`.

concernant les listes, il faudra se rendre sur au moins trois pages différentes : `https://docs.python.org/3/tutorial/introduction.html#lists`, `https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range`, `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`.

Par contre, Java est un langage dont je trouve la documentation exemplaire, aussi bien en ce qui concerne le langage en lui-même que pour la documentation des bibliothèques standard grâce à la Javadoc [16].

Consommation des ressources, portabilité, popularité

Certains critères, certes secondaires, peuvent malgré tout être pris en compte dans le choix d'un langage pédagogique.

Consommation des ressources

À l'heure du réchauffement climatique, la consommation des ressources doit être prise en compte. Cette consommation concerne aussi bien l'espace disque occupé par le programme que la mémoire consommée. Mais le critère prépondérant reste la consommation en énergie. Sur ce point, le choix de l'algorithme est déterminant, mais un langage « lent » consommera plus de CPU et, à ce titre, aura un impact carbone beaucoup plus important qu'un langage « rapide ». Les langages compilés, notamment sont à favoriser sur ce plan.

Portabilité

La portabilité est un critère important, au même titre que la standardisation des bibliothèques évoquée précédemment. C'est elle qui va permettre à différents enseignants travaillant sur des architectures différentes de travailler ensemble, et c'est elle également qui va permettre aux élèves motivés de valoriser sur leur propre machine ce qu'ils auront appris en classe.

Popularité

La popularité d'un langage ne garantit pas sa qualité. PHP est l'exemple type d'un langage mauvais bien que populaire. Toutefois, faire travailler les élèves sur un langage « niche » risque de décrédibiliser l'enseignement auprès des élèves.

16. Les classements de popularité des langages reposant entre autres sur le nombre de requêtes effectuées dans les moteurs de recherche et les messages postés dans certains forums, il n'est pas impossible que la qualité de la documentation de Java par rapport à celle de Python explique en partie le surclassement récent de Python par Java.

S'il fallait choisir...

Il est difficile d'étudier tous les langages tellement il en sort de nouveaux chaque mois. Mais je proposerais volontiers deux solutions qui me semblent pertinentes pour un bon langage pédagogique.

La première proposition n'est pas tout à fait un langage puisqu'il s'agit de la surcouche *Processing* [17] pour le langage Java. En effet, dans les parties précédentes, j'ai pu détailler les intérêts non négligeables de Java, mais également ses inconvénients, dont le passage indispensable par la notion de classe et de méthode de classe. Processing permet justement de lever cet inconvénient, sans nuire à tous les avantages du langage Java.

Une deuxième proposition est le langage Kotlin. Ce langage est complètement interopérable avec Java, ce qui signifie que toutes les bibliothèques standard Java sont accessibles. C'est par ailleurs un langage fortement typé, orienté objet et à base de classe, de plus en plus utilisé, notamment pour le développement des applications Android, portable... Il partage ainsi l'essentiel des avantages de Java. Mais il résout également un nombre important des défauts de Java : possibilité de passer outre la notion de classe, verbosité grandement réduite, sûreté accrue... Je pense qu'il s'agit d'un excellent candidat pour le choix d'un langage pédagogique.

Il ne faut pas non plus négliger les langages Rust [18] ou Ruby [19] par exemple qui peuvent également être de bons candidats à un langage pédagogique.

Références

[1] Benjamin J Evans, David Flanagan. Java in a Nutshell : A Desktop Quick Reference. *O'Reilly Media, Inc.*, Octobre 2014.

17. `https://fr.wikipedia.org/wiki/Processing`.
18. `https://fr.wikipedia.org/wiki/Rust_(langage)`.
19. `https://fr.wikipedia.org/wiki/Ruby`.