Séquence 2 - Algorithmes - Langages - Programmes - Processus

I. Algorithmes

I. 1. Définition

Knuth donne en 1968 une définition précise d'un algorithme : « *un ensemble fini de règles décrivant une série d'instructions destinées à résoudre un type de problème spécifique* ».
Un algorithme doit vérifier les cinq propriétés fondamentales suivantes :

Finitude : un algorithme doit toujours se terminer après un nombre fini d'étapes ;

Précision de la description : chaque étape d'un algorithme doit être définie précisément, les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas ;

Présence d'entrées : zéro ou plusieurs quantités sont données à l'algorithme avant qu'il ne commence. Ces entrées sont prises dans des classes d'objets spécifiées ;

Création de sorties : l'algorithme renvoie des quantités ayant des relations spécifiées avec les entrées ;

Effectivité : « *toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées en une durée finie, par un homme utilisant un papier et un crayon.* ».

I. 2. Comment représenter et décrire le fonctionnement de mon algorithme ?

En général, avant de se lancer dans la programmation d'un algorithme sur un ordinateur, **on travaille d'abord sur papier**.

I. 2. a. Diagramme entrée-sortie.

Avant toute chose, il faut s'atteler à décrire précisément le but de l'algorithme et notamment :

ce qu'il prend en entrée : description (nom, type) des informations **indispensables** à fournir en entrée de l'algorithme pour effectuer le travail demandé

ce qu'il doit renvoyer en sortie : description (nom, type) des résultats attendus en sortie de l'algorithme

On utilise en général un **diagramme entrée-sortie**. La Figure 1 donne le diagramme entrée-sortie de l'algorithme d'Euclide, permettant de calculer le PGCD d de deux nombres m et n entiers.

I. 2. b. Écriture en pseudo-code.

Knuth explique très précisément comment écrire des algorithmes destinés à être exécutés par une machine et préconise un formalisme très riche et détaillé permettant de lever toute ambiguïté, en accord avec les propriétés de précision et d'effectivité. La Figure **??** détaille

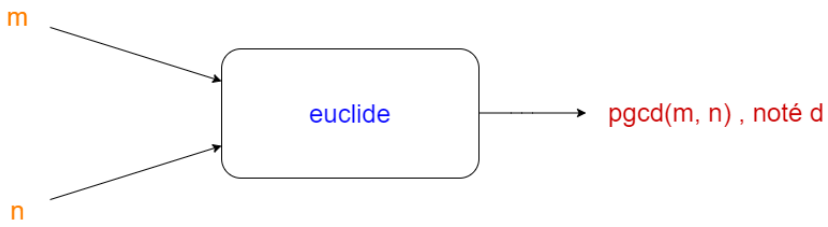

FIGURE 1 – Diagramme entrée-sortie de l'algorithme d'Euclide.

Algorithme 1 : `euclide`

Donnée : `m`, `entier`
Donnée : `n`, `entier`, $m > n$

Variable de travail : `c`, `entier`
Variable de travail : `d`, `entier`
Variable de travail : `r`, `entier`

1 **Stocker** la valeur de `m` dans `c` — `c` ← `m`
2 **Stocker** la valeur de `n` dans `d` — `d` ← `n`

3 **Stocker** le reste de la division euclidienne de `c` par `d` dans `r` — `r` ← `c`% `d`

4 **Tant que** *la valeur contenue dans* `r` *est différente de 0* **faire**
5 **Stocker** la valeur de `d` dans `c` — `c` ← `d`
6 **Stocker** la valeur de `r` dans `d` — `d` ← `r`
7 **Stocker** le reste de la division euclidienne de `c` par `d` dans `r` — `r` ← `c`% `d`

8 **Renvoyer en sortie de l'algorithme** la valeur contenue dans la zone de stockage `d`

FIGURE 2 – Algorithme d'Euclide écrit en pseudo-code : il permet de calculer le plus grand commun diviseur PGCD de deux nombres.

l'algorithme d'Euclide en pseudo-langage dans un formalisme proche de celui suggéré par Knuth.
On dit que l'on écrit l'algorithme en **pseudo-code** informatique. Cela consiste à écrire, dans un langage humain (français, anglais...) compréhensible mais très basique, le détail des instructions qui doivent être exécutées pour mener à bien l'algorithme.

I. 2. c. Diagramme de flux.

De temps à autre, il peut être utile de représenter un algorithme sous la forme d'un **diagramme de flux**, ce qui permet de mieux repérer les **branchements** (**boucles** et **instructions conditionnelles**)
La Figure 3 montre l'algorithme d'Euclide sous la forme d'un diagramme de flux.

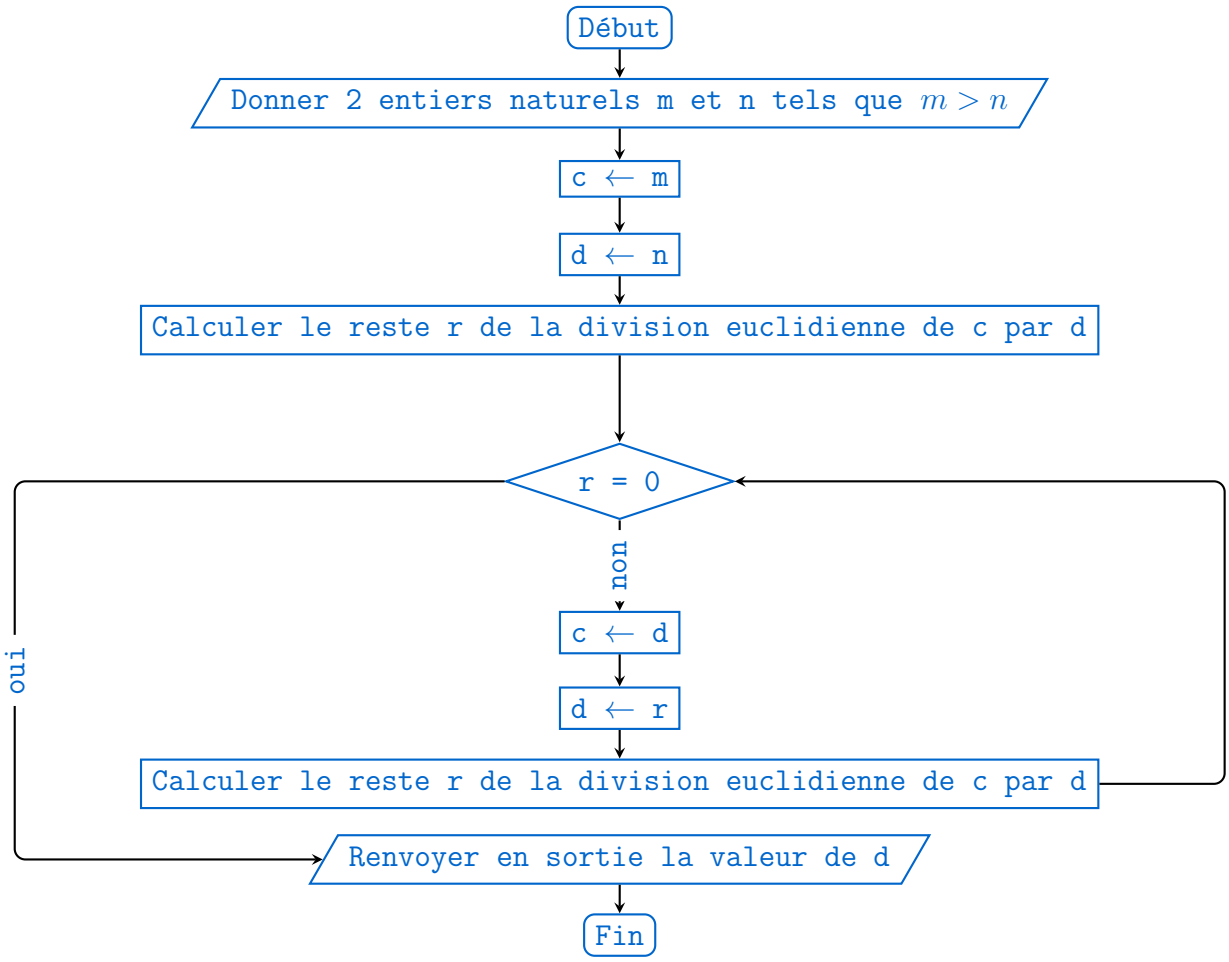

FIGURE 3 – L'algorithme d'Euclide simple représenté par un diagramme de flux.

I. 2. d. Granularité.

Lorsque l'on représente un algorithme, on en vient forcément à se demander jusqu'à quel degré de détail cette représentation doit être poussée. Par exemple, dans une recette de cuisine, je peux simplement donner l'instruction :

Casser un œuf dans un saladier

ou alors expliquer en détail comment procéder pour casser un œuf dans un saladier :

Approcher l'œuf près du bord du saladier
Frapper le d'un coup sec contre le rebord du saladier
Finir d'ouvrir l'œuf brisé avec vos doigts
Verser le contenu resté dans l'œuf dans le saladier

On parle parfois de **granularité** de la description : plus la description est détaillée, plus la granularité sera fine.
Le choix du niveau de détail de la description dépend essentiellement de l'objectif final : s'il s'agit simplement d'expliquer un algorithme à une autre personne, un niveau assez peu détaillé sera suffisant car un autre esprit humain saura généralement interpréter correctement tous les détails sous-entendus, comme dans l'exemple de l'œuf cassé.
Si l'on souhaite in fine faire exécuter l'algorithme par une machine, le choix de granularité dépendra essentiellement du degré d'évolution (plus ou moins haut niveau) du langage de programmation qui sera utilisé pour coder l'algorithme. Plus le langage sera bas niveau, plus les instructions devront être détaillées. Le niveau de détail dépend également de l'utilisation de références à d'autres algorithmes supposés connus. Par exemple, de nombreuses instructions se cachent derrière l'instruction

Prendre la racine carrée de ce nombre

Si l'on suppose l'algorithme de calcul de la racine carré d'un nombre déjà connu, nul besoin de détailler. Sinon, il faudra détailler cette instruction pour rendre l'algorithme suffisamment précis au sens de Knuth.

I. 3. Comment architecturer mon algorithme ?

Architecturer un algorithme revient à s'interroger sur son organisation, que cela soit au niveau des

instructions : comment organiser de façon lisible et cohérente la série d'instructions ?

structures de données : comment organiser les informations (données d'entrées, variables de travail) manipulées ?

I. 3. a. Organisation des instructions : programmation modulaire

La **programmation modulaire** consiste à regrouper des instructions associées à une même tâche dans des sous-algorithmes distincts. Ces sous-algorithmes permettent de partitionner la masse d'instructions en blocs et sous-blocs logiques relativement indépendants appelés **fonctions** ou **procédures**. Parfois, des fonctions ayant un thème ou un objectif commun sont regroupées dans des bibliothèques. Le fait de regrouper des instructions associées au même travail dans des sous-algorithmes autonomes permet d'améliorer :

la lisibilité du code : le découpage des différentes tâches est plus clair, la compréhension plus aisée. Le code est **factorisé**, c'est-à-dire que si une même tache est réalisée plusieurs fois, elle n'est codée qu'une seule fois sous la forme d'un sous-algorithme (d'une fonction) et appelée autant de fois que nécessaire sans recopier toute la série d'instructions correspondantes. Cela permet donc de réduire la taille du code ;

le test et le débogage du code : la factorisation du code limite l'introduction de bugs et facilite la mise en place de **tests unitaires** ;

l'évolution du code : il est plus aisé de faire évoluer le code, le travail collaboratif est facilité car les parties du code qui devront être modifiées peuvent être plus clairement déterminées en amont. Il est possible de réutiliser du travail effectué par d'autres dans des bibliothèques partagées ;

le temps de compilation : le découpage du code en modules séparés dans des fichiers séparés permet de réduire le temps de compilation.

I. 3. b. Organisation des données : structures de données

Le choix des structures de données constitue l'autre partie du travail d'architecture, peut-être le plus déterminant. Il s'agit de choisir une **représentation logique voire sémantique des données** permettant d'abord une efficacité d'exécution optimale sur machine, mais aussi une maintenabilité et une lisibilité accrue du code. Nous reviendrons très largement sur les structures de données et les enjeux associés au cours de l'année.
Plusieurs solutions d'architecture sont généralement possibles, mais certaines seront plus optimales que d'autres. Les choix d'architecture des algorithmes sont les clés de sa pérennité et de son efficacité une fois codés et exécutés par une machine.

La conception même des différents langages de programmation s'appuie sur des paradigmes architecturaux. Par exemple, les langages orientés objets incitent à adopter une architecture privilégiant la réflexion sur les structures de données, les langages fonctionnels favorisent la réflexion sur le découpage en sous-algorithmes et la limitation des **effets de bords**.

II. Langages de programmation et traductions successives

Définition 1 (Programmer)

Programmer, c'est traduire un algorithme dans un langage qui peut être rendu compréhensible par un ordinateur pour qu'il puisse l'exécuter.

II. 1. Différents niveaux de langage d'une machine programmable

Cette activité de traduction peut s'effectuer à différents niveaux plus ou moins proches de l'architecture physique de la machine. On distingue 3 niveaux :

Langage machine. En langage machine, les instructions sont écrites comme séries de 0 et de 1. Les instructions écrites en langage machine sont totalement dépendantes

de l'architecture de la machine et de son processeur car elles font appel au **jeu d'instructions** élémentaires mis à disposition par le concepteur du processeur. Les 8 premiers bits d'une instruction machine encodent le type d'opération, les suivants servent à désigner les opérandes, c'est à dire les données nécessaires à la réalisation de l'instruction (mots mémoires, registres du processeur ou valeurs immédiates). Ce niveau de programmation est extrêmement pénible pour un être humain et très peu pratiqué.

Langage d'assemblage (bas niveau). Une première amélioration a consisté à substituer aux chaînes binaires représentant des codes opérations ou des adresses mémoires, des chaînes de caractères plus aisément manipulables par l'être humain que l'on appelle des mnémoniques : c'est le langage d'assemblage qui constitue une variante symbolique du langage machine, permettant au programmeur de manipuler les instructions de la machine en s'affranchissant notamment des codes binaires et des calculs d'adresse. Le langage d'assemblage comporte le même jeu d'instructions que le langage machine et est également spécifique de la machine. Pour pouvoir exécuter un programme écrit en langage d'assemblage, il faut donc traduire les instructions de celui-ci vers les instructions machine correspondantes. Cette phase de traduction est réalisée par un outil appelé l'**assembleur**.

Langage de programmation (haut niveau). Ces langages se caractérisent principalement par le fait qu'ils sont, contrairement aux deux autres types de langage que nous venons d'aborder, totalement indépendants de l'architecture de la machine et du processeur. Par ailleurs, ils offrent un pouvoir d'expression plus riche et plus proche de la pensée humaine, rendant ainsi plus aisée la traduction des algorithmes établis pour résoudre un problème. Ces langages de fait sont davantage définis par rapport aux besoins d'expression du programmeur que par rapport aux mécanismes sous-jacents de la machine physique. Ce sont ces langages qu'utilisent la très grande majorité des programmeurs. Bien sûr, des outils permettent de traduire les codes écrits dans ces langages en langage machine, cf Section **??**

II. 2. Chaîne de production d'un programme exécutable sur une machine

Pour transformer un algorithme écrit en pseudo-code en une série d'instructions exécutables par une machine, deux traductions successives ont donc lieu.

1ère traduction. L'algorithme écrit sur papier en pseudo-code est traduit dans un langage de programmation [1] (par exemple Python). Cette première traduction est généralement effectuée par un humain (l'élève, le programmeur) : l'algorithme traduit en langage de programmation est écrit sur l'ordinateur dans un fichier texte simple. Le fichier texte de la traduction est appelé **code source**. Le nom de ce fichier possède une extension qui indique le langage de programmation utilisé (`.py` pour un code écrit en Python par exemple).

Pour écrire le code source, un simple éditeur de texte (par exemple `emacs`) suffit, mais il est également possible d'utiliser un **Environnement de Développement Intégré** (EDI) [2] qui facilite l'écriture du code source grâce à de nombreuses fonctionnalités additionnelles (coloration syntaxique, complétion automatique, recherche de

1. Nous détaillons cette notion en Section III.
2. En anglais, l'acronyme est IDE pour Integration Development Environment

bibliothèques, analyse syntaxique, indentation automatique, compilation et débogage intégré, logiciel de gestion de version intégré...etc).

2ème traduction. Le code source est ensuite traduit en langage machine binaire lors de l'étape de compilation ou d'interprétation que nous détaillons en Section II. 4.. Cette étape complexe est réalisée par la machine grâce à un logiciel dédié appelé compilateur ou interpréteur. Chaque langage de programmation doit proposer un compilateur ou un interpréteur permettant sa traduction en langage machine.

Le code source constitue ainsi une traduction intermédiaire entre le pseudo-code en langage humain et le langage machine. Un **programme** est une **implémentation** d'un algorithme dans un **langage de programmation**. Il s'agit ni plus ni moins que de la traduction de l'algorithme dans une langage formel qui, après **une deuxième traduction**, est compréhensible par la machine.

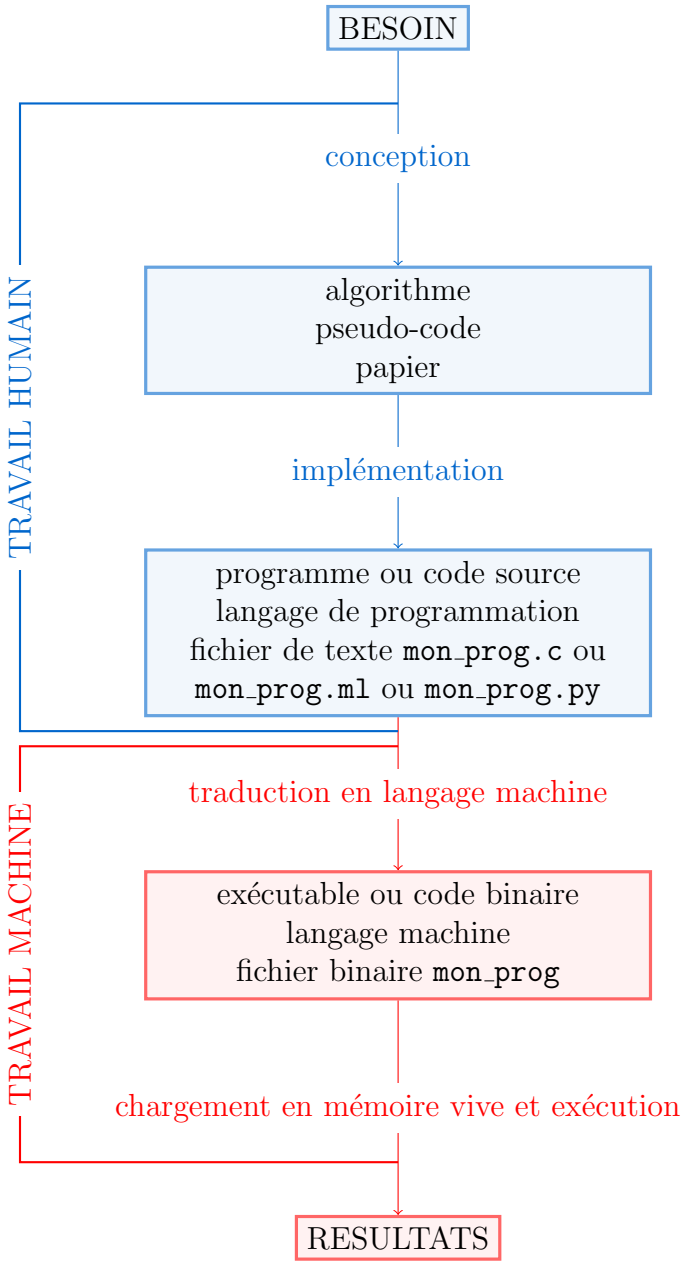

FIGURE 4 – Chaîne de conception d'un algorithme jusqu'à son exécution sur une machine programmable.

II. 3. 1ère traduction : bonnes pratiques pour l'écriture de programmes

La traduction d'algorithmes dans un langage de programmation est l'activité centrale du programmeur. Plusieurs choix doivent être effectués avant de se lancer dans cette première traduction.

Choix du langage de programmation : c'est un choix crucial, dont nous détaillons les enjeux en Section III.. Cette année, deux nouveaux langages seront abordés, le **langage C** et langage **OCaml**. Nous continuerons également à coder occasionnellement en Python.

Choix des outils de programmation : c'est un choix secondaire mais qui peut être très impactant, en particulier si plusieurs programmeurs travaillent en même temps sur le même code (travail collaboratif), entre autres :

- choix d'un éditeur de texte simple (`vi`, `BlocNote`), d'un éditeur de texte augmenté prenant en charge le langage de programmation choisi (`emacs`) ou d'un EDI sophistiqué (`VSCode`, `VisualStudio`, `Eclipse`, `EduPython`, `Anaconda`...)
- mise en place d'un logiciel de gestion de version de code (Git, CVS, Perforce) [3]
- mise en place d'une plateforme d'intégration continue (tests automatiques) [4]
- mise en place d'outils de gestion de projet : planification des tâches, plateforme de suivi de bugs [5]
- utilisation d'un logiciel de génération automatique de documentation (Doxygen)...etc

Étant donnée la relative simplicité des programmes que vous réaliserez cette année et du fait que vous travaillerez majoritairement seul, nous nous contenterons d'utiliser l'éditeur de texte augmenté `emacs`. Lors des TIPE, il pourra être pertinent, dans le cadre d'un travail collaboratif, d'utiliser le logiciel de gestion de version GIT.

Une fois ces choix effectués, l'écriture du code source proprement dite requiert de la rigueur et de bonnes pratiques :

- Toujours faire une analyse sur **papier** avant de se lancer dans l'écriture du code source : diagramme entrée-sortie, écriture en pseudo-code, choix d'architecture (instructions et données), analyse de terminaison, correction, complexité...
- Choisir des **noms pertinents** pour les différents objets (variables, fonctions, déclarations), adopter des conventions de nommage cohérentes
- Systématiquement **commenter** le code. Indiquer avant chaque fonction sous forme de commentaire : son objectif, chacune de ses données d'entrée (nom, sens, type), chacune de ses sorties....
- Respecter les **indentations** du langage
- Procéder par **petits pas**. Coder une première fonctionnalité, la compiler, la tester et la déboguer de façon indépendante. Puis passer à la suivantes...etc

Le processus de conception d'un programme informatique est fondamentalement itératif. Il est fortement déconseillé d'écrire des dizaines de lignes de code d'affilée sans effectuer de test ou de compilation. Grâce à l'analyse sur papier et à la réflexion en amont sur l'architecture du code, notamment la réflexion sur la modularité et les structures de données, le

3. `https://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions`
4. `https://fr.wikipedia.org/wiki/Int%C3%A9gration_continue`
5. `https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_suivi_des_bugs`

travail de programmation peut généralement être découpé en petites tâches relativement autonomes. Chaque tâche est codée, testée et déboguée, d'abord de façon relativement indépendante puis les différents codes sont progressivement assemblés, en testant et en déboguant à chaque nouvel assemblage. C'est la méthode des **petits pas**, qui est centrale dans les méthodes de développement Agile [6].

II. 4. Deuxième traduction : traduction en langage machine

Nous détaillons ici le processus permettant la création d'un programme exécutable prêt à être chargé en mémoire vive à partir d'un programme source écrit dans un langage de haut niveau, dans un ou plus fichiers texte de code source. Ce processus se décompose en plusieurs étapes détaillées en Figure 5 :

Prétraitement (*preprocessing*) : le fichier de code source est traité par un préprocesseur qui fait des transformations purement textuelles en complétant le fichier (remplacement de chaînes de caractères, inclusion d'autres fichiers source) ou en le nettoyant (effacement des commentaires, des caractères blancs inutiles...etc).

Compilation : le fichier engendré par le préprocesseur est traduit en langage d'assemblage

Assemblage : le fichier écrit en langage d'assemblage est traduit en langage machine dans un fichier appelé fichier objet, encodé en format binaire ELF (*Editable and Linkable Format*) [7]

Édition de liens : dans le cas où le code utilise plusieurs fichiers, ou bien si des bibliothèques sont utilisées, les liens sont faits pour retrouver les codes correspondant à toutes les fonctions appelées. L'édition de liens produit un fichier exécutable.

Une fois le fichier binaire exécutable généré, à chaque exécution du programme, le fichier binaire exécutable est chargé en mémoire par le **chargeur** (*loader*) qui va notamment recalculer (translater) toutes les adresses mémoires apparaissant dans le programme pour prendre en compte le décalage lié au positionnement du fichier binaire exécutable à l'endroit choisi dans la mémoire vive.

II. 4. a. Compilation

Cette étape permet la traduction du code source nettoyé et complété par le pré-processeur et écrit dans un langage de haut niveau (C, OCaml, Python) en un programme écrit en langage d'assemblage bas niveau propre à l'architecture sur laquelle le code source est compilé. Le programme en assembleur obtenu peut également également être stocké sur le disque dur de l'ordinateur, généralement de manière temporaire.
Le compilateur est un logiciel dépendant de la machine physique vers laquelle il doit traduire le langage de haut niveau. Ainsi, un programme compilé sur une machine donnée ne s'exécutera pas forcément sur une autre machine, notamment si cette autre machine fonctionne avec un microprocesseur possédant une autre architecture ou un système d'exploitation différent.
Le travail du compilateur se divise en plusieurs phases :

l'analyse lexicale : il s'agit de reconnaître dans le fichier texte les mots du langage (vocabulaire) ;

6. https://fr.wikipedia.org/wiki/M%C3%A9thode_agile
7. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

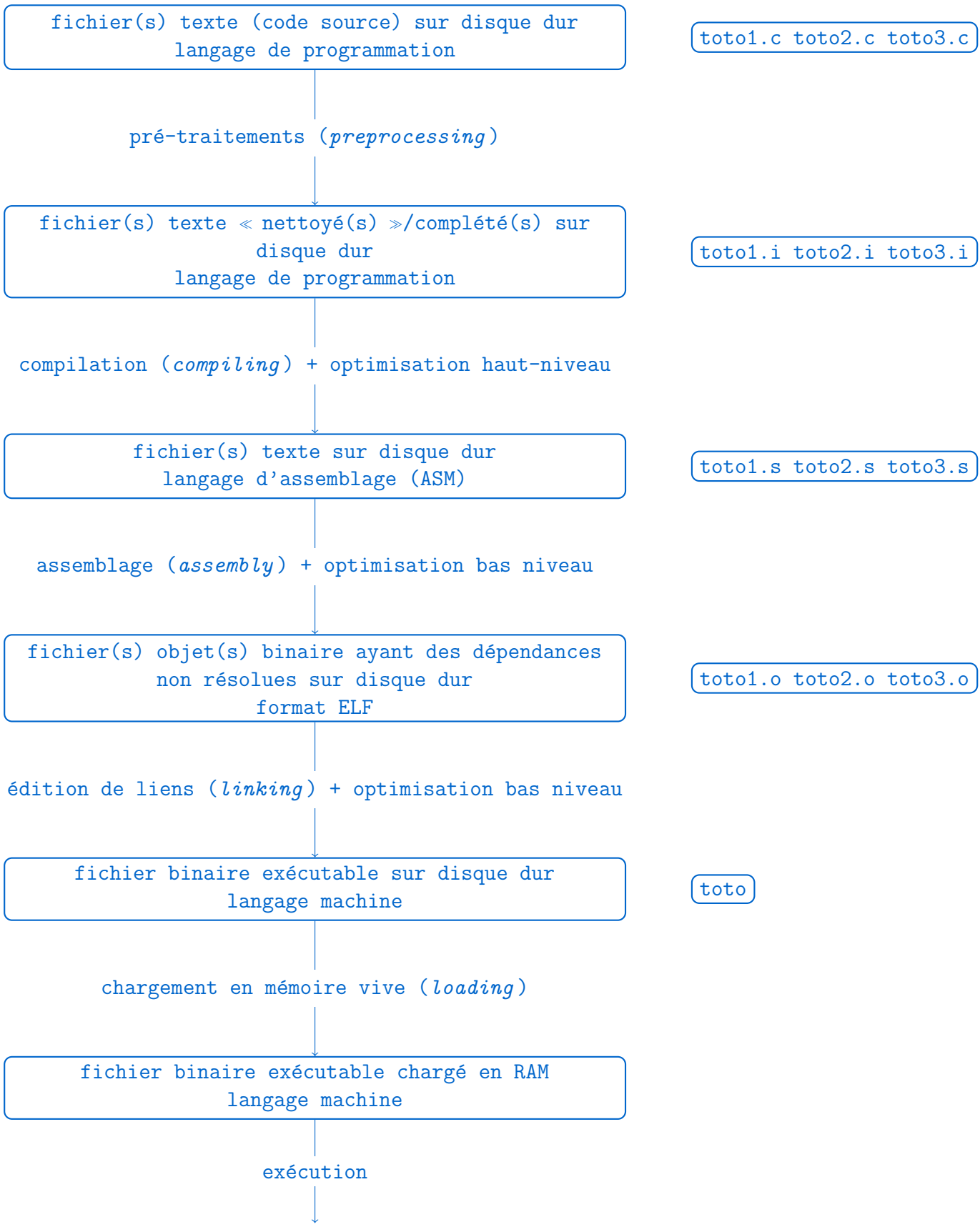

FIGURE 5 – Détail de la deuxième traduction du code source en langage machine

l'analyse syntaxique : il s'agit d'analyser la syntaxe des instructions écrites dans le fichier texte (grammaire, cf programme de 2ème année) ;

l'analyse sémantique : il s'agit de vérifier la sémantique, c'est à dire le sens des instructions et leur enchaînement logique ;

optimisation haut-niveau : le code est analysé pour éviter les instructions répétées inutilement par exemple, où éviter des allocations mémoires inutiles ;

génération du code en langage d'assemblage : les instructions sont transformées en langage d'assemblage après une phase d'optimisation haut-niveau permettant d'améliorer le temps d'exécution.

II. 4. b. Assemblage

Cette phase transforme le programme écrit en langage d'assemblage vers un programme dit **programme objet** qui est écrit en langage machine au format ELF [8]. Généralement, on donne l'extension `.o` aux fichiers objets.
Durant cette phase, tout ce qui peut être traduit en langage machine est traduit, sauf les références non résolues (appel à des fonctions ou des variables inconnues à l'échelle du fichier par exemple). Durant cette étape, un travail d'optimisation bas niveau est généralement effectué afin d'améliorer encore plus finement l'efficacité de l'exécution au niveau machine (parallélisme d'instructions par exemple)

II. 4. c. Édition de liens

Dans le cas d'une programmation modulaire (codage des différents sous-algorithmes dans des fichiers séparés, appel à des bibliothèques externes...), chaque fichier source est compilé et plusieurs fichiers objets sont générés : un fichier objet par fichier de code source.
Il faut alors **résoudre les dépendances**, c'est-à-dire relier les différents fichiers objets pour que, par exemple l'appel d'une fonction dans l'un des fichiers objets (lien à satisfaire) soit mis en relation avec la fonction en question présente dans un autre fichier objet (lien utilisable). Une fois ces liens repérés, il faut mettre à jour toutes les adresses mémoires impliquées dans les instructions pour pouvoir générer un programme exécutable cohérent qui pourra être chargé en mémoire de travail avec des variables et des données ayant les bonnes adresses mémoire.

II. 4. d. Compilateur ou interpréteur ?

En réalité, il existe des variantes dans le processus de traduction en langage machine. Dans certains cas, les créateurs d'un langage de programmation peuvent choisir de mettre à disposition un interpréteur à la place ou en plus d'un compilateur.
Dans ce cas, au moment de l'exécution, le programme à exécuter n'a pas été entièrement et/ou définitivement traduit en langage machine. La traduction en langage machine des instructions est effectuée au moins en partie au moment même de l'exécution par l'interpréteur. L'exécution nécessite ainsi de disposer non seulement du programme, mais aussi de l'interpréteur correspondant. [9]

Avantages. L'avantage principal des interpréteurs est la **portabilité** des programmes : le concepteur du programme ne transmet plus directement le programme exécutable à l'utilisateur, il n'y a donc plus besoin de se soucier de l'architecture de microprocesseur auquel est destiné ce programme exécutable. N'importe quelle machine sera en mesure

8. `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
9. Attention, on appelle parfois **machine virtuelle** ces interpréteurs, mais cela n'a pas grand chose à voir avec les machines virtuelles émulant un système d'exploitation, comme Virtual Box.

d'exécuter un code écrit dans ce langage de programmation du moment qu'un interpréteur de ce langage y est installé. Ainsi, un programme unique pourra être exécuté sur des machines variées ayant des configurations matérielles différentes sans aucun changement, pourvu qu'il existe un interpréteur spécifique à chacune de ces configurations.
L'interpréteur, en tant que couche intermédiaire entre le code à exécuter et la machine, permet aussi de prendre en charge une partie de l'effort de programmation dévolu au programmeur (type des données, contrôle de la mémoire, gestion des erreurs).
L'interpréteur permet enfin d'avoir un meilleur contrôle de l'exécution des programmes et facilite le travail d'analyse et de débogage.
Inconvénient. L'inconvénient est une plus grande lenteur d'exécution : en plus de l'exécution proprement dite des instructions, l'interpréteur analyse le code et en retraduit certaines parties, créant un surcoût (**overhead**) en mémoire et nombre d'instructions à effectuer qui ralentit généralement l'exécution du code.
En pratique, il existe une continuité entre interpréteurs et compilateurs. Un interpréteur "pur" lit chaque instruction et la transforme en langage machine à la volée, un compilateur "pur" traduit l'intégralité du code en langage machine avant l'exécution. La plupart des interpréteurs actuels se situent entre ces deux extrêmes avec :

— des phases d'analyse et de compilation dans un langage intermédiaire généralement appelé **bytecode**

— des phase d'exécution durant laquelle le *bytecode* est traduit en langage machine par ce que l'on appelle le *runtime*, le *runtime* proposant aussi parfois des fonctionnalités de contrôle de l'exécution (ramasse-miette, gestion des erreurs, introspection [10]...etc) et d'aide au développement (débogage, analyse de performances...etc)

II. 5. Bonnes pratiques de programmation

Erreurs de compilation. Si des problèmes apparaissent au moment de la compilation, il convient d'abord de corriger ces **erreurs de compilation** en corrigeant les fichiers sources incriminés. La phase de compilation peut ne révéler aucune erreur mais toutefois avertir de potentiels problèmes. Ces **avertissements** de compilation doivent faire l'objet d'une attention particulière.

Erreurs de programmation. Si la compilation se fait sans problèmes ni avertissements, le fichier binaire obtenu est exécutable sur la machine hôte. Des **erreurs d'exécution** peuvent apparaître. Souvent plus délicates à corriger que les erreurs de compilation, certaines d'entre elles sont *fatales* et mènent à l'arrêt prématuré du programme. C'est le cas d'opérations illégales comme la division par 0 ou l'accès à certaines parties non autorisées de la mémoire.
D'autres erreurs *non fatales* sont plus difficiles à corriger car leur cause peut avoir des conséquences bien après le début de l'exécution du programme ! C'est pourquoi des tests les plus exhaustifs possibles mais également des procédures de validation, voire des preuves formelles de la pertinence d'un programme sont nécessaires. Encore faut-il pouvoir les mettre en place. Et encore faut-il également que le compilateur lui-même ne soit pas la source de nouvelles erreurs !
Erreurs de conception. Ce sont les plus difficiles car elles touchent aux aspects théoriques : mauvais algorithme, cas particuliers mal gérés...etc
Là encore, l'important est de procéder itérativement : je code une fonctionnalité, je la compile, je la teste de façon exhaustive et je passe à la fonctionnalité suivante quand tout est validé, comme détaillé en Figure 6.

10. https ://fr.wikipedia.org/wiki/R%C3%A9flexion_(informatique)

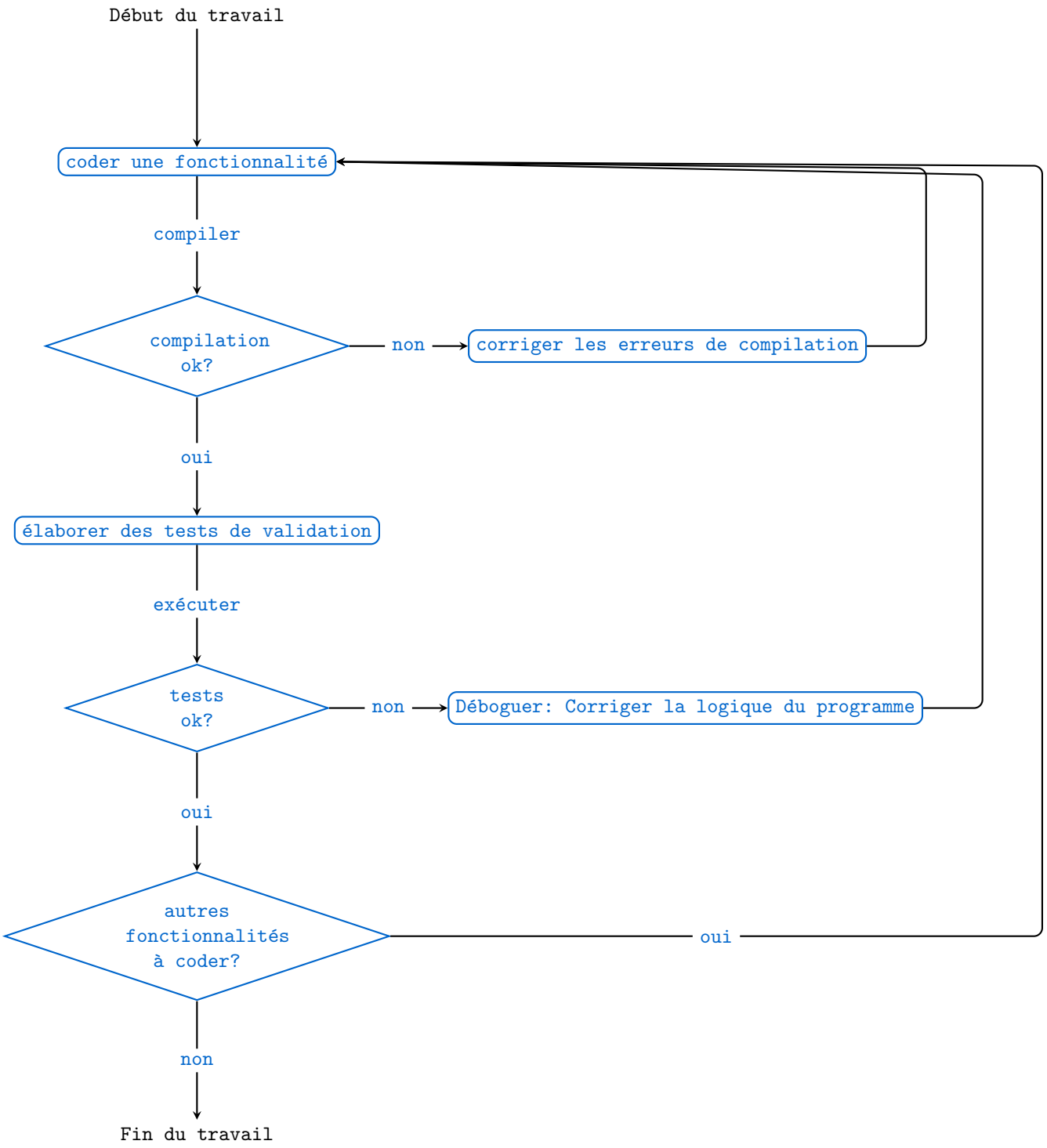

FIGURE 6 – Bonnes pratiques de développement d'un logiciel.

III. Paradigmes et langages de programmation

III. 1. Paradigmes de programmation

Définition 2 (Paradigme de programmation)

Un paradigme de programmation est une « philosophie » de programmation. Il désigne la manière dont le programmeur pense la conception de son code et quelle logique sous-tend son architecture et son implémentation.

III. 1. a. Paradigmes de programmation impérative

Lorsque l'on suit le paradigme de programmation impérative, on cherche surtout à répondre à la question *Comment ?* en listant les instructions à suivre l'une après l'autre. Les instructions sont donc décrites de manière séquentielle, comme dans une recette de cuisine. L'algorithme est perçu comme une suite d'états modifiés par des instructions.
C'est le paradigme principal qui a naturellement orienté la conception des premiers langages de programmation haut niveau car il correspond à ce qui se passe réellement dans une machine à architecture de Von Neumann.
Le paradigme de programmation impérative consiste à décrire un programme sous la forme de séquences d'instructions à effectuer dans un certain ordre pour résoudre un problème ou pour effectuer une action. Cette série d'instructions affecte l'état du programme (ses variables, ses fonctions) ou de l'ordinateur (ses fichiers, ses programmes, ses périphériques).
Parmi les caractéristiques de la programmation impérative, on distingue :

L'importance de l'ordre des instructions : elles sont exécutées l'une après l'autre et modifient l'état des mémoires (registres, mémoire principale...) de l'ordinateur.

L'usage d'instructions d'affectation. Ce type d'instruction demande à la machine de stocker une valeur dans une variable. Nous la noterons avec une flèche vers la gauche en pseudo-code. Par exemple, $a \leftarrow 5$ signifie « stocker la valeur 5 dans la variable a »

L'utilisation de boucles. Ce type d'instruction indique la répétition en boucle d'un bloc d'instructions sous certaines conditions. Il existe deux types de boucles, qui nous écrirons comme ceci en pseudo-code :

```
1 Tant que test faire
2     Bloc d'instructions à exécuter tant que le test reste vérifié
```

```
1 Pour un compteur évoluant de .... à .... par pas de ..... faire
2     Bloc d'instructions à exécuter tant que le compteur reste dans son intervalle
```

Les instructions conditionnelles permettent d'indiquer qu' une série d'instructions ne doit être effectuée que si certaines conditions sont vérifiées. La syntaxe en pseudo-code est :

```
1 Si test alors
2 |  Bloc d'instructions exécutées si le test est vérifié
3 Sinon
4 |  Bloc d'instructions exécutées si le test n'est pas vérifié
```

Elles sont autorisées dans le paradigme impératif mais aussi dans le paradigme fonctionnel.

III. 1. b. Paradigmes de programmation fonctionnelle

Dans ce type de programmation, tout algorithme est pensé en termes de descriptions et d'appels imbriqués de fonctions au sens mathématique, ces fonctions étant considérées comme **pure**.
Une **fonction est pure** si elle possède deux propriétés : respecter la **transparence référentielle** et ne pas créer d'**effets de bord**.
Transparence référentielle. La transparence référentielle est le principe selon lequel le résultat d'un programme est toujours le même si on remplace une expression par une autre expression de même valeur. Par exemple, dans les équations suivantes, a et b valent tous les deux 22 car il n'y a pas de différence entre $f(3,\,5)$ et la valeur 8 :

$$f(x,\,y) = x + y \quad a = f(3,\,5) + f(f(3,\,5),\,6) \quad b = 8 + f(8,6)$$

Cela signifie en particulier que deux appels de fonctions avec les mêmes arguments d'entrée doivent toujours retourner le même résultat. Cette propriété n'est pas forcément respectée dans les fonctions de la programmation impérative.
Effets de bord. On dit qu'une fonction crée un effet de bord lorsqu'elle génère une modification d'un élément qui lui est extérieur lors de son calcul.
Afin d'éviter les effets de bord, la programmation fonctionnelle pure **décourage certains aspects presque incontournables de la programmation impérative, comme l'affectation** d'une valeur à une variable. Cela signifie par extension que **l'usage de boucles est découragé** car elles-mêmes nécessitent d'affecter des valeurs à des variables de comptage de boucle.
Pour répéter des traitements comme le ferait une boucle, la programmation fonctionnelle utilise des fonctions récursives.
Les contraintes d'une fonction pure ont également pour conséquence qu'**une fonction pure doit toujours retourner une valeur**. En effet, une fonction n'ayant pas définition aucun effet de bord et produisant toujours le même résultat avec les mêmes données d'entrée, elle n'aurait aucun usage si elle ne renvoyait pas de valeur.
C'est le paradigme principal adopté pour la conception du langage OCaml. Il ne s'agit pas seulement de passer par des fonctions pour gérer les opérations, mais de percevoir des les fonctions elles-mêmes comme des objets pouvant être passés en argument d'autres fonctions.

- Il n'a a pas d'affectation de variables, seulement des déclarations de valeurs avec des étiquettes associées
- Il n'y a pas de boucles
- Tout algorithme est un enchaînement de transformations par des fonctions

Le principal avantage de ce paradigme est de limiter intrinsèquement les **effets de bord** : une fonction ne peut modifier que les données internes auxquelles elle a accès. L'apprentissage est par contre plus complexe et requiert un niveau d'abstraction élevé.

III. 1. c. Paradigme de programmation orientée objet (POO)

Ce paradigme de programmation, qui peut se combiner aux deux précédents, pense prioritairement les algorithmes à travers les données qu'ils manipulent. L'effort de conception est centré sur l'architecture des données. Ce paradigme s'attache à définir des classes d'objets ayant des caractéristiques (appelées attributs) et auxquels sont associées des actions (appelées méthodes). Une fois ces classes d'objets définies, on peut créer différents objets de cette classe. On dit que l'on crée des **instances** de cette classe. La Figure 7 montre un exemple de classe `Voiture` et l'instanciation d'un objet de cette classe.

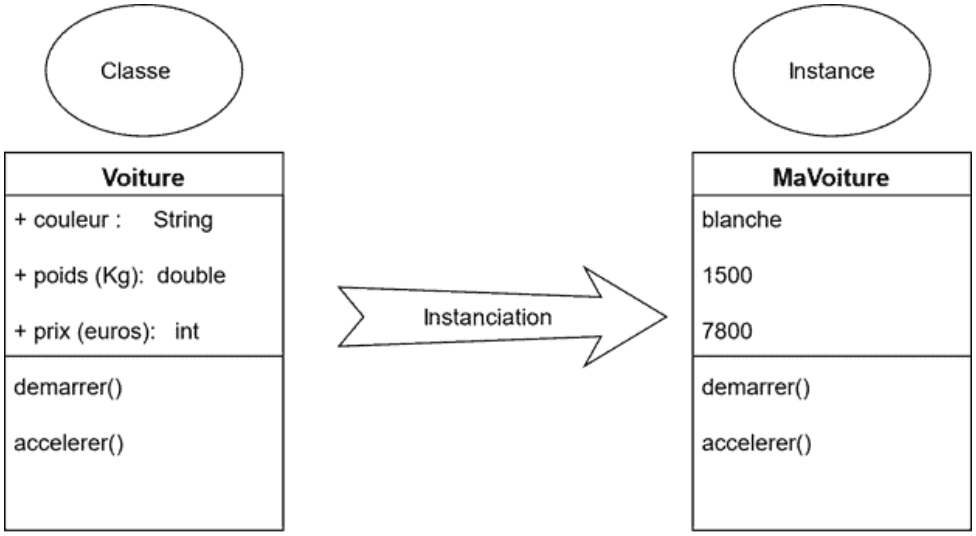

FIGURE 7 – Un exemple de classe et d'instanciation d'un objet de cette classe selon le paradigme objet

De nombreux langages de très haut niveau (Java, C++, Python) permettent de suivre le paradigme de programmation orientée objet.
Pour terminer, nous donnons quelques éléments de comparaison des différents paradigmes et de leurs mérites respectifs :

FIGURE 8 – Quelques langages de programmation...

	Impératif sans objets	Impératif à objets	Fonctionnel
Cas d'usage courants	Traitement de données, applications nécessitant des changements d'états ou d'interagir avec l'utilisateur.	Facilite et maintient la représentation et le traitement d'entités sémantiques cohérentes et de façon plus lisible.	Calculs mathématiques, production de résultats plutôt que de comportements ; rarement le seul paradigme pour une application entière.
Niveau d'abstraction	Le plus proche du processeur, qui fonctionne avec des séquences d'instructions.	Abstraction supplémentaire qui peut entraîner des traitements additionnels cachés.	Doit être converti en instructions impératives ; la récursion peut avoir un coût important.
Maintenance	Effets de bord : peut devenir complexe à maintenir et à déboguer ; plus intuitif pour certains.		Pas d'effet de bord : plus facile à déboguer, à tester, à optimiser dans certains cas.

III. 2. Langages de programmation

En général, le choix d'un langage de programmation va de paire avec le choix d'un paradigme de programmation privilégié. Mais le lien entre paradigme de programmation et langage de programmation est loin d'être univoque.

— Plusieurs langages encouragent le même paradigme de programmation : par exemple, `C++` et `Java` encouragent la programmation orientée objet.

— Certains langages sont très stricts et ne permettent de suivre qu'un seul paradigme de programmation. C'est par exemple le cas d'Haskell, qui requiert l'adoption d'un paradigme de programmation purement fonctionnel

— D'autres permettent d'utiliser simultanément plusieurs paradigmes différents. C'est le cas de `OCaml` et de `Python`, qui permettent de faire de la programmation impérative et de la programmation fonctionnelle.

— Le C, qui est plutôt influencé par un paradigme impératif, tolère en fait quasiment tous les paradigmes, mais souvent au prix d'une moindre lisibilité

Le choix du langage de programmation utilisé dépend en fait de nombreux facteurs :

— efficacité du code machine produit par les compilateurs ou les interpréteurs du langage,

— adéquation du langage aux choix d'architecture et au(x) paradigme(s) de programmation retenus

— rapidité de développement : mise à disposition de modules et de bibliothèques, exigences de rigueur et de détails demandées au programmeur

— dynamisme et importance de la communauté de développeurs formés sur ce langage (documentation, mises à jour, évolution du langage), portabilité du code (par exemple s'il existe des interpréteurs), documentation disponible

— usages et standards du domaine applicatif ciblé (calcul scientifique, application web, application mobile, embarqué, programmation graphique...etc)

III. 3. Le langage C

Le langage C a été inventé au cours de l'année 1972 dans les Laboratoires Bell. Il était développé en même temps que le système d'exploitation Unix par Dennis Ritchie et Kenneth Thompson. Il s'agit d'un langage de haut niveau au sens de la hiérarchie décrite en Section II. 1. mais moins évolué que des langages plus récents comment Java ou Python. Il est encore extrêmement utilisé dans le domaine du calcul scientifique, de la programmation embarqué, de la conception de systèmes d'exploitation ou de compilateur, et sert encore très souvent de langage intermédiaire pour la traduction des langages de très haut niveau en langage machine.

— Plusieurs compilateurs très performants, le plus connu étant `gcc`, permettent de créer des exécutables en langage machine très efficaces, avec une phase d'optimisation de génération du code machine éprouvée. Il n'existe pas d'interpréteur.

— Le paradigme de programmation impérative domine la conception de ce langage, mais le C permet aussi de faire de la programmation fonctionnelle et de la programmation objet.

— Le langage C est un langage exigeant : le programmeur doit préciser explicitement de nombreux aspects qui sont souvent gérer par l'interpréteur dans les langages de plus haut niveau : désallocation des blocs mémoire alloués au cours du programme, déclaration des types de toutes les variables utilisées (langage à typage statique), gestion des erreurs... De nombreuses bibliothèques sont toutefois disponibles pour faciliter le travail de développement.

— La communauté de développeurs est très grande et très dynamique

— Le langage C est majoritaire dans le domaine de la programmation embarquée, de l'internet des objets [11], du calcul scientifique haute performance...

11. IoT Internet of Things

III. 4. Le langage Python

Le langage Python date du début des années 90. Il s'agit d'un langage de très haut niveau. Il est devenu l'un des langages les plus utilisés ces dernières années.

Le langage machine est généralement généré lors de l'exécution par un interpréteur, [12]. Le code Python est généralement traduit dans un langage intermédiaire appelé `bytecode`, qui est ensuite exécuté par un interpréteur très performant contrôlant et optimisant son exécution. Actuellement, les interpréteurs Python deviennent très performants. Il est possible de lancer cet interpréteur dans une console pour une **exécution interactive** (une commande, une réponse, une commande, une réponse...etc). Il existe enfin des compilateurs permettant de traduire directement des codes Python en code machine avant exécution.

Le langage Python est multi-paradigme : il permet la programmation impérative, concurrente, fonctionnelle et objet.

Il s'agit d'un langage de très haut niveau permettant un développement très rapide. C'est un langage très souple qui permet de soulager le programmeur de certains aspects techniques (présence d'un ramasse-miette, typage dynamique, mécanismes de gestion des exceptions...etc). De très nombreuses bibliothèques permettent facilement l'intégration de fonctionnalités avancées et accélèrent ainsi le temps de développement en s'appuyant sur le travail d'autres programmeurs chevronnés.

La communauté s'agrandit chaque année et Python est actuellement l'un des langages les plus utilisés et qui connaît le plus fort développement

Python est utilisé dans le domaine de l'enseignement, du prototypage, de l'analyse de données, du calcul scientifique, de l'embarqué et dans de plus en plus de domaines...

III. 5. Le langage OCaml

Le langage OCaml date du début des années 90. Il s'agit d'un langage de très haut niveau. Il est peu utilisé dans le monde professionnel.

Il existe plusieurs façon de faire exécuter du code écrit en OCaml par une machine :

- La première possibilité est de compiler nativement le code OCaml en langage machine grâce au compilateur `ocamlopt`, lorsque celui-ci est présent sur la machine.
- La deuxième possibilité est d'utiliser l'interpréteur en mode **interactif** sous la forme d'une boucle d'interaction en ligne de commande, appelé `toplevel`, qui va analyser chaque instruction à la volée et la traduire en langage machine avant de l'exécuter. On lance cet interpréteur en appelant la commande `ocaml`.
- Enfin, il est possible de transformer un code écrit en OCaml en `bytecode` avant l'exécution en utilisant la commande `ocamlc`, puis de faire exécuter ce `bytecode` par le *runtime* `ocamlrun`.

Le langage OCaml s'appuie essentiellement sur le paradigme de la programmation fonctionnelle mais la programmation impérative est aussi possible

Il s'agit d'un langage de très haut niveau, très souple pour le programmeur (présence d'un ramasse-miette, typage dynamique, mécanismes de gestion des exceptions...etc). Des bibliothèques sont disponibles.

12. le plus utilisé étant `CPython`

La communauté est assez peu développée

Les domaines d'application sont plus restreints, majoritairement pour l'enseignement et la recherche, la démonstration théorique de codes...