Chapitre 0

Introduction à l'Informatique

Notions	Commentaires
Notion de programme comme mise en oeuvre d'un algorithme. Paradigme impératif structuré, paradigme déclaratif fonctionnel, paradigme logique.	On ne présente pas de théorie générale sur les paradigmes de programmation, on se contente d'observer les paradigmes employés sur des exemples. La notion de saut inconditionnel (instruction `GOTO`) est hors programme. On mentionne le paradigme logique uniquement à l'occasion de la présentation des bases de données.
Caractère compilé ou interprété d'un langage.	Transformation d'un fichier texte source en un fichier objet puis en un fichier exécutable. […]

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Graphe de flot de contrôle. […]	[…]

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

Sommaire

Définition 1 (src : CNRTL).
« Informatique (Subt., Fem.) : Science du traitement rationnel, notamment par des machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les dommaines technique, économique et social. »

0 Algorithmes et pseudo-codes

0.0 Problèmes

Définition 2 (Problème).
Un problème est composé de deux éléments :

- une instance (aussi appelée entrée).
- une question ou une tâche à réaliser sur cette instance.

Exemple.

- PGCD :

Entrée : x, y deux entiers strictement positifs.
Tâche : calculer leur plus grand diviseur commun.

- TRI :

Entrée : des éléments comparables selon un ordre $\leqslant$.
Tâche : trier ces entiers par ordre $\leqslant$ croissant.

- PLUS COURT CHEMIN :

Entrée : une carte de réseau de transport.
un point de départ sur ce réseau, et un d'arrivée.
Tâche : calculer un plus court chemin du départ à l'arrivée.

- PUZZLE :

Entrée : un cadre et des pièces de puzzle.
Question : peux-t-on agencer les pièces dans le cadre selon les règles ?

- #PUZZLE :

Entrée : un cadre et des pièces de puzzle.
Tâche : calculer de combien de façons il est possible d'agencer légalement les pièces dans le cadre.

- SEUIL-PUZZLE :

Entrée : un cadre et des pièces de puzzle.
un entier $n \in \mathbb{N}$
Question : Existe-t-il plus de n (inclus) façons d'agencer légalement les pièces ?

On parle de problème de :

- **décision** si la réponse est binaire (Oui/Non).
- **optimisation** si la réponse est une solution "meilleure" que tout autre.
- **dénombrement** si la réponse est un nombre de solutions.
- liste non exhaustive.

Exemple. Associer un type (ou plusieurs) à chacun des problèmes précédents.

0.1 Algorithmes

Le terme « algorithme » a de nombreux sens. La définition que j'en donne ici est celle que nous utiliserons cette année ; mais ce n'est forcément pas celle du langage courant [1]. Cela ne signifie pas que l'une des deux est meilleure que l'autre : un même mot a des sens différents dans des contextes différents, c'est normal, et cela nous arrivera même entre différentes sections d'un même chapitre.

Définition 3 (Algorithme).
Un algorithme est la description finie d'une suite d'opérations résolvant un problème. On veut de plus avoir :

- Terminaison : la suite d'opération décrite est atteint toujours sa fin. On dit que l'algorithme termine.
- Précisément définie : les opérations sont définies précisément, rigoureusement et sans ambiguïté.
- Entrée : un algorithme a 0 ou plusieurs quantités de nature connue et spécifiée qui lui sont données avant ou pendant son exécution.
- Sortie : un algorithme renvoie une ou plusieurs quantités dont le lien avec les entrées est spécifié.
- Élémentarité : les opérations décrites doivent être assez simples pour être réalisées par une personne avec un papier et un crayon en temps fini.

Remarque.

- On rapproche souvent les algorithmes des recettes de cuisine.
- Aucune notion de langage informatique n'est présente dans cette définition. Langage naturel [2], idéogrammes, schéma, pseudo-code, Python, C, OCaml, SQL : tout pourrait convenir.
- Attention à ne pas confondre la terminaison avec le fait que la description elle-même est finie.
- Le caractère précisément défini demande que l'on sache *quoi* faire, alors que l'élémentaire demande que l'on sache *comment* le faire.
- L'élémentarité dépend du public-cible. « Diviser un entier par un autre » est une étape élémentaire pour vous, pas pour des élèves de CE2. De même pour la précision : elle peut dépendre du contexte.
- Cette définition d'algorithme ne contient aucun critère d'efficacité en temps (ou en espace mémoire).

Exemple. L'algorithme d'Euclide pour résoudre PGCD est un exemple connu. Dedans, « assigner $x \leftarrow y$ » signifie « écraser la valeur de x en la remplaçant par celle de y ».

Algorithme 1 : Algorithme d'Euclide.

Entrées : m et n deux entiers strictement positifs
Sorties : $pgcd(m, n)$

1 E1 (calcul du reste) : diviser m par n, et noter r le reste
2 E2 (égal à 0 ?) : si $r = 0$, renvoyer n
3 E3 (réduction) : assigner $m \leftarrow n$, $n \leftarrow r$. Recommencer en E1.

1. Il me semble que de plus en plus, le terme a une connotation d'obscur, d'inconnu, de traitement caché. Ce n'est pas le contexte dans lequel je me place dans ce cours. Notez que je n'affirme pas ladite connotation est absurde.
2. Le langage naturel est la langue des humain·es.

On peut représenter cet algortihme sous une forme visuelle :

FIGURE 1 – Graphe de Flot de Controle de l'algorithme 1.

Exercice. Vérifier qu'il s'agit bien d'un algorithme. Le faire tourner sur l'instance $m = 15, y = 12$.

0.2 Un langage de pseudo-code

Pour utiliser une machine automatique, on doit transformer l'algorithme en une suite d'instruction que la machine comprend. La première et principale étape consiste en générale à écrire cet algorithme dans un *langage de programmation*. Ceux-ci ont de nombreuses différences, mais aussi énormément de concepts communs. On parle de **pseudo-code** pour décrire quelque chose qui ressemble à des langages de programmation, manie ces concepts communs, mais limite les détails techniques propres à chaque langage.

Le pseudo-code que je vais présenter ici est très **impératif**, c'est à dire que l'on programme en décrivant des modifications successives de la mémoire à effectuer.

0.2.0 Variables et sauts conditionnels

Définition 4 (Variable, version intuitive).
Une variable peut-être vue comme une boîte qui a :

- un identifiant, aussi appelé nom de variable.
- un contenu. Celui-ci peut-être par exemple :
  - une valeur précise indiquée.
  - le résultat d'un calcul.
  - la valeur renvoyée par un autre algorithme sur une entrée précise.

On note $n \leftarrow \alpha$ le fait que la variable n contienne dorénavant le contenu α.

Exemple.

$x \leftarrow 3$	on stocke 3 dans x
$x \leftarrow (9-4)$	x contient dorénavent 5 et non plus 3
$y \leftarrow 2x$	on stocke 10 dans y
$x \leftarrow y^3$	x contient dorénavent 1000 et non plus 5.

Définition 5 (Saut conditionnel).
Un saut conditionnel permet d'exécuter différentes instructions selon une valeur logique, souvent le résultat d'un test. On note :

```
si valeur logique alors
| instructionsDuAlors
sinon
| instructionsDuSinon
```

Pseudo-code G.F.C associé.

Les instructions du Alors s'effectuent si la valeur logique s'évalue à Vrai, celles du Sinon… sinon.
Ces instructions sotn respectivement nommées le **corps du Alors/Sinon**.
Lorsque le corps du Sinon est vide, on n'écrit tout simplement pas le Sinon (et son corps).

Exemple.

```
1 x ← 3
2 si x ≥ 2 alors
3 |  x ← 3x
4 |  x ← x − 1
5 sinon
6 |  x ← 1.5x
7 |  x ← −2x
```

À la fin de l'exécution de ce code, x contient 8.

La notion de conditionnel est central en informatique. C'est elle qui permet de faire en sorte qu'un programme puisse avoir des comportements différents en fonction des informations qui lui sont donnés ! Imaginez un moteur de recherche internet qui renvoie toujours les mêmes résultats, peu importe l'entrée…

Remarque.

- Les barres verticales sont très utiles à la relecture. Je vous encourage *très très fortement* à les mettre dès que vous êtes sur papier, y compris lorsque vous écrivez du vrai code.
- En anglais, Si/Alors/Sinon se dit If/Then/Else.
- Par abus du langage, on parle souvent du « corps du Si » pour désigner le corps du Alors.
- Un saut conditionnel est lui-même une instruction, et on peut donc les imbriquer :

Résumé grossier du fonctionnement de M. Domenech aux portes ouvertes.

```
si l'élève veut faire 12h de maths alors
|  si l'élève aime ou veut découvrir l'informatique alors
|  |  orienter en MP2I
|  sinon
|  |  orienter en MPSI
sinon
|  orienter en PCSI (10h de maths quand même)
```

Notez que 10h vs 12h de maths n'est pas un excellent critère. Dans le groupe nominal « résumé grossier », il se pourrait que le qualificatif soit plus important que le substantif.

- Considérez l'imbrication suivante : « Si b_0 Alors Si b_1 Alors $instr_0$ Sinon $instr_1$ ». Ici, il n'est pas possible de savoir de quel Si $instr_1$ est le Sinon. On parle du problème du *Sinon pendant* (*Dangling else* d'Alan Turing) : il y a une ambiguïté sur l'imbrication.

  Sur papier, une écriture avec retour à la ligne, indentation (décalage horizontal du corps) et barre verticale permet de l'éviter.

  Les langages de programmation, eux, ont chacun une règle qui force une unique façon d'interpréter cette imbrication. Il s'agit généralement soit d'indiquer explicitement la fin du saut conditionnel, soit de la règle du parenthèsage à gauche d'abord ; nous en reparlerons.
- Les sauts inconditionnels (« goto ») sont hors-programme et prohibés.

0.2.1 Fonctions

Lorsque vous avez découvert les fonctions en maths, on vous les a possiblement présenté comme des machines qui prennent quelque chose en entrée et le transforme en autre chose (en termes savants, on dit qu'elles associent un antécédant à une image). C'est la même idée en informatique.

Définition 6 (Fonction, version intuitive).
Une **fonction** est transforme des **arguments** en une nouvelle valeur. On dit aussi qu'elle transforme des **entrées** en **sorties**.
En pratique, une fonction est un bloc de pseudo-code qui attend certaines variables en entrée, décrit des modifications à faire dessus, et indique une valeur à renvoyer à la fin. On dit que les instructions contenues dans la fonction forment le **corps** de la fonction.
On dit qu'une fonction est **appelée** lorsqu'on l'utilise pour calculer la sortie associée à une entrée. On dit qu'elle **renvoie** sa sortie. On note généralement comme en maths : $f(x_0, x_1, \dots)$ est la valeur renvoyée par la fonction f sur l'entrée $(x_0, x_1, \dots)$.

Exemple. Voici deux fonctions. Que font-elles ?

Entrées : x une variable contenant un entier.
Sorties : L'entier 2.

```
renvoyer 2
```

Entrées : x une variable contenant un entier.
Sorties : L'entier x^2.

```
y ← x
renvoyer x * y
```

Exercice. Proposer une autre façon d'écrire le corps de cette seconde fonction qui évite la création d'une variable dans la fonction.

Notez que la notion de corps induit une notion de « monde extérieur » : on distingue le (pseudo-)code qui se trouve dans la fonction du (pseudo-)code qui se trouve ailleurs.

Exemple. Indiquer ce que vaut `main()` avec les fonctions ci-dessous :

Entrées : x une variable contenant un entier.
Sorties : y un entier

```
y ← 2x
x ← 0
renvoyer y
```

Fonction `fun`

Entrées : Rien.
Sorties : Mystère.

```
x ← 3
z ← fun(x)
renvoyer x
```

Fonction `main`

Il y a deux réponses possibles : 0 (car `fun` a remis x à 0) et 3 (car `fun` ne travaille pas sur le x d'origine mais sur une copie). Cela correspond respectivement aux deux notions suivantes :

Définition 7 (Passages et effets secondaires).
Soit `F` une fonction et `x` un de ses arguments. On dit que `x` est :

- **appelé par référence** si toute modification de `x` par le corps de `F` modifie également le `x` initial (la variable qui avait été donnée en entrée à `F` lors de son appel). Tout se comporte comme si le corps de la fonction et le monde extérieur partagent le même `x`.
- **appelé par valeur** si, au contraire, toute modification de `x` par le corps de `F` *ne* modifie *pas* le `x` initial. Tout se comporte comme si le corps de la fonction manipule une copie du contenu `x`, distincte du `x` du monde extérieur.

Sauf mention contraire explicite dans ce cours, tous les arguments sont appelés par valeur. La mention contraire principale concernera les tableaux.

En particulier, dans l'exercice précédent, la réponse que donnerait tout langage de programmation raisonnable [3] est 3.

3. Et toute étudiant·e de MP2I raisonnable ;)

Définition 8 (Effets secondaires, version courte).
L'ensemble des modifications effectuées par une fonction sur son monde extérieur est appelé les **effets secondaires** de la fonction.

Remarque.

- On parle aussi de **passage par référence/valeur**. Nous verrons également le **passage par pointeur**, qui à notre niveau ne sera pas distinct du passage par référence.
- Le passage par valeur est couteux : l'appel de la fonction doit consommer du temps et de l'espace mémoire afin de créer une copie de l'argument concerné. Les langages de programmation réservent donc généralement le passage par valeur aux variables de taille raisonnable (comme des entiers) et passent tout ce qui est potentiellement gros (comme une collection de valeurs plus simples) par référence.

  Par exemple, Python passe les entiers par valeur et les listes par référence.
- L'exemple de `fun` et `main` montre également un exemple où une fonction en appelle une autre. cela nous arrivera très frequemment. **Le rôle principal des fonctions est de décomposer et structurer un gros code en petites unités élémentaires simples à comprendre.**

  Nous verrons également qu'il peut-être très comode pour une fonction de s'appeler elle-même, ce que l'on nomme la **récursion**.

Définition 9 (Prototype, signature).

- La donnée de l'identifiant (le nom) d'une fonction ainsi que de ses types d'entrées et de sortie est appelée la **signature** de la fonction.
- Si l'on ajoute les identifiants des entrées, on parle de **prototype**.

Exemple.

- Signature de PGCD : `PGCD : entier , entier →entier`
- Prototype de PGCD : `PGCD :` m `entier,` n `entier →entier`

En mathématiques, on note $pgcd : \mathbb{N}^* \times \mathbb{N}^* \to \mathbb{N}^*$. C'est la signature !

0.2.2 Boucles

De même que l'informatique sans Si/Alors/Sinon n'est pas très intéressante, il nous manque pour l'instant une capacité très importante : pouvoir répéter des instructions un nombre inconnu de fois. C'est par exemple ce que faisait l'algorithme d'algorithme grâce à son étape E3 qui indiquait « retourner en E1 ». On dit que E1-E2-E3 forment une boucle.

Définition 10 (Boucle à précondition).
Une boucle à précondition, ou boucle TantQue (« While ») répète des instructions tant qu'une valeur logique (souvent une condition) est vraie :

tant que *valeur logique* **faire**
| instructions

Pseudo-code

G.F.C associé.

Les instructions répétées sont appelées le **corps**. On dit qu'on effectue une **itération** de la boucle lorsque l'on exécute les opérations du corps.

Exemple.

Entrées : x un entier.

```
1 p ← 0
2 tant que x < 436 et p < 10 faire
3 |   x ← 2x
4 |   p ← p + 1
5 renvoyer x
```

Fonction `demo`

Pour calculer à la main `demo(1)`, on peut écrire un tableau où on décrit le contenu des variables à la fin de chaque itération :

	p	x
Avant la première iter.	0	1
Fin de la première iter.	1	2
Fin de la seconde.	2	4
Fin de etc	3	8
Fin de etc	4	16
Fin de etc	5	32
Fin de etc	6	64
Fin de etc	7	128
Fin de etc	8	256
Fin de etc	9	512

Après cette dernière itération, on quitte la boucle car $x \geq 436$ et donc la valeur logique d'entrée dans la boucle devient fausse.

Remarque.

- Si la valeur logique du TantQue reste éternellement vraie, on ne quitte jamais la boucle et on effectue de sitérations à l'infini. On parle de **boucle infinie**. C'est un des bugs les plus courants.
- Plusieurs fois dans l'année, nous referons des tableaux d'itération comme celui-ci. Nous noterons avec des primes (p' et x') les quantités en fin d'itération afin d'éviter des confusions.
- On peut tout à fait faire le tableau avec les quantités en début d'itération. Cela évite la ligne « avant la première itération », mais demande souvent de rajouter une ligne à la fin « début de "l'itération qui n'a pas lieu car on quitte la boucle" ».

  Dans ce cours et toute cette année, je travaillerai plutôt avec les fins.

Exercice. Calculer l'image de 14 par la fonction suivante :

Fonction Collatz

Entrées : x un entier strictement positif.
Sorties : Mystère mystère.

```
1 tant que x > 1 faire
2 |   si x est pair alors
3 |   |   x ← x/2
4 |   sinon
5 |   |   x ← 3x + 1
6 renvoyer x
```

CultureG : les valeurs successives de x durant ces itérations forment ce que l'on appelle une suite de Syracuse [4] [5]. Il est conjecturé et presque prouvé [6] que toutes les suites de Collatz contiennent tôt ou tard l'entier 1.

4. Pas Syracuse comme la ville grecque, voyons. Syracuse comme l'université de Syracuse, bien sûr. Qui tient son nom de la ville de Syracuse, New-York (l'état, pas la ville), États-Unis. Évidemment.

5. L. Collatz, lui, est un mathématicien ayant participé à populariser l'étude de ces suites.

6. « Presque » au sens où T. Tao a prouvé que cette conjecture est vraie pour presque tous les entiers. La définition de « presque tous » est subtile. Retenez qu'en pratique elle est vraie.

Définition 11 (Boucle itérative).
Une boucle itérative, ou boucle Pour (« For »), permet de répéter des instructions un nombre connu de fois tout en mémorisant le numéro de la répétition en cours. C'est en fait un simple raccourci pour une boucle TantQue :

pour *compteur allant de 0 inclus à* n *exclu* **faire**
  instructionsDuCorps

Pseudo-code d'un For

$compteur \leftarrow 0$
tant que $compteur < n$ **faire**
  instructionsDuCorps
  $compteur \leftarrow compteur + 1$

Pseudo-code du While équivalent

G.F.C. associé.

On parle là aussi du **corps** de la boucle.

Remarque.

- Notez que la version While et la version G.F.C. explicitent toutes deux le fait que **la mise à jour du compteur a lieu à chaque itération, en toute fin d'itération**.
- On croise souvent les boucles PourChaque, qui permettent d'effecter le corps de la boucle sur chacun des éléments d'une collection d'objets (e.g. « PourChaque x entier de $[\![8; 35]\!]$). C'est par exemple ce que fait `for x in ...` de Python.

Définition 12 (break, continue).
On crée deux nouvelles instructions pour les boucles : `break` permet de quitter instantanément la boucle en cours, et `continue` permet de passer immédiatement à l'itération suivante (en ayant mis à jour le compteur).
Ces deux instructions ne s'appliquent *que* à la boucle en cours (et pas aux éventuelles sur-boucles en cas d'imbrication).

Attention, ces deux instructions sont une erreur commune de bugs des débutant·es. La plus « dangereuse » des deux, `continue`, est hors-programme pour cette raison.

0.3 Convention de représentation des G.F.C.

Le flot d'un code désigne l'enchainement de ses instructions. Un graphe de flot de controle est une représentation du déroulement de l'exécution (pseudo-)code. Le but est de représenter quelle instruction peut mener à quelle instruction, c'est à dire de représenter le flot.

Dans les schémas précédents, on a utilisé les conventions suivantes :

- Le début du flot est marqué par un triangle (base en bas), et par une double flèche y menant. Au départ de la flèche, on indique les variables d'entrées.
- Les fins du flot sont marquées par un triangle (base en haut), et une double flèche en sortant. À la fin de la flèche, on indique la sortie / valeur renvoyée par cette fin du flot.

- Les instructions sont écrites dans des rectangles. On peut utiliser un même rectangle pour tout un corps.
- Le passage d'un point à un autre est représenté par une flèche simple.
- Les embranchements (sauts conditionnels, entrée/sortie de boucle) sont représentés par un "losange", coin vers le bas. Dans le losange on indique la valeur logique qui est interrogée. Sur les flèches sortantes, on indique Vrai/Faux (ou Oui/Non) pour indiquer quelle flèche correspond à quelle option.
- On représente les appels de fonction par un cercle contenant le non de la fonction. On détaille éventuellement le GFC de la fonction à part.

Exemple. Cf G.F.C. de l'algorithme d'Euclide.

0.4 Tableaux

On a parfois besoin de stocker une très grande quantité de données, au point où créer une variable par donnée ne serait pas pratique. Par exemple, si je veux stocker la note d'anglais au bac de chaque élève de CPGE scientifique de Camille Guérin, il me faudrait $48 \times 5 = 240$ variables.

Définition 13 (Tableaux, version intuitive).
On peut penser un tableau comme un long meuble muni de L tiroirs numérotés. Chacun des tiroirs est une variable.
Un peu plus formellement, un **tableau** est défini par sa **longueur**, souvent notée L, qui est son nombre de **cases**. Chaque case se comporte comme une variable. On peut accéder à chaque case à l'aide de son indice, compris entre 0 et $L-1$ inclus : si `T` est un tableau et `i` un indice valide, on note `T[i]` la case d'indice `i` de `T`.
Le contenu de toutes les cases d'un même tableau doit avoir la même nature (tous des entiers, ou bien tous des nombres à virgule, etc).
Dans ce cours, si un tableau à L cases contient dans l'ordre les valeurs $x_0, x_1, ..., x_{L-1}$, on le notera en pseudo-code `[|`x_0`;`x_1`;...;`x_{L-1}`|]`.

Lors des appels de fonction, un tableau se comporte comme si son contenu était passé par référence.

Exemple.

```
1 T ← tableau de longueur 3
2 T[0] ← 42
3 T[1] ← -20
4 T[2] ← T[0] + T[1]
```

À la fin de l'exécution, T est `[|42; -20; 22|]`.

Exemple. La fonction `mainbis()` ci-dessous est la fonction constante égale à 0, car les cases du tableau se comportent comme si passées par référence :

Entrées : T un tableau d'entiers.

```
1 y ← 2 × T[0]
2 T[0] ←0
3 renvoyer y
```

Fonction `funbis`

Entrées : Rien.

```
1 T ←tableau d'entiers non-vide
2 z ←funbis(T)
3 renvoyer T[0]
```

Fonction `mainbis`

Remarque.

- La longueur d'un tableau est fixe. Les tableaux ne sont *pas* redimmensionnables.
  En particulier, les listes Python ne sont pas des tableaux. C'est une structure plus complexe, qui permet plus de choses mais est un peu plus lentes : ce sont des tableaux dynamiques (de pointeurs), que nous verrons comment construire cette année.
- Très souvent, nous serons amenés à traiter les unes après les autres les cases d'un tableau ; on dit qu'on **parcourt** le tableau. Les boucles sont l'outil adapté pour cela :

```
1 i ←0
2 tant que i < L faire
3 |   traiter T[i]
4 |   i ←i+1
```

Avec une boucle While

```
1 pour i allant de 0 inclus à L exclu faire
2 |   traiter T[i]
```

Avec une boucle For

- Le contenu des cases d'un tableau `T` peut tout à fait être des tableaux : dans ce cas, la case `T[i]` de `T` est elle-même un tableau, et on peut donc en demander la case `j`. On demande donc `(T[i])[j]`, que l'on préfère écrire `T[i][j]`.

1 Machines, programmes et langages

1.0 Machines, modèles

Définition 14 (Ordinateur (src : wikipedia.fr)).
Un ordinateur est un système de traitement de l'information programmable.

Exemple.

- L'ordinateur PC-prof de la salle B004. C'est le sens commun du terme « ordinateur », et c'est sur ces ordinateurs-ci que nous apprendrons à programmer.
- Vos téléphones, consoles de jeu.
- Les télévisions, voitures, vidéoprojecteurs, etc modernes.
- Un métier Jacquart.
- Certains jeux, comme Minecraft ou BabaIsYou (on peut simuler un ordinateur dedans).
- Les machines à laver modernes (si si).

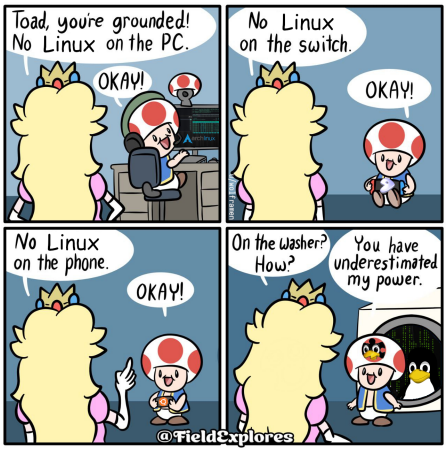

FIGURE 2 – Inspiré de faits réels.

Définition 15 (Modèle de calcul).
Un modèle de calcul est un ensemble de règles qui permettent de simuler les étapes d'un calcul. C'est souvent une abstraction d'une machine de calcul (e.g. un ordinateur) du monde réel, afin de prouver des propriétés sur la machine et son fonctionnement.

Remarque. Le modèle est une *simplification*. Par exemple, on lui suppose souvent une mémoire infinie [7].

7. Et moi qui me trouve chanceux avec mes $2 * 8\times$ Go de RAM, alors que je pourrais viser l'infini...

Exemple.

- Machine à RAM : représente un ordinateur muni d'une mémoire vive (RAM) infinie.
- Machine de Turing : idem, sauf que la mémoire est un ruban qu'il faut dérouler pour aller d'un point à un autre de la mémoire. Ce modèle représente les premiers ordinateurs modernes et a eu un très fort impact théorique et historique.
- Lambda-calcul : a inspiré la famille des langages de programmation fonctionnels. Très théorique, ne correspond pas à une machine réelle.

1.1 Programmes

Voici un schéma (très) simplifié des ordinateurs modernes [8] :

FIGURE 3 – B-A-BA de l'architecture d'un ordinateur.

Définition 16 (Programme.).
Un **programme**, aussi appelé exécutable, est une suite d'instructions à effectuer par le processeur. Ces instructions sont écrites en **langage machine**.

Remarque. Un compilateur ne peut lire *que* du langage machine, qui est une suite de 0 et de 1 !

Exemple. Ce sont des opérations très basiques. Voici un exemple "lisible" par des humains, où l'on a "traduit" les suites de 0 et de 1 du langage machine en mots-clefs (on parle de **langage assembleur**) :

```
mov r3, #17          stocke 17 dans la case mémoire r3
add r3, r3, #42      r3 ←r3 + 42
```

Le langage assembleur, et plus encore le langage machine, est très dur à lire et permet assez peu d'abstraction. C'est pour cela que nous ne programmerons pas en assembleur mais dans des langages de programmation, pensés pour être plus proches des humains.

Définition 17 (Langage de programmation).
Un langage de programmation est défini ensemble de règles syntaxiques propres. L'objectif est de rédiger des programmes de manière plus lisible que le langage machine. « Coder dans un langage de programmation » signifie « écrire un texte qui respecte les règles du langage. »

8. Non-contractuel, j'ai vraiment simplifié comme un bourrin.

Définition 18 (Compilateur).
Pour que le fichier écrit dans un langage de programmation devienne un programme, il faut le traduire en langage machine : un tel traducteur est appelé un **compilateur**.
Lorsque l'on traduit d'un langage de programmation vers un autre langage de programmation (et non vers le langage machine), on dit que l'on **transpile** (pour distinguer de compile).

Exemple.

- C, OCaml, Rust, C++, Haskell, Fortran, etc sont des
- Le langage machine est techniquement un langage de programmation (c'est même le seul qui n'a pas à être compilé).
- L'assembleur (la traduction est assez directe, il suffit [9] de traduire mot à mot en 0-1).
- On considère souvent que les langages comme le HTML, qui ne servent pas à exprimer des calculs mais à décrire un agencement, ne sont pas des langages de programmation. Pour HTML, on parle de langage de balisage.

  Cela ne signifie pas que travailler sur ces langages est moins "noble" ou plus simple. Cela signifie juste que ce n'est pas la même chose car le but (et les moyens offerts) est radicalement différent.

Remarque.

- Pour un même langage, différents compilateurs peuvent exister. Ils peuvent proposer des façons de traduire différentes, qui optimisent des critères distincts.

  Par exemple, pour C, il en existe au moins 3 : `gcc` (que nous utiliserons), `clang` et `compcert`.
- Un compilateur peut faire des modifications à votre code afin de l'accélérer. Par exemple, si un compilateur détecte qu'une boucle ne sert qu'à calculer $\sum_{i=1}^{n} i$, il peut supprimer la boucle et la remplacer par $\frac{n(n+1)}{2}$.
- Différents processeurs ont des langages machines différents. Si l'on veut distribuer un logiciel, il faut donc en prévoir une version compilée par langage machine que l'on veut pouvoir supporter. C'est pour cela que vous trouvez par exemple des versions `arm` et des version `x86` des logiciels sur leurs pages de téléchargement (en réalité il faut aussi une version par système d'exploitation : différents Linux, MacOS, Windows, etc).
- Une fois un code compilé, il n'est plus nécessaire d'avoir le compilateur (ou le fichier source du code) pour lancer le programme : seule compte la version compilée.

Définition 19 (Interpréteur).
Un interpréteur est un programme qui lit du code écrit dans un langage de programmation et l'exécute au fur et à mesure de sa lecture.

Exemple. Python, Java, OCaml (qui dispose d'un compilateur *et* d'un interpréteur).

Remarque.

- Le code n'est donc jamais traduit en un programme exécutable et a besoin de l'interpréteur pour fonctionner.
- Le fait que les lignes sont lues les unes et interprétées après les autres permet beaucoup moins d'optimisations qu'un compilateur (qui lit tout le fichier avant de compiler). Un code interprété est donc typiquement plus long qu'un code compilé (et consomme souvent plus de mémoire car il faut faire tourner l'interpréteur en plus du code).
- L'avantage est que l'on peut exécuter uniquement petit morceau du code grâce à l'interpréteur. Cela permet de tester un morceau choisi en ignorant les erreurs qui pourraient se trouver ailleurs. À l'inverse, un compilateur refusera de compiler un fichier tant qu'il reste un non-respect des règles du langage dedans.

9. Modulo simplification pédagogique.

- Un autre avantage est qu'il n'y a pas besoin de recompiler pour chaque langage processeur ! Il faut juste avoir accès à un interpréteur (lui-même un programme compilé en général) pour ce langage processeur.
- Différents interpréteurs peuvent exister pour un même langage. Python en a 3, chacun écrits dans un langage différent. Vous avez probablement utilisé sans le savoir `cpython`, codé en C, qui est l'interpréteur le plus courant.
- On parle de **langage interprété** (respectiveent **langage compilé**) pour désigner un langage qui dipose d'un compilateur (respectivement interpréteur).

1.2 Paradigmes

Définition 20 (Paradigmes de programmation).
Il existe différentes façons de penser la programmation :

- **paradigme impératif** : programmer, c'est décrire une suite de modifications de la mémoire.
- **paradigme fonctionnel** : programmer, c'est décrire une le calcul à effectuer comme une suite d'évaluation de fonctions mathématiques.
- **paradigme logique** : programmer, c'est décrire très précisément le résultat attendu à l'aide d'un ensemble de formules logiques.

Un langage peut correspondre à différents paradigmes.

Ce sont des concepts qui se comprennent mieux en pratiquant.

Exemple.

- Pseudo-code ci-dessus, C, Python, Rust, C++, OCaml : paradigme impératif.
- OCaml, Haskell, Erlang, Elixir : paradigme fonctionnel.
- Prolog, un peu SQL : paradigme logique.