Calculabilité - Décidabilité

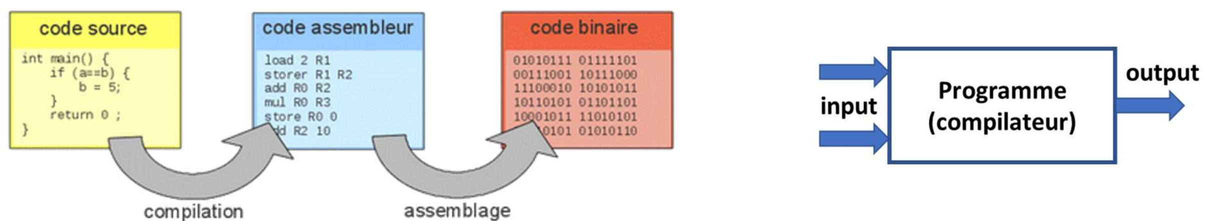
1. Programme en tant que donnée

La plupart des programmes attendent en argument des données (input) qu'ils traitent pour produire en sortie un résultat (output).

```python
def prog(t):
    m = t[0]
    for e in t:
        if e < m:
            m = e
    return m
```

Certains programmes utilisent comme données le code source d'autres programmes.

- Les compilateurs sont des bons exemples. Par exemple le langage C qui est un langage compilé : une fois le code source terminée, le compilateur (qui est un logiciel comme un autre) "transforme" ce code source en langage machine.



- Un système d'exploitation (Linux, Windows…) est un programme qui exécute d'autres programmes (traitement de textes…).
- Pour télécharger un logiciel, un gestionnaire de téléchargement est utilisé. Ce gestionnaire de téléchargement est lui-même un logiciel.

En conclusion, un programme peut être utilisé comme une donnée par un autre programme.

2. Calculabilité

La notion de calculabilité date de 1936, il s'agit de savoir ce qui peut être calculé par un ordinateur, et donc permet de voir les **limites des problèmes** que peuvent résoudre les ordinateurs.

On dira qu'une fonction est **calculable** si elle peut être programmée dans l'un ou l'autre des langages de programmation usuels, comme par exemple en langage Python. Une fonction est calculable si on peut la programmer en Python.

On peut calculer beaucoup de choses avec un ordinateur comme le nombre π, les nombres rationnels √2, √3…

Par contre, il a été prouvé que certains problèmes n'étaient pas calculables comme par exemple :

- Savoir si un énoncé mathématique est un théorème ou pas *(s'il peut être démontré)*.
- Créer un programme qui prend un programme en entrée, et qui indiquera si le programme s'arrête ou pas : le problème de l'arrêt.

Il s'agit de problèmes de **décidabilité**.

(cc) BY-NC-SA

3. Décidabilité

Un problème de décision est dit décidable s'il existe un algorithme, une procédure mécanique qui se termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème. S'il n'existe pas de tels algorithmes, le problème est dit indécidable. Par exemple, le problème de l'arrêt est indécidable.

3.1. Exemples de problèmes décidables

Tous les sous-ensembles finis des entiers sont décidables, par exemple :

- Décider si un entier naturel est pair ou non.
- Décider si un entier naturel est premier ou non.

Décidable ne signifie pas résolvable

Notons qu'un ensemble peut être théoriquement décidable sans qu'en pratique la décision puisse être faite, parce que celle-ci nécessiterait trop de temps (plus que l'âge de l'univers) ou trop de ressources (plus que le nombre d'atomes de l'univers). L'objet de la théorie de la complexité des algorithmes est d'étudier les problèmes de décision en prenant en compte ressources et temps de calcul.

3.2. Exemple de problème indécidable : le problème de l'arrêt

L'indécidabilité du problème de l'arrêt a été démontrée par Alan Turing en 1936.
On peut l'interpréter ainsi : il n'existe pas de programme permettant de tester n'importe quel programme informatique afin de conclure dans tous les cas s'il s'arrêtera en un temps fini ou bouclera à jamais.

Preuve par l'absurde de non-décidabilité de l'arrêt

Supposons qu'il existe une fonction calculable **`termine(programme, données)`** qui prend 2 arguments :

- un programme
- des données d'entrée pour ce programme

et qui renvoit **`True`** si le programme termine et **`False`** s'il entre dans une boucle infinie.

Exemple

```
def est_positif(n):
    if n >= 0:
        return True
    else:
        while n < 0:
            n = n - 1 # boucle infinie
        return False
```

La fonction **`est_positif(n)`** entraine une boucle infinie pour les nombres négatifs, et on a **`termine(est_positif, 128)`** renvoie **`True`** alors que **`termine(est_positif, -2)`** renvoie **`False`** non pas car -2 n'est pas positif mais parce que l'appel **`est_positif(-2)`** ne se termine jamais.

Soit une fonction **`test_sur_soi`**.

```
def test_sur_soi(programme):
    if termine(programme, programme):
        while True: pass # boucle infinie
```

On obtient alors une contradiction si on appelle **`test_sur_soi`** sur elle-même : **`test_sur_soi(test_sur_soi)`**

```
# l'appel exécute l'algorithme suivant
if termine(test_sur_soi, test_sur_soi):
    while True: pass
```

On arrive au paradoxe suivant :
`test_sur_soi(test_sur_soi)` termine ⟺ **`test_sur_soi(test_sur_soi)`** boucle indéfiniment.

(cc) BY-NC-SA