Florent Becker | Olivier Bournez | Jean-Loup Carré
Mathieu Liedloff | Julien Reichert | Gérard Rozsavolgyi

INFORMATIQUE

MP2I-MPI

TOUT-EN-UN

DUNOD

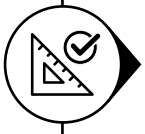
Couverture : création Hokus Pokus, adaptation Studio Dunod

Retrouvez nos ouvrages pour les prépas scientifiques ici

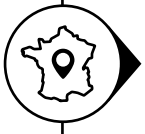
http://dunod.link/prepassc

NOUS NOUS ENGAGEONS EN FAVEUR DE L'ENVIRONNEMENT :

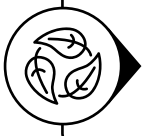
Nos livres sont imprimés sur des papiers certifiés pour réduire notre impact sur l'environnement.



Le format de nos ouvrages est pensé afin d'optimiser l'utilisation du papier.



Depuis plus de 30 ans, nous imprimons 70 % de nos livres en France et 25 % en Europe et nous mettons tout en œuvre pour augmenter cet engagement auprès des imprimeurs français.



Nous limitons l'utilisation du plastique sur nos ouvrages (film sur les couvertures et les livres).

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-085176-8

Table des matières

Avant-propos

Cet ouvrage s'adresse aux élèves des classes préparatoires scientifiques aux grandes écoles filière MP2I/MPI. Son contenu respecte le programme officiel d'informatique de ces deux années et, à ce titre, il peut être un allié des candidats à l'agrégation d'informatique, offrir un support solide aux étudiants de licence d'informatique et bien sûr proposer un contenu utile sur de nombreux points pour les étudiants de BUT informatique.

Chaque chapitre commence par une partie nommée *L'essentiel du cours*. On y présente les points les plus importants à connaître ; il est impératif de les maîtriser pour aborder ensuite les exercices proposés. Certains chapitres présentent des méthodes pour expliquer la démarche attendue lorsque c'est nécessaire. Par ailleurs, des questionnaires permettent au lecteur d'assurer sa compréhension élémentaire du cours, avant de s'engager dans la résolution des exercices.

L'ouvrage met l'accent sur une variété importante d'exercices, de difficultés variées. Parfois ces exercices sont issus d'annales de sujets de concours ou d'examens. Bien que les exercices proposés soient accompagnés de leur correction intégrale, nous déconseillons au lecteur de se plonger immédiatement dans la correction après avoir lu l'énoncé. Il est tout à fait profitable que le lecteur s'essaye par lui-même à trouver ses propres solutions (éventuellement différentes de celles proposées, sans que cela soit problématique). En principe, tout étudiant de MP2I/MPI, de licence, ou se préparant à l'agrégation devrait être en mesure de résoudre les exercices proposés, avec un travail adapté.

Cet ouvrage n'a pas pour objectif de se substituer à un livre traitant de la programmation en langage C ou en langage OCaml. Bien que ces deux langages servant de support au programme de MP2I/MPI soient bien introduits et détaillés, ainsi qu'utilisés très largement dans cet ouvrage, le lecteur désireux d'approfondir l'étude de ces langages est invité à se tourner vers des ouvrages spécifiques.

Nous avons utilisé certains pictogrammes tout au long de cet ouvrage :

Pour attirer l'attention du lecteur sur une remarque spécifique.

Pour attirer l'attention du lecteur sur des pièges potentiels.

Pour apporter au lecteur des conseils stratégiques.

Pour les remarques culturelles. [a]

a. Rendons à César ce qui est à César, cette icône a été réalisée par **momkik** du site flaticon.com

Le symbole  est utilisé pour indiquer une question ou un exercice assez difficile. Plus un exercice a de cerveaux, plus il est difficile ! La mention *1A* désigne un contenu plus spécique à la première année (MP2I) tandis qu'un contenu marqué *2A* est plus spécifique à la seconde année (MPI).

L'ouvrage se termine par des annexes variées sur les langages C, OCaml, SQL notamment, puis un index complet.

Malgré tout le soin apporté à cet ouvrage, il peut subsister quelques coquilles. Nous invitons le lecteur à nous les signaler, ainsi que tout commentaire de nature à faire progresser le contenu ou la présentation de l'ouvrage.

`info_mp2i_mpi@protonmail.com`

Avant de clore cet avant-propos, les auteurs souhaitent exprimer leurs remerciements à Martial Aufranc pour ses notes d'option informatique, ainsi qu'aux éditions Dunod.

CHAPITRE

Bases de l'informatique 0

L'essentiel du cours

■ 0 Notion d'algorithme

♦ 0.0 Introduction

Un **algorithme** est une description rigoureuse, en un nombre fini d'étapes, de la résolution d'un problème. Le problème est défini par un certain nombre d'entrées précises et donne lieu à un résulat. Ce résultat peut être une donnée complexe (comme un nombre, un texte, une image…) ou un simple booléen (vrai/faux). Dans ce dernier cas, on parle de **problème de décision**.
L'humanité utilise des algorithmes depuis très longtemps et la notion d'algorithme existe depuis la nuit des temps, bien avant l'apparition des premiers ordinateurs. Voici quelques repères historiques.

- Au II^{e} millénaire avant notre ère, les Mésopotamiens connaissaient déjà plusieurs algorithmes, par exemple pour le calcul de la racine carrée. Plusieurs tablettes de cette époque, présentant des calculs algorithmiques, ont été découvertes.
- Environ 300 ans avant notre ère, le mathématicien Euclide rédige ses *Eléments*, dans lesquels il décrit notamment un algorithme de calcul du PGCD. Cet algorithme s'appelle aujourd'hui « algorithme des soustractions successives », paradoxalement, on appelle « algorithme d'Euclide » une version améliorée de cet algorithme.
- Aux II^{e} et I^{er} siècles avant notre ère, le livre *Les Neuf Chapitres sur l'art mathématique* est compilé. On y trouve notamment la plus ancienne présentation de l'algorithme connu aujourd'hui sous le nom de « pivot de Gauss ».
- À peu près de l'an 780 à 850, vécut le mathématicien Muḥammad ibn Mūsā al-Khwārizmī, dont le nom, après déformation, donnera le mot « algorithme ».
- Au XX^{e} siècle est inventé l'ordinateur.
- En 1955, Jacques Perret propose au président d'IBM France de traduire le mot anglais « computer » par « ordinateur [0] [**Dep15**].

L'algorithme des soustractions successives, décrit dans le livre VII des *Éléments*, peut être présenté comme suit.

---

0. Sa lettre commence par : « *Cher monsieur, Que diriez-vous d'ordinateur ? c'est un mot correctement formé, qui se trouve même dans le Littré comme adjectif désignant Dieu qui met de l'ordre dans le monde.* »

Algorithme 1 : `euclide_originel(`a`, `b`)`

Entrées : Deux entiers positifs ou nuls a et b avec $a \geqslant b$
Sorties : Le plus grand commun diviseur de a et b
Tant que $a \neq b$ **faire**
  si $a > b$ **alors**
    $a \leftarrow a - b$
  sinon
    $b \leftarrow b - a$
Renvoyer a `// Retourne le PGCD`

Nous utilisons souvent des algorithmes, sans en avoir toujours conscience, par exemple lorsque nous cherchons un mot dans le dictionnaire [1], ou encore lorsque nous posons une addition, une multiplication ou calculons un pourcentage.
Le concept d'algorithme est indépendant de tout langage de programmation, mais pour le tester, les informaticiens seront toujours heureux de pouvoir concrétiser leur algorithme dans un langage de programmation spécifique. Ce langage permettra de dialoguer avec la machine à travers différentes couches logicielles impliquant notamment, outre le langage de haut niveau comme OCaml ou Python ou de niveau « intermédiaire » comme le C, des compilateurs, des langages de plus bas niveau comme le langage d'assemblage ou encore le micro-code qui seront plus « proches » de la machine.
Nous utiliserons des descriptions de haut niveau pour nos algorithmes comme :

- Le français, au moins pour une description informelle du problème, parfois également pour une description plus technique.
- Une description algorithmique dans un style normalisé [2].
- Le plus souvent le langage C ou le langage OCaml.
- Parfois un peu de Python ou un autre langage.

On compare souvent les algorithmes aux recettes de cuisine mais l'analogie a ses limites, une recette de cuisine étant souvent informelle et contenant beaucoup de notions implicites. Un algorithme doit être précis, ses entrées et sorties doivent être détaillées et satisfaire certaines **préconditions** et il se doit de traiter tous les cas envisageables et être prouvé, au moins partiellement, contrairement à une recette de cuisine. Il doit aussi être décrit de la façon la plus détaillée possible pour éviter les ambiguïtés lors de l'implémentation dans un langage donné.
Nous aurons à traiter pour cela les notions de **terminaison**, **correction**, **préconditions**, **variants**, **invariants**, qui pourront donner lieu à des commentaires dans les codes sources des programmes pour mieux les documenter.

♦ 0.1 Paradigmes de programmation

On pourra envisager un programme comme une instance d'un algorithme, mais concevoir un algorithme nécessite un travail de conceptualisation préalable, souvent non trivial.

1. On effectue en général une recherche par dichotomie sans le savoir.
2. Nous utilisons, dans cet ouvrage, le paquet *algorithm2e*, disponible en LaTeX, utilisé ci-dessus pour décrire l'algorithme d'Euclide

Traduire ensuite cet algorithme dans un langage particulier est une tâche généralement plus simple, bien que nécessitant une connaissance plus ou moins fine du langage employé.
Leslie Lamport, un grand nom de l'informatique distribuée (Prix Turing et inventeur de LaTeX) va même jusqu'à dire « *Coding is to programming what typing is to writing* », [3] c'est-à-dire qu'il pense que l'activité d'écrire des programmes dans un langage donné et de maîtriser la syntaxe de base de ce langage n'est pas du tout la même activité que concevoir des programmes et de comprendre les structures sous-jacentes, souvent d'ordre mathématique [**Lam**].
Il existe néanmoins divers *paradigmes de programmation* qui rendent le codage particulièrement différent selon le langage choisi. La notion de « paradigme de programmation » est un peu comme le « niveau de langue » lorsqu'on s'exprime. Cela fait référence à un style particulier pour présenter un algorithme. Comme pour présenter une recette, on pourrait le faire avec des pictogrammes, à l'aide d'un texte littéraire ou sous forme de liste à puces…
De la même façon, pour un même algorithme, on aura naturellement plusieurs manières de l'exprimer. On distingue notamment les paradigmes :

(0) Impératif
(1) Fonctionnel
(2) Logique
(3) Objet

Le **paradigme Impératif** est actuellement le plus répandu. Il recouvre des langages comme **C**, **Javascript** ou **Python**. Lorsqu'on écrit un programme en paradigme Impératif, on déclare des variables, et à travers leur évolution en mémoire, on cherche à maintenir ou à suivre l'**évolution d'un état**. On fournit généralement une séquence d'instructions ou de commandes à l'ordinateur permettant de modifier un ensemble de variables représentant cet état et de renvoyer à la fin un résultat.
Dans le **paradigme Fonctionnel**, comme en **Haskell**, **OCaml** ou **Lisp**, on utilise des fonctions comme briques de base d'un raisonnement et on va ensuite les assembler ou les composer pour construire des algorithmes plus sophistiqués. On dit parfois que, dans le paradigme Fonctionnel, les fonctions sont traitées comme « des citoyens de première classe » et peuvent être passées en arguments à d'autres fonctions ou retournées comme valeurs par d'autres fonctions. La programmation devient plus conceptuelle et concise que dans le paradigme impératif. Certains parlent « d'élégance » mais c'est aussi une question de goût ou d'habitude…
Dans le **paradigme Logique**, comme dans les langages **Prolog** ou **Datalog**, on programme en suivant un raisonnement logique en déclarant des faits et règles qui serviront à répondre à des requêtes grâce à un moteur d'inférence.
Dans le **paradigme Objet**, comme dans les langages **Java** ou **Smalltalk**, on conçoit des classes dont les instances représentent des objets du monde réel ou du monde conceptuel (Produits, Paniers d'achat, Véhicules, Comptes en banque, etc.). Un objet est une entité qui contient à la fois des données et des méthodes pour manipuler ces données. Les notions d'**encapsulation**, **héritage** entre classes et de **polymorphisme** dans les méthodes offrent au paradigme objet une grande puissance expressive.
Dans les paradigmes Objet et Fonctionnel, on se pose en particulier en premier lieu la question : « De quoi parle-t-on ? », en décrivant notre univers en termes d'objets ou

3. « Le codage est à la programmation ce que taper à la machine est à l'écriture. »

de fonctions, sans nécessairement tenter de répondre directement à la question « Que veut-on faire ? » comme on le fait souvent avec le paradigme Impératif, où l'évolution d'un ensemble de variables dans un but précis masque parfois les concepts sous-jacents et limite l'abstraction et la réutilisabilité.

♦ 0.2 Patrons de conception

Des **patrons de conception** (**design patterns**) peuvent également être utilisés pour concevoir une architecture évolutive, claire, maintenable et réutiliser des objets dans différents contextes. Les patrons de conception ont été initialement présentés par l'architecte américain **Christopher Alexander**.

Christopher Alexander, les « Design Patterns » et la méthodologie Agile L'idée de Christopher Alexander était de dégager des conceptions récurrentes et réutilisables dans le domaine de l'architecture, qu'il a présentées dans son ouvrage *A Pattern Language*, publié en 1977 [**Ale77**]. Cet ouvrage décrit un langage formel pour la conception de bâtiments, de villes et d'objets, et présente 253 patterns spécifiques. L'idée était de créer un ensemble de solutions à des problèmes communément rencontrés lors de la conception de bâtiments. Les conceptions d'Alexander ont ensuite essaimé dans de nombreux domaines comme le design urbain, la sociologie ou la conception de programmes informatiques. La mise en place du premier **wiki** s'est ainsi largement inspiré de Christopher Alexander. L'ouvrage précédent d'Alexander, *The Oregon Experiment*, publié en 1975 [**Ale75**], décrit un processus expérimental pour concevoir le campus universitaire de l'Université d'Oregon. Le livre est une exploration de la manière dont une communauté peut participer à la conception de ses propres espaces de vie et de travail, y compris les bâtiments et les paysages urbains. Christopher Alexander y aborde notamment les thèmes de :

- Patterns : ils sont également abordés dans cet ouvrage et l'idée qu'ils peuvent être réutilisés et combinés dans des conceptions complexes.
- Processus de conception organique : l'auteur y promeut une approche de conception qui permet une évolution et une modification continues, plutôt qu'un plan rigide et fixe dès le départ. L'idée est de permettre au design d'évoluer naturellement, en réponse aux besoins et aux objectifs évolutifs de la communauté.
- Adaptabilité : l'importance de la flexibilité dans la conception est affirmée. Les bâtiments et les espaces doivent être capables de changer et de s'adapter incrémentalement au fil du temps.
- Participation communautaire et décision locale : Plutôt que de mettre toute la prise de décision entre les mains d'un petit groupe d'architectes ou d'administrateurs, Alexander plaide pour la décentralisation de la prise de décision, permettant à ceux qui utilisent et vivent dans les espaces comme les professeurs et les étudiants de contribuer à leur conception.

The Oregon Experiment est aussi considéré comme un ouvrage précurseur, non seulement des design patterns mais aussi des approches modernes de gestion de projets informatiques comme la méthodologie **Agile**. En effet,

l'Agile promeut une planification continue et une amélioration itérative, où le produit se développe organiquement au fil des itérations du projet, appelés **sprints**. Le développement itératif et incrémental permet aux équipes de travailler en cycles courts et de livrer régulièrement des versions partielles mais fonctionnelles du produit final. Dans la démarche Agile, la collaboration avec clients et utilisateurs finaux est également un élément clé.

L'idée des design patterns dans la programmation informatique, influencée par le travail de Christopher Alexander a ensuite été popularisée en informatique par le livre *Design Patterns : Elements of Reusable Object-Oriented Software*, écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (connus sous le nom de *Gang of Four* ou *GoF*)[**GHJV94**]. Cet ouvrage a en particulier identifié 23 design patterns et les a organisés en trois catégories : les patterns de création, de structure et ceux de comportement. Ces patterns sont devenus un point de passage obligé de la littérature et de l'éducation en génie logiciel.
L'étude des design patterns est hors programme mais ils peuvent éclairer certains choix de conception ou manières génériques de résoudre des problèmes récurrents en programmation. Ces patrons de conception sont importants surtout dans une optique de construction de grands projets logiciels avec une architecture solide et évolutive. Ce n'est pas l'objet du présent ouvrage, qui sera davantage centré sur l'étude des algorithmes, leur prouvabilité et leur performance ainsi que les structures de données afférentes dans les langages C et OCaml.

Choix du paradigme De nombreux langages de programmation supportent de multiples paradigmes (avec plus ou moins de bohneur) comme le C++, le JavaScript, le Java ou le Python et un progammeur n'est pas obligé de se restreindre à un paradigme unique. En outre, un programmeur expérimenté peut choisir le meilleur paradigme (et un langage spécifique) en fonction du contexte du problème qui lui a été soumis.

Outre ces patterns, des notions de sémantique des programmes, l'utilisation de langages plus sûrs, fortement typés, dynamiques ou non, de systèmes de preuves de programmes, offrent la possibilité de prouver, au moins partiellement, du code et permettent la conception et la réalisation des cathédrales abstraites contemporaines que sont par exemple les systèmes d'exploitation, les systèmes de navigation aériens ou encore les écosystèmes de programmation d'une très grande richesse comme les **IDE** ou les chaînes de compilation. Nous aborderons certains de ces thèmes dans les prochains chapitres.

♦ 0.3 Modèles de calcul

Pour implémenter des programmes, il nous faut un ordinateur. Mais pour pouvoir raisonner sur notre objet « ordinateur », auquel on fournira des « programmes », il faut qu'on le modélise : on en extrait les caractéristiques fondamentales et on simplifie les détails spécifiques.
On fixe ainsi un modèle concret de calcul, comme les architectures les plus connues de type **von Neumann**, élaborées à la fin de la Première Guerre mondiale (voir le programme de NSI Terminale, par exemple : [**CPRS23**]).

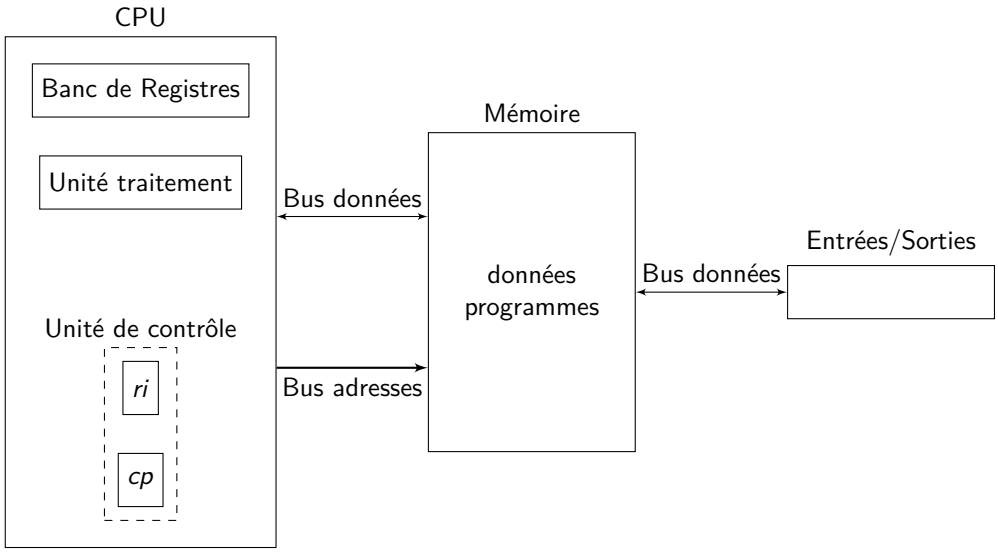


FIGURE 0.0 – Architecture de Von Neumann

On distingue sur ce schéma :
- L'unité centrale, ou CPU.
- La mémoire où sont stockés les données et les programmes.
- Des **bus** de données ou d'adresses permettant d'échanger des données entre composants.
- Des entrées-sorties pour toutes sortes d'échanges avec l'extérieur.

Un programme est enregistré dans la mémoire. L'adresse de l'instruction en cours de traitement est stockée dans le *registre compteur de programme* (`cp` ou `pc` en Anglais). La valeur de cette instruction (aussi un entier) est stockée dans le *registre d'instruction* `ri`.
L'UAL (Unité Arithmétique et Logique) effectue les opérations arithmétiques et logiques.
Le CPU dispose d'une horloge qui va cadencer l'exécution des instructions. L'unité est appelée **cycle**.
Lorsqu'on parle de processeur cadencé à 3,7 GHz, cela signifie qu'il y a 3,7 milliards de cycles d'horloge par seconde. Jusqu'au milieu des années 2000, la fréquence des processeurs a augmenté linéairement avant de stagner du fait de certaines limitations physiques (dissipation de chaleur, etc.). La puissance globale des microprocesseurs a néanmoins remarquablement suivi la loi de Moore généralisée qui prédit son doublement tous les deux ans environ. Les améliorations récentes portent plutôt sur la miniaturisation (gravure jusqu'à 3 nm) et l'intégration de tous les éléments sur un seul **SoC** (System on a Chip) comme dans la famille des processeurs ARM de type RISC.
Dans un processeur, chacune des actions suivantes est exécutée durant un nombre prédéfini de cycles :

(0) Lire l'instruction.
(1) Décoder l'instruction.
(2) Exécuter l'opération dans l'UAL.
(3) Lire ou écrire dans la mémoire.
(4) Écrire le résultat dans des registres.

En dehors de ce modèle concret de Von Neumann, le modèle abstrait le plus connu est celui des machines de Turing (voir le chapitre 11).
De très nombreux autres modèles de calcul, inspirés par différents modèles de machines, existent et sont d'une étonnante diversité :
- Machines de Turing quantiques,
- Automates cellulaires (du type Jeu de la Vie),
- Modèle MMIX de Donald Knuth,
- **Lambda calcul**,
- Machine de Krivine,
- Noyaux fonctionnels de langages comme OCaml, Haskell, Lisp, Scheme (en tant que modèles de calcul théoriques, voir le chapitre 11),
- Modèle rationnel de John Conway : FRACTRAN,
- Modèles de calcul continu comme la théorie des signaux de Durand-Lose,
- Modèles de calcul analogiques, comme les ordinateurs basés sur de l'électronique analogique,
- Modèles de calcul à ADN.

Donald Knuth et l'Art de programmer

Donald Ervin Knuth est un informaticien et mathématicien américain, professeur émérite à l'Université de Stanford. Il est l'un des pionniers de l'informatique moderne, ayant fait des contributions à l'analyse des algorithmes, à la conception de systèmes de programmation, ainsi qu'à la typographie et à la mise en page de documents.

Son œuvre la plus connue est la série de livres intitulée *The Art of Computer Programming*, un projet monumental qui a débuté dans les années 1960, toujours en cours de rédaction, qui lui a valu le Prix Turing en 1974 [**Knu68**]. C'est une série exhaustive et influente de livres qui couvre de nombreux sujets comme la conception d'algorithmes et l'évaluation mathématique de leur complexité.

Le travail est réparti en plusieurs volumes spécifiques :

- Volume 1 : *Fundamental Algorithms* – Couvre les bases, comme les algorithmes arithmétiques, la manipulation de données, et les structures de données fondamentales [**Knu68**].
- Volume 2 : *Seminumerical Algorithms* – Traite des sujets tels que l'arithmétique en virgule flottante et la génération de nombres aléatoires [**Knu14**].
- Volume 3 : *Sorting and Searching* – Se concentre sur les algorithmes de tri et de recherche [**Knu98**].
- Volume 4 : *Combinatorial Algorithms* – Explore les algorithmes combinatoires [**Knu13**].

Knuth prévoit d'autres volumes, mais le travail est titanesque et inachevé. Ses livres sont réputés pour leur rigueur et leur précision et souvent cités comme des références sur l'analyse mathématique des algorithmes.

Outre cet ouvrage, Knuth est également connu pour avoir créé le système de composition de documents **TEX** et le système de génération de graphiques et de fontes vectorielles `METAFONT`. Le système TEX est largement utilisé dans la communauté scientifique pour rédiger des articles, des rapports et des thèses.

Knuth a également proposé un modèle de calcul qu'il utilise dans l'analyse des algorithmes : **MIX**, mis à jour dans le modèle **MMIX**, un modèle de machine à registres, basé sur une architecture 64 bits possédant 256 registres généraux. MMIX est doté d'un ensemble complet d'instructions, arithmétiques et logiques, d'autres dédiées au chargement/stockage ou encore au contrôle de flux. Le modèle est décrit en détail dans [**Ruc15**].

Donald Knuth offre une récompense de $2,56$ \$ (USD) à la première personne qui trouvera une erreur dans ses livres publiés, qu'elle soit technique, typographique ou historique. Knuth explique que $2,56$ dollars, soit 256 cents, correspondent à un dollar hexadécimal.

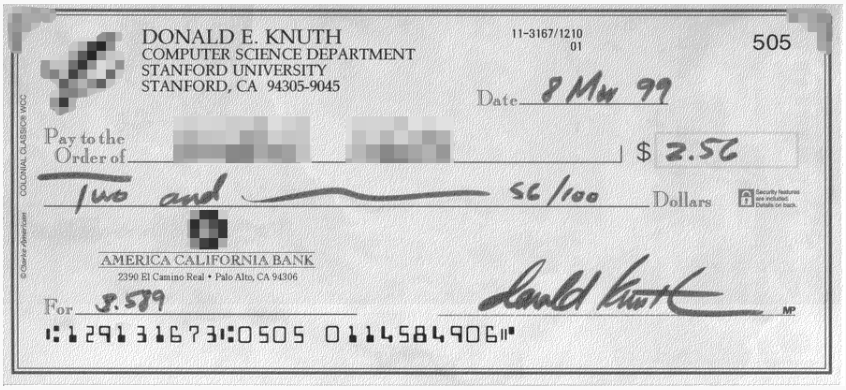


FIGURE 0.1 – Le chèque de Donald Knuth de 256 cents.

■ 1 Notion de machine

♦ 1.0 Le problème

Considérons une machine concrète *Machine* 0 qui travaille sur des représentations binaires avec un jeu d'instructions assembleur déterminé. On peut le programmer avec un certain langage d'assembleur *Langage* 0.

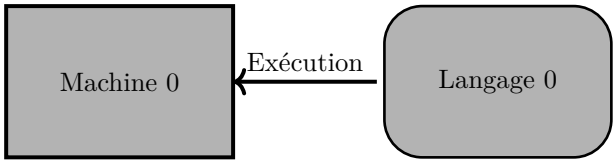


FIGURE 0.2 – Machine 0, langage 0 « natif ».

Ceci constitue la situation élémentaire classique, mais l'informatique se construit sur de multiples couches et nous pouvons envisager un langage *Langage* 1 distinct de *Langage* 0, de plus haut niveau et nous demander comment exécuter des programmes écrits en $L1$.
Cela serait aisé si nous disposions d'une machine *Machine* 1 capable d'exécuter directement des programmes écrits en *Langage* 1 :

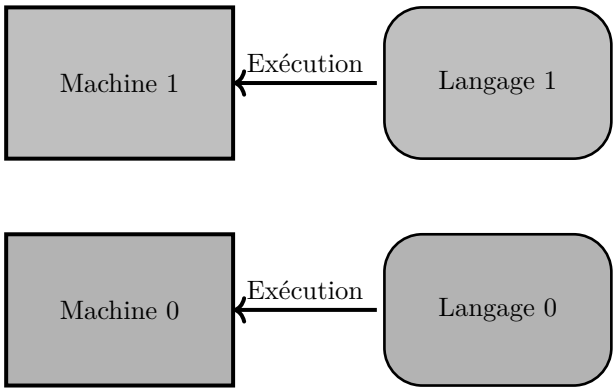


FIGURE 0.3 – Machine M1 native.

♦ 1.1 Traduction et interprétation

Mais si nous ne disposons pas de machine dédiée à exécuter des programmes écrits en *Langage* 1, nous pouvons envisager deux solutions principales pour exécuter tout de même des programmes écrits dans ce langage, à savoir la **traduction** ou l'**interprétation**.

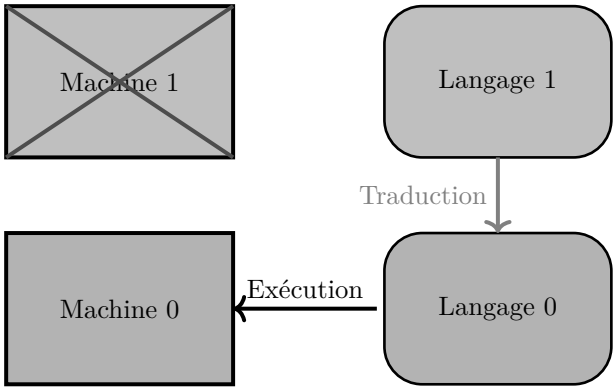


FIGURE 0.4 – La traduction

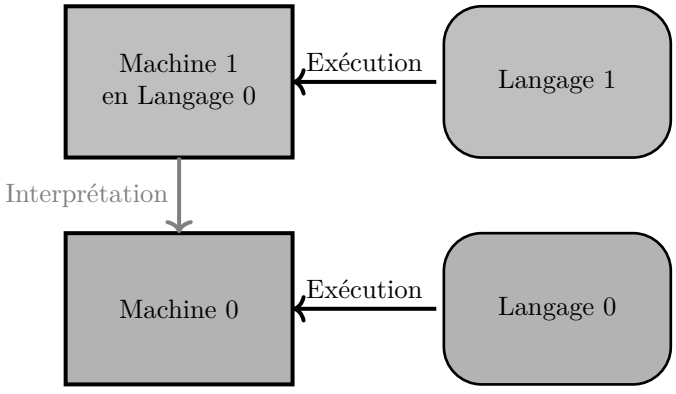


FIGURE 0.5 – L'interprétation

♦ 1.2 Combinaisons pour différents langages

On peut envisager des combinaisons variées avec de multiples langages impliqués, dans lesquels on dispose soit de Machines Virtuelles (VM) dédiées, soit de traducteurs comme par exemple :

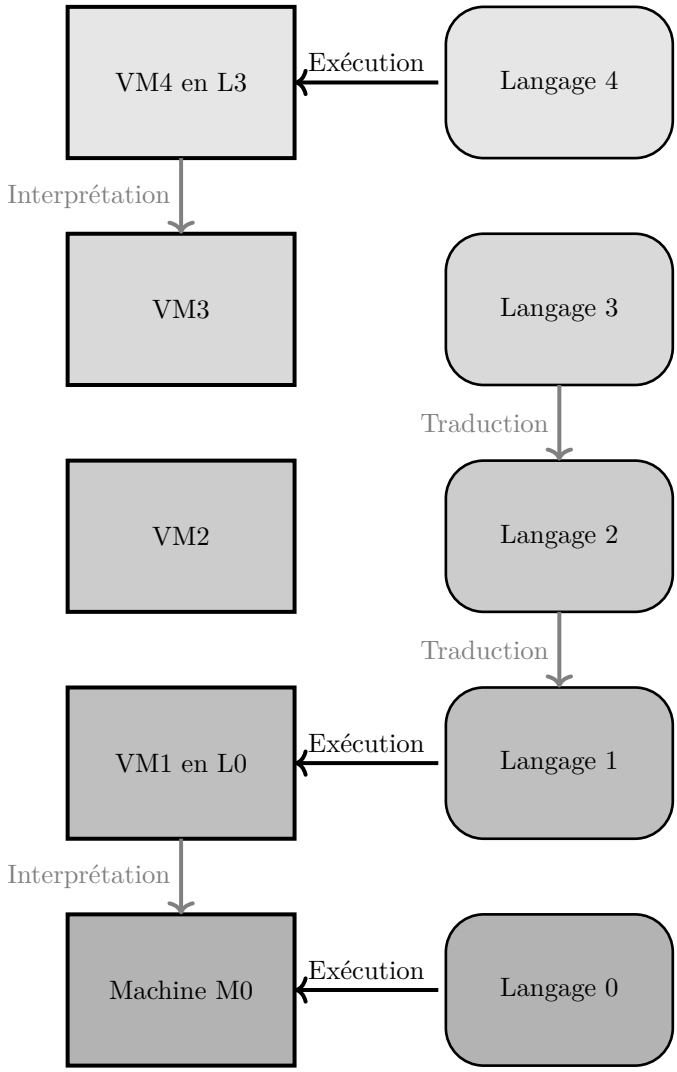


FIGURE 0.6 – Gestion de plusieurs langages avec des traductions et des VM.

Ces situations se retrouvent couramment en informatique, par exemple Java ou Python peuvent être compilés en byte-code puis exécutés par des machines virtuelles. En Java, cette machine virtuelle s'appelle la **JVM** (Java Virtual Machine).
Les programmes C se compilent directement en code natif.
OCaml quand à lui offre différentes options pour s'exécuter :

- Bytecode : OCaml peut être compilé en bytecode, tout comme Java ou Python. Ce bytecode peut ensuite être exécuté par la machine virtuelle OCaml (l'équivalent de la JVM pour Java). La compilation en bytecode est généralement plus rapide que la compilation en code natif, mais le bytecode résultant est le plus souvent moins performant que le code natif.
- Code natif : OCaml peut aussi être compilé directement en code machine pour un grand nombre d'architectures. Si le code résultant peut être bien plus rapide que le bytecode, la compilation elle-même peut être un peu plus lente. OCaml est souvent reconnu pour sa capacité à produire du code natif très performant.
- Interprétation : OCaml dispose également d'un interpréteur (dit **toplevel**), permettant d'exécuter du code OCaml directement sans nécessité de compilation préalable, utile pendant les phases de développement, de test ou d'apprentissage.

Nous allons ensuite nous consacrer à une introduction aux deux langages principaux à l'honneur en MP2I/MPI pour exprimer nos algorithmes, à savoir le langage OCaml et le langage C.

■ 2 Le langage OCaml

Pour le lecteur débutant en OCaml, l'annexe 2 présente le langage depuis les premiers pas.

♦ 2.0 Les données et leurs types en OCaml

Les types de données, découverts par exemple en Python[4], ont une utilisation plus rigoureuse en OCaml, dans la mesure où le compilateur[5] ne va jamais faire de conversion lui-même en cas d'incompatibilité du type d'une expression avec le type attendu par son utilisation.
Ainsi, la plupart des fonctions attendent un certain nombre d'arguments d'un type précis et leur valeur de retour en possède un également, ce qui se lit dans la signature de chaque fonction.
Le typage en OCaml est dit **typage statique**, au contraire du **typage dynamique** d'autres langages. La différence réside dans le moment où un type est associé à chaque expression : respectivement *avant* ou *pendant* l'exécution. On retrouve aussi le typage statique en C, par exemple dans la déclaration du type (le `int` de `int x = 42;`) au moment de créer une variable.
Pour autant, le type d'une expression n'est pas forcément précis. Il est déduit en OCaml par les opérateurs et fonctions, entre autres, impliqués dans l'expression : c'est ce qu'on appelle **l'inférence de type**, ce qui laisse parfois un doute. Par exemple, la fonction identité sera de type « fonction qui à une expression de type noté `'a` associe

4. Cependant, le système des types de données est très différent entre les deux langages.
5. Les expressions ne sont pas à proprement parler interprétées en OCaml, mais ceci est une autre histoire ; *a contrario*, on n'a pas de compilateur en Python.

un objet de type `'a` » (on parle de **polymorphisme**). De même, l'opérateur de comparaison nécessite deux opérandes du même type, mais celui-ci peut être quelconque. En particulier, on n'a pas le droit de comparer `0` et `0.0`, ce qui risque d'être assez déstabilisant.
La gestion de la mémoire en OCaml est automatique. Inutile donc de s'inquiéter avec les allocations. En outre, il existe un ramasse-miettes pour libérer la mémoire qui n'est plus utilisée.

♦ 2.1 Types de base

Les types de base comprennent les entiers (`int`), les flottants (`float`) et les booléens (`bool`). Attention, les booléens, notés `true` et `false`, n'héritent pas de la structure d'entier et on ne peut donc pas les assimiler à `0` et `1`.
Les entiers sont encodés, du moins sur les ordinateurs habituels, sur 63 bits. Des dépassements arithmétiques peuvent donc intervenir autour de 10^{18}, justifiant l'existence d'un module pour gérer des entiers de précision arbitraire (hors programme). Les flottants sont évidemment également limités en précision et sujets aux dépassements et soupassements arithmétiques.
Plus précisément, les opérations spécifiques sont :

— `+`, `-`, `*`, `/` et `mod` pour les entiers[6].
— `+.`, `-.`, `*.`, `/.` et `**` pour les flottants[7].
— `&&`, `||` et **`not`** pour les booléens.

Le point ne se met qu'après ces opérateurs (et on l'utilise aussi comme séparateur décimal), inutile de le mettre après les noms de fonctions agissant sur les booléens…

On produit des booléens à l'aide de comparaisons, les symboles étant `<`, `>`, `<=`, `>=`, `=` (égalité[8], les affectations fonctionnant différemment comme nous le verrons plus tard) et `<>` (différence[9]).
L'évaluation des opérations booléennes est **paresseuse**. Pour rappel, cela signifie que l'évaluation de l'expression `b1 && b2` ignore `b2` si `b1` s'évalue à `false`, n'occasionnant ni les éventuels effets de bord ni les éventuelles erreurs. De même pour `b1 || b2` si `b1` s'évalue à `true`.

6. L'opérateur `mod` remplace donc le `%` de beaucoup d'autres langages. On pourra s'apercevoir que la division euclidienne a un comportement anormal sur les valeurs négatives, par exemple `-3/2` vaut `-1`, mais il n'est pas nécessaire de le savoir, et en cas de bug faire le test suffira à comprendre l'origine du problème. Quant au modulo, il est bon de savoir que `a mod (-b)`, en n'oubliant pas les parenthèses, équivaut à `a mod b`, mais que cette valeur est comprise entre `0` et `b-1` inclus si `a` est positif, et entre `-b+1` et `0` inclus si `a` est négatif.

7. Ceci est une première illustration de ce qui a été annoncé pour les types : même les opérateurs doivent être modifiés. En pratique, ce n'est simplement pas le cas pour la puissance car elle n'existe pas pour les entiers et s'applique donc à deux flottants. Il est également interdit d'omettre le `0` devant le point si un flottant a `0` pour partie entière.

8. Au passage, l'opérateur `==` existe et son utilisation est rare : il détermine si les deux objets comparés occupent la même adresse mémoire.

9. La même remarque s'applique quant à `!=`.

Pour additionner un entier et un flottant, on est obligé de procéder à une conversion manuelle, à l'aide de `float_of_int` (ajoute un point) et `int_of_float` (troncature, pas la partie entière).
Les caractères (`char`) forment un type de base à part entière, ils sont entourés d'apostrophes simples sans second choix. Il est interdit de placer plusieurs caractères entre apostrophes, comme en langage C (les caractères échappés du genre `'\n'` ne comptent que pour un).
Les chaînes de caractères sont un type que l'on considère encore comme simple, quand bien même ils possèdent une structure similaire à des types composés comme les tableaux (une remarque similaire peut être faite pour Python).
On les produit en entourant un certain nombre de caractères par des guillemets, également sans second choix, la chaîne vide étant alors `""`.
L'accès à un caractère se fait à l'aide de la syntaxe `t.[i]`, où `t` est une chaîne de caractères et `i` est un indice entre `0` et la longueur de `t` moins un, cette longueur s'obtenant à l'aide de la fonction `String.length`, puisque là aussi on ne dispose pas d'une unique fonction calculant la longueur pour des objets de types totalement différents.
Une chaîne de caractères n'est plus mutable dans les dernières versions du langage. Pour simuler une chaîne de caractères mutable, les deux possibilités sont le type `bytes` ou, bien plus simplement, l'utilisation d'un tableau de caractères.
On peut obtenir la concaténation de deux chaînes de caractères par l'opérateur `^` et extraire une sous-chaîne à l'aide de la fonction `String.sub`, prenant en arguments une chaîne, l'indice de départ et la longueur de la sous-chaîne à extraire, retournant une erreur en cas de débordement. Une chaîne de caractères peut également s'initialiser à l'aide de la fonction `String.make`, qui prend pour arguments la longueur et le caractère à répéter.
Caractères et chaînes de caractères supportent la comparaison, aussi bien en ce qui concerne le test d'égalité que les tests d'infériorité. L'ordre total sur les caractères correspond à la comparaison de leur position dans la table de caractères utilisée, et l'ordre total sur les chaînes de caractères, s'appuyant sur l'ordre précédent, est l'ordre lexicographique, reprenant les règles de l'ordre alphabétique sans se limiter aux lettres.
Ainsi, une chaîne `s` sera inférieure à une chaîne `t` si et seulement si `t` commence par `s` en entier ou s'il existe un entier naturel `k` tel que tous les caractères de `s` et de `t` jusqu'au `k`-ième inclus sont les mêmes et le `k+1`-ième caractère de `s` est strictement inférieur au `k+1`-ième caractère de `t`.

♦ 2.2 Séquences de base

La structure de **tableau** (`array`) fournit des séquences indexables d'éléments, avec une certaine rigidité : un tableau ne peut contenir que des expressions du même type. Ainsi, un tableau sera par exemple déclaré comme un `int array` s'il contient des entiers, un `string array array` (le type se lit dans ce cas de droite à gauche, en quelque sorte) s'il contient des tableaux, eux-mêmes contenant des chaînes de caractères, etc. [10]

10. Un `'a array` contient des données dont le type n'est pas encore imposé, par exemple le tableau vide.

On produit un tableau en délimitant les données par des **points-virgules** et en les entourant par `[|` et `|]` [11], le tableau vide étant alors `[||]`.
L'accès à un élément du tableau se fait cette fois-ci à l'aide de la syntaxe `t.(i)` (analogue), la longueur du tableau s'obtenant à l'aide de la fonction `Array.length`. La fusion de tableaux n'est pas prévue, ce qui veut dire que la taille n'est pas modifiable. En revanche, on peut modifier les éléments à l'aide de l'opérateur `<-` et extraire un sous-tableau à l'aide de la fonction `Array.sub`.
Un tableau peut s'initialiser en le donnant tel quel ou par la fonction `Array.make` (syntaxe similaire à celle de `String.make`).

Les soucis classiques lorsqu'on déclare, dans un langage de programmation habituel, une variable de même fonctionnement qu'un tableau comme étant égale à une autre telle variable, sans faire de copie, existent en OCaml. De manière analogue, définir par exemple un tableau `m` de dimension deux comme `Array.make 3 (Array.make 3 0)` fait que modifier `m.(0).(0)` entraîne la même modification de `m.(1).(0)` et `m.(2).(0)`. La solution est de définir des matrices à l'aide de `Array.make_matrix`, prenant pour arguments le nombre de lignes, le nombre de colonnes et l'élément à répéter.

Les listes [12] (`list`) sont une structure spéciale, dans la mesure où elles ont un lien fort avec la récursivité.
Le plus simple est de donner une définition récursive : une liste est soit vide, soit la donnée d'un élément et d'une autre liste. [13]
Ainsi, on ne peut pas accéder à un élément de la liste, sauf le premier, en une opération, comme on le ferait pour un tableau.
Une liste ne peut contenir que des données du même type, et on aura donc par exemple des `int list`, des `float array list`…
On les produit en délimitant les données par des **points-virgules**, mais cette fois-ci en les entourant par `[` et `]`.
On peut aussi effectuer un préfixage à l'aide de l'opérateur « conse » `::`, dont la partie gauche est nécessairement un seul élément, ou une fusion à l'aide de l'opérateur `@`, dont les deux parties sont des listes à fusionner. Il faut cependant avoir à l'esprit que la fusion occasionne un coût linéaire en la taille de l'opérande de gauche.

L'expression `a::l` produit une nouvelle liste sans modifier `l`.

La longueur d'une liste s'obtient à l'aide de la fonction `List.length` [14].
Une liste en elle-même n'est pas modifiable (pour rebondir sur la mise en garde), cependant, et comme dit précédemment, l'accès à ses éléments ne se fera que par filtrage, ce sur quoi nous reviendrons. Il existe une fonction pour accéder au premier élément d'une liste non vide, dont on évitera l'utilisation : `List.hd`, et de même pour accéder au reste : `List.tl`. On comprendra bien que les fonctions `List.sub` et `List.make` n'existent pas.

11. La barre verticale s'obtient à l'aide de AltGr + 6/- sur un clavier AZERTY de PC.
12. La structure algorithmique est celle d'une liste simplement chaînée. Le raccourci de liste est courant pour les langages fonctionnels.
13. À ce stade, on peut imaginer des listes infinies, et en fait… elles existent en OCaml.
14. Attention, la fonction parcourt la liste, donc son coût est linéaire.

♦ 2.3 Conversions

Les conversions entre types sont possibles (cf. `float_of_int`), et pour cela on dispose de multiples fonctions. Il est bon de les connaître, mais elles sont censées être rappelées en cas de besoin [15] :

- `int_of_string` et `string_of_int`, de même en remplaçant `int` par `float` ou `bool`.
- `int_of_char` et `char_of_int` procèdent à une conversion, l'entier étant la position du caractère dans la table ASCII étendue (256 caractères).
- `Array.of_list` et `Array.to_list`, de complexité linéaire en raison de la structure de liste.

Les **n-uplets**, ou **tuples**, sont la structure de données en OCaml la plus proche de leur pendant en Python : ils rassemblent des éléments de types quelconques, on les forme en donnant les éléments séparés par des virgules et entourés de parenthèses ou non au choix (attention cependant aux cas d'ambiguïté), et ils peuvent être déconstruits. Cependant, ils n'ont pas de type à part entière (ce qui a pour conséquence directe qu'on ne peut pas calculer leur longueur), mais leur type est le produit des types de leurs éléments ; ainsi, un couple formé par une chaîne de caractères et une liste d'entiers aura pour type `string * int list`, alors que `["",1]` sera une `(string * int) list`, en notant bien qu'elle comporte un seul élément, qui est un couple.
Pour des couples uniquement, on dispose des fonctions `fst` et `snd` retournant respectivement le premier et le deuxième élément du couple.

♦ 2.4 Types avancés

Le type `unit` doit être mentionné ici par souci d'exhaustivité, mais il sera bien plus détaillé ultérieurement. On peut l'assimiler au type de `None`, `NULL` ou `nil` d'autres langages, car c'est le type des objets vides.
En parlant de `None`, cette expression existe aussi en OCaml, et correspond à un type composé, appelé `option`. Un type `'a option` propose une alternative : avoir quelque chose (constructeur `Some` suivi d'une expression de type `'a`) ou ne rien avoir (constructeur `None`).
Il est bien entendu possible de se passer totalement des types `option`, mais parfois cela arrange de les utiliser, et en tout état de cause il faut les connaître car ils sont désormais officiellement au programme.
Le dernier type prédéfini au programme est la *référence*, qu'on présentera au moment de mentionner la programmation impérative.

♦ 2.5 Variables

Cette information peut faire un choc : en OCaml, la notion de variable n'existe pas. En pratique, on peut tout à fait considérer que les références sont des variables, et c'est ce que nous allons faire par la suite, lorsque nous aurons besoin de traits impératifs.

15. … besoin qui n'est pas à la hauteur de l'abus de telles fonctions !

Pour commencer, donnons enfin la syntaxe des affectations : `let objet = valeur;;` définit globalement `objet` comme étant `valeur` *ad vitam æternam*, en pratique jusqu'à la prochaine affectation (qui ne fait que masquer la précédente) :
`let objet = autre_valeur;;`.
Comme dans la plupart des langages de programmation, un nom de variable doit commencer par une minuscule ou un **tiret-du–bas** ou **underscore** : `_` et contenir uniquement des chiffres, minuscules, majuscules et `_`.
Une définition locale s'écrit `let objet = valeur in morceau_de_code;;`, auquel cas `objet` ne sera `valeur` que dans le morceau de code en question.
Une façon de voir les choses est que toutes les occurrences de `objet` (là où on l'a défini) sont remplacées par `valeur` (sauf présence d'effets de bord dans la définition).
Cependant, une affectation ne peut pas se faire n'importe comment, et on utilisera majoritairement et prioritairement des définitions locales.

♦ 2.6 Programmation fonctionnelle en OCaml

OCaml étant un langage fonctionnel, la notion de fonction est bien sûr ici d'une importance capitale.
Une fonction est une expression qui attend un ou plusieurs arguments[16] et qui retourne une valeur.
La définition la plus simple d'une fonction est la donnée du nom de celle-ci, puis de noms de variables pour ses arguments et enfin de l'expression (ou la séquence d'instructions, comme nous le verrons plus tard) après le symbole `=`, par exemple :

```
let module_carre (x, y) = x *. x +. y *. y;;
```

La signature d'une fonction est l'information sur les types des arguments et de la valeur de retour. Elle est donnée en OCaml par la syntaxe :
`nom : type_arg1 → type_arg2 → ··· → type_retour`
La fonction ci-avant a pour signature `module_carre : float * float → float`, et la console affiche `val module_carre : float * float → float = <fun>`.
Bien entendu, une fonction peut prendre une autre fonction en argument, nous en verrons plusieurs cas et en créerons également. On parle alors de **fonction d'ordre supérieur**.
Les arguments d'une fonction sont évalués avant que le code de la fonction ne s'exécute. En outre, ils sont passés **par valeur**, c'est-à-dire que la fonction travaille sur une copie de la mémoire d'où les arguments proviennent. Cela n'empêche cependant pas que les mutations d'un objet mutable passé en argument d'une fonction ne se répercutent sur la mémoire. On pourra comparer le passage par valeur, le passage **par référence** et le passage **par affectation** pour en savoir plus.

♦ 2.7 Curryfication et fonctions partielles

Ce qu'il faut comprendre, dans la syntaxe précédente de la signature d'une fonction, c'est qu'en fait la fonction `nom` n'attend qu'un argument et retourne une autre fonction

16. La syntaxe de la signature d'une fonction permet de constater qu'une fonction sans arguments est simplement une constante. Si une fonction ne doit dépendre de rien, on lui donne en fait comme argument `()`, qui est de type `unit`. En particulier une constante est évaluée lors de sa définition : `let a = Random.int 6;;` affecte à `a` un nombre aléatoire entre `0` et `5`, ce qui n'a rien à voir avec `let b () = Random.int 6;;` qui définit une fonction retournant à chaque appel un nouveau nombre aléatoire entre `0` et `5`.

de signature `nom1 : type_arg2 -> ... -> type_retour`, pour laquelle le principe est le même.
Il est donc tout à fait possible de définir une **fonction partielle** à partir d'une telle fonction en renseignant successivement des arguments.
Tester à ce sujet le code suivant :

```
let add x y = x + y;;

let add3 = add 3;;
add3 4;;
```

ou encore, en utilisant les opérateurs standards d'addition et de multiplication d'OCaml qui attendent deux arguments, on peut fixer l'un des deux pour obtenir simplement des fonctions d'incrémentation ou de multiplication par deux :

```
let inc = ( + ) 1;;
```

```
let double = ( * ) 2;;
```

Ceci ne serait par ailleurs pas possible avec une fonction qui prend en entrée un n-uplet d'arguments. Pour des raisons pratiques, on utilisera donc moins souvent ce dernier type de fonction. On dit qu'une fonction attendant *en quelque sorte* plusieurs arguments est **curryfiée** [17].
On notera qu'il est possible d'écrire en OCaml une fonction pour curryfier et décurryfier des fonctions ayant le même nombre d'arguments.
Ainsi, `let curry2 f a b = f (a,b);;` décrit une fonction dont la signature est `curry2 : ('a * 'b * -> 'c) -> 'a -> 'b -> 'c`,
et si `f` est une fonction prenant un argument sous forme de couple, `curry2 f` est sa version curryfiée prenant deux arguments.

♦ 2.8 Disjonction de cas (if)

La disjonction de cas peut être utilisée en programmation fonctionnelle comme en programmation impérative, on en aura besoin très rapidement et on l'introduit ici.
La syntaxe de la disjonction de cas est `if condition then expr1 else expr2`, là encore le code pouvant être espacé comme on le souhaite. Il est impératif que la condition s'évalue en un booléen et surtout que le type obtenu en évaluant les deux expressions (pouvant se résumer à des instructions) soit le même.
Il n'y a pas de raccourci de type `elif` ou `elseif`, il faut imbriquer les disjonctions le cas échéant, ce qui alourdit la notation et renforce le besoin de bien présenter.

Dans le code `if cond1 then if cond2 then bloc1 else bloc2`, OCaml comprendra que le **else** correspond au **if** intérieur, et si ce n'est pas ce qu'on souhaite, il faut parenthéser (pas d'accolades comme dans d'autres langages).

Au passage, si les parenthèses offrent un souci de lisibilité, on peut les remplacer par **begin** et **end**. Ces mots-clés sont équivalents en tout point aux parenthèses, mais on ne peut pas apparier l'un d'entre eux à une parenthèse.

17. Du nom de l'informaticien Haskell Curry

♦ 2.9 Filtrage

Une fonction peut aussi se définir par **filtrage** (exhaustif, sinon OCaml déclenche un avertissement) des cas, et on retiendra pour commencer la syntaxe suivante (une syntaxe alternative, non explicitement au programme, sera introduite au fur et à mesure sur des exemples) : `match expr with`, où `expr` peut être n'importe quelle expression, habituellement un simple nom de variable de n'importe quel type ou un n-uplet de noms de variables.
Le filtrage se fait en introduisant chaque cas (sauf éventuellement le premier) par une barre verticale et en le faisant suivre d'une flèche et du bloc d'instructions quand le cas est rencontré, en utilisant pour l'esthétique une représentation alignée verticalement. On peut se servir du joker `_` pour signifier « tous les cas restants » en ayant bien à l'esprit que **seul le premier cas favorable est retenu**. Ainsi, un filtrage peut s'assimiler à un enchaînement de tests conditionnels.
Voici une illustration de cette syntaxe pour définir des fonctions booléennes.

```
let et b1 b2 = match (b1, b2) with
|(true, true) -> true
|_    -> false (* (_,_) marche aussi *);;

let ou (b1, b2) = match (b1, b2) with (* cas avec un seul argument *)
|(true, _) -> true
|(_, true) -> true
|_ -> false;;
```

Attention, les signatures dépendent de la façon dont les arguments sont présentés. On retrouve la notion de fonction curryfiée, sachant que dans le filtrage même une fonction curryfiée nécessitera de rassembler les arguments en un n-uplet.
Ainsi, on aura `et : bool` $\to$ `bool` $\to$ `bool`, mais `ou : (bool * bool)` $\to$ `bool` car la signature dépend de la définition et non du filtrage.
Il apparaîtra très vite que les filtrages tels quels manqueraient de puissance, mais on peut les compléter par des conditions (remplaçant avantageusement des tests conditionnels après la flèche) introduites par le mot-clé `when` suivi d'une condition.

Cette syntaxe n'est pas mentionnée dans le programme, mais au vu de son utilité, elle sera toutefois introduite progressivement et utilisée après rappels.

Attention, un nom de variable dans le filtrage est local et ne peut être utilisé qu'une fois dans un même cas de filtrage. Par exemple, les deux premiers codes ci-après sont erronés (le premier donne une fonction incorrecte, le deuxième déclenche même une erreur), mais le suivant est correct (mais inutilement lourd, il s'agit surtout d'un premier exemple illustrant la syntaxe avec `when`) :

```
let mauvais_xor b1 b2 = match b2 with
|b1 -> false (* shadowing sur b1, vu comme une nouvelle variable *)
|_ -> true (* conséquence : ce cas n'est jamais rencontré *);;

let xor_qui_plante b1 b2 = match (b1, b2) with
|(b, b) -> false (* Interdiction d'utiliser deux fois un nom *)
|_ -> true;;

let bon_xor b1 b2 = match b2 with
|a when a = b1 -> false
|_ -> true;;
```

```
(* Sans filtrage : let xor a b = not (a = b);; *)
```

Le filtrage est en fait une recherche de motifs, plutôt qu'un test d'égalité, et OCaml affecte dans la foulée les variables correspondant aux arguments quand il trouve le motif. On peut voir cela comme une version plus puissante du `switch` de certains langages.
Le fait qu'un nom de variable soit local et ne puisse être utilisé qu'une fois est dû à un souci d'optimisation de la recherche de motifs, afin qu'elle reste de complexité linéaire en temps.
En outre, si la partie à droite de la flèche contient un nouveau filtrage, un conflit de syntaxe peut être déclenché, car un nouveau motif sera compris pour OCaml comme correspondant au filtrage le plus intérieur ; un parenthésage pourra s'imposer.
Il s'avère également qu'on peut capturer plusieurs constantes à la fois dans un cas de filtrage (mais avec une gestion compliquée si des variables interviennent), il suffit de ne pas mettre de flèche à la suite des cas équivalents :

```
let voyelle carac = match int_of_char carac with
| 65 | 69 | 73 | 79 | 85 | 89
| 97 | 101 | 105 | 111 | 117 | 121 -> true
| c -> if c > 65 && c < 91 || c > 97 && c < 123 then false
       else failwith "Ceci est une façon de déclencher une erreur";;
```

♦ 2.10 Programmation impérative en OCaml

En pratique, la notion d'instruction n'existe pas vraiment en OCaml. Elle sera utilisée ici par abus, mais il est bon de garder à l'esprit que tout est expression, et ce qu'on assimilera à une instruction est à considérer comme une expression impure.

En OCaml, un morceau de code se termine usuellement par deux points-virgules, même lorsqu'on demande de calculer `2+2`.
Dans les fichiers à compiler (ou les notebooks Jupyter pour des instructions isolées), ce n'est pas une nécessité absolue, mais c'est une habitude à prendre.

♦ 2.11 Avant de commencer, les références

Une référence en OCaml permet de travailler sur une version mutable d'un objet immutable. En fait, le type d'une variable définie comme une référence est précisément… une référence du type de l'objet.
Pour fixer les idées, si on veut affecter à une variable `i` des valeurs entières dans un certain intervalle, on ne va pas affecter à `i` toutes les valeurs successivement et écrire **`let`** `i = i + 1` (ce qui provoque des erreurs de syntaxe en plein milieu d'une boucle, entre autres).
Au contraire, `i` sera une référence d'entier : on la créera par **`let`** `i = ref 1` (ce sera souvent localement), mais comme `i` sera de type `int ref`, on ne pourra pas faire de calculs sur `i` directement, il faudra déréférencer pour accéder à la valeur correspondante, en écrivant `!i` (qui sera cette fois de type `int`).

Modifier la valeur d'une référence se fait par l'opérateur `:=` dont le membre gauche est une référence d'un certain type et le membre droit une valeur du type référencé, par exemple `i := 42`, cette nouvelle valeur pouvant dépendre de l'ancienne.
Pour des références d'entiers, deux fonctions assez pratiques permettent d'additionner ou de soustraire 1 : ce sont respectivement `incr` et `decr` (pour **incr**émentation et **décr**émentation).
Bien entendu, on peut faire une référence d'à peu près tout (et même une référence de références), mais l'intérêt est limité (notamment en ce qui concerne des références de tableaux ou de chaînes de caractères, alors que les références de listes s'imaginent le temps de maîtriser suffisamment la programmation fonctionnelle).

♦ 2.12 Séquence d'instructions

Il est possible d'enchaîner deux instructions au sein d'un même morceau de code, ce qui nécessite de les séparer par un point-virgule. Cependant, OCaml déclenche un avertissement lorsqu'une instruction qui n'est pas la dernière possède une valeur, c'est-à-dire qu'elle n'est pas du type `unit` (pour le moment, les instructions de ce type déjà mentionnées sont les modifications de références et d'éléments d'un tableau).
Ceci peut s'expliquer intuitivement par le fait qu'il n'y a pas d'instruction `return` en OCaml, et donc la valeur d'une expression contenant une séquence d'instructions est nécessairement la valeur obtenue en évaluant la dernière des instructions, toutes les autres valeurs n'étant donc pas retournées, ce qui mérite d'être signalé par OCaml si ce n'est pas le comportement attendu par l'utilisateur.
Plus précisément, si on souhaite provoquer un effet de bord en se servant d'une fonction retournant une valeur à l'intérieur d'une séquence d'instructions, on peut stocker dans une sorte de variable poubelle le résultat, un peu comme en Python, en écrivant `let _ = f(x) in suite` au lieu de `f(x); suite`.[18]
Au passage, le code `let x = 2; let x = 2*x;;` provoque une erreur de syntaxe, et plus généralement faire suivre une définition d'un point-virgule simple cause des ennuis divers et variés. Une version acceptée s'obtient en remplaçant le point-virgule par `in` ou par deux points-virgules.
OCaml ne s'encombre pas de considérations quant à l'organisation du code en termes d'espaces, de tabulations et de sauts de lignes. Cependant, quelqu'un qui lit les programmes s'y intéresse, lui[19], et une bonne recommandation est **d'indenter comme si, à l'instar de Python, le fonctionnement du programme en dépendait**.

♦ 2.13 Complément sur la disjonction de cas

Bien qu'il soit autorisé de ne pas écrire la partie avec `else`, ceci sera considéré comme `else ()`, qui est de type `unit`, et la première partie doit donc être de type `unit` dans ce cas.
Cela correspond à « si cette condition est vérifiée alors faire ceci, sinon ne rien faire ».

♦ 2.14 Boucle inconditionnelle (for)

La boucle inconditionnelle s'écrit `for variable = debut to fin do bloc done`, ou (à ne pas oublier, c'est moins fréquent mais utile) de la façon suivante :

18. OCaml dispose également de la fonction `ignore` permettant aussi d'écrire `ignore(f(x)); suite`, ce qui est sans doute plus agréable à lire.
19. Ou plutôt : souhaite ne pas avoir à s'y intéresser…

`for variable = debut downto fin do bloc done`, suivant qu'on veuille incrémenter ou décrémenter la variable de boucle à chaque étape. **Les bornes sont incluses.**
Le type lors de l'évaluation de `bloc` doit toujours être `unit`, sous peine de recevoir un avertissement, tout comme dans les séquences d'instructions [20].
Si par exemple dans le premier cas `debut` est strictement supérieur à `fin`, la boucle n'est jamais exécutée. Ainsi, `for i = 3 to 2 do print_int (i / 0) done` ne provoquera pas d'erreur de division par zéro.
Puisque les seuils sont évalués une fois pour toutes avant d'entrer dans une boucle inconditionnelle, on ne peut pas perturber une boucle, ainsi :
`for i = 1 to !n do decr n done;;`, écrite dans un contexte où `n` aura été créé en tant que `ref 10`, sera effectivement parcourue dix fois (et la valeur de `!n` sera bien `0` après la boucle).
En outre, puisqu'on crée une variable `i` en donnant son nom, on ne peut pas en faire une référence, donc elle augmentera (ou diminuera) forcément d'une unité par tour de boucle (malheureusement, on ne peut pas choisir d'autre pas, et il faut alors soit créer une variable qui en dépend par une relation affine, soit utiliser une boucle conditionnelle).
En revanche, la variable de boucle est locale à celle-ci et n'a donc aucune existence en dehors si elle n'y est pas définie par ailleurs [21].
Il convient de signaler à présent que la définition d'une fonction peut faire intervenir, entre autres, des séquences ou des blocs d'instructions, et la valeur retournée est la dernière instruction rencontrée. Par exemple :

```
let fact n = let res = ref 1 in
    for i = 2 to n do res := !res * i
    (* Ici le corps de boucle, rien ne peut être retourné *)
    done;
    !res;; (* Ici la valeur de retour *)
```

♦ 2.15 Boucle conditionnelle (while)

La boucle conditionnelle s'écrit `while condition do bloc done`, où `condition` doit là aussi être un booléen et le type lors de l'évaluation de `bloc` doit toujours être `unit`. La condition est évaluée avant chaque tour dans la boucle.

♦ 2.16 Entrées et sorties

Les fonctions d'entrées et de sorties sont multiples, précisément parce qu'il en faut une par type de base.
Ainsi, pour imprimer un entier `x`, on écrira `print_int x` (de signature `int` → `unit`), mais si `x` est un flottant, on écrira `print_float x`.
Les autres fonctions d'impression classiques sont `print_char`, `print_string`, ainsi que `print_newline` (retour à la ligne, de signature `unit` → `unit`) et `print_endline` (imprime la chaîne en argument, puis retourne à la ligne).

20. Pire que cela : l'évaluation de `for i = 0 to 0 do 2+2 done;;` ne donnera même pas `4`.
21. Et même dans ce cas, on ne fait que la masquer (notion de « shadowing ») : on ne récupère pas la valeur de `i` à la dernière itération après la boucle mais la valeur de `i` en dehors de la boucle.

Remarques :

— Rien n'est prévu dans la bibliothèque standard pour les booléens, les listes, les tableaux, les n-uplets, et d'autres types exotiques.

— La fonction `print_string` imprime dans un buffer qui n'est affiché que lorsque les instructions sont terminées ou quand un saut de ligne est effectué. Voir par exemple le résultat de `print_string "plop"; print_string "\b";;` et celui de `print_string "plop\n!"; print_string "\b\b\b";;`, en signalant que le caractère spécial imprimé est le retour arrière (*backspace*).

En ce qui concerne la lecture, les fonctions suivantes attendent une saisie de l'utilisateur et les convertissent si possible : `read_int`, `read_float`, `read_line` (pour les chaînes de caractères).
Pour les fichiers, on ouvre un fichier (en tant que canal) en mode lecture ou écriture à l'aide de deux fonctions différentes : `open_in` et `open_out`, de signatures respectives `string` → `in_channel` et `string` → `out_channel`, l'argument étant le chemin (*path*) vers le fichier à ouvrir. La fermeture se fait logiquement à l'aide de `close_in` et `close_out`.
Trois canaux sont toujours ouverts (et mieux vaut éviter de les fermer) : `stdin`, `stdout` et `stderr`, qui sont l'entrée standard, la sortie standard et le canal d'erreur (de sortie, donc) standard.
La lecture et l'écriture depuis et dans un fichier se fait à l'aide des fonctions de base (mais seuls les caractères et chaînes de caractères fonctionnent), en remplaçant `read` par `input` ou `print` par `output` et en précisant en premier argument le canal. Ainsi, la fonction `print_string` est équivalente à `output_string stdout`, par exemple.

■ 3 Compléments

De nombreux passages de cette section sont en fait essentiels et sont couramment utilisés, il ne s'agit pas d'une collection de notions juste pour aller plus loin.

♦ 3.0 Définitions récursives simultanées

Les définitions peuvent aussi être simultanées, en se servant du mot-clé **`and`**[22]. Dans ce cas, si l'un des éléments dépend de l'autre, OCaml provoquera une erreur.
Par conséquent, on peut écrire `let x = 2 in let y = 4 * x and z = 4 - x;;` mais pas `let x = 2 and y = x * x;;`.
Pire que cela, on peut l'écrire si `x` existait avant, et c'est l'ancienne valeur qui est prise en compte, d'où des confusions.
Au passage, comparer les résultats des deux codes suivants :

```
let x = 42;;
let y = 19;;

let x = true and y = 12 in x;;

let x = 42;;
let y = 19;;

let x = true && y = 12 in x;;
```

22. C'est utile dans le cas de fonctions mutuellement récursives.