Chapitre 21 - Calculabilité, décidabilité

Objectifs :

- ▷ Comprendre que tout programme est aussi une donnée.
- ▷ Comprendre que la calculabilité ne dépend pas du langage de programmation utilisé.
- ▷ Montrer, sans formalisme théorique, que le problème de l'arrêt est indécidable.

1 Introduction

Comme nous l'avons vu en classe de *Première*, un programme est la traduction électronique d'un algorithme afin qu'il puisse être compris par une machine. Dans ce chapitre, nous allons montrer qu'un programme ne peut pas tout **calculer** ou **décider**.

L'**indécidabilité** est un concept fondamental en logique. Il a plusieurs sens. Au sens de la démonstration, il signifie qu'on ne peut démontrer ni qu'une proposition est vraie ni qu'elle est fausse dans un système d'axiomes ; ici, le contenu sémantique ne permet d'affirmer ni que l'objet est une pipe ni qu'il n'en est pas une. Au sens de la **calculabilité**, il signifie qu'il n'existe aucun algorithme capable de résoudre un problème (ici déterminer si l'objet est une pipe ou non).

2 Notion de programme en tant que donnée

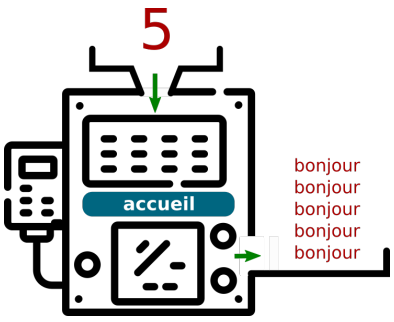
Les codes que nous manipulons ressemblent souvent à cela :

```
1 def accueil(n):
2     for k in range(n):
3         print("bonjour")
```

Le programme s'appelle `accueil`, et pour fonctionner il a besoin d'un paramètre, qui est un nombre entier `n`.

Voici comment nous pouvons représenter notre machine `accueil` :

- son paramètre d'entrée :`5`
- sa sortie : `bonjour` affiché 5 fois



Maintenant, enregistrons le code suivant dans un fichier `test.py` :

```python
def accueil(n):
    for k in range(n):
        print("bonjour")

accueil(5)
```

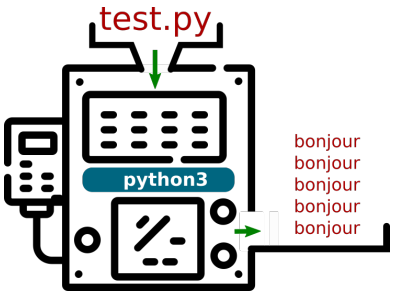
Pour exécuter ce code, nous devons taper dans un terminal l'instruction suivante : `python test.py`

Ce qui donne :

```
C:\Users\trahobisoa\Downloads>python test.py
bonjour
bonjour
bonjour
bonjour
bonjour
```

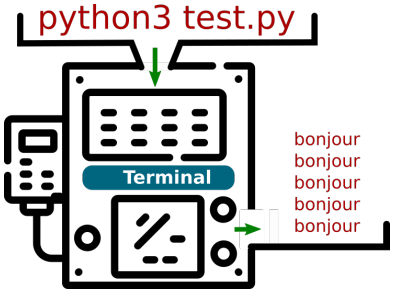
Le programme utilisé est alors `python`, qui prend comme paramètre le programme `test.py`. Ce paramètre `test.py` est un ensemble de caractères qui contient les instructions que le programme `python` va interpréter.

L'illustration correspondante sera donc :



Mais nous pouvons aller encore plus loin. L'instruction `python test.py` est saisie dans mon `Terminal Windows`, qui lui-même est un programme appelé `Terminal`.

Et donc :



Conclusion :

Il n'y a donc aucun obstacle à considérer **un programme comme une simple donnée**, pouvant être reçue en paramètre par un autre programme. (voire par lui-même !)

Ainsi, certains programmes utilisent comme données le code source d'autres programmes.

Exemples :

- un système d'exploitation(comme *Linux* ou *Windows*) est un programme qui éxecute d'autres programmes (traitement de texte, tableur, ..., *etc.*).
- l'interpréteur *Python*, est un programme qui traduit le code source de votre programme Python en instructions exécutables par la machine : du langage machine (cf. cours de *Première* sur l'architecture de *Von Neumann* et le langage assembleur).

3 Décidabilité

3.1 Mon programme va-t-il s'arrêter ?

Considérons le programme suivant :

```
def countdown(n):
    while n != 0:
        print(n)
        n = n - 1
    print("fini")
```

En l'observant attentivement, on peut prévoir que `countdown(10)` affichera les nombres de `10` à `1` avant d'écrire `"fini"`. Puis le programme s'arrêtera.

Mais que va provoquer `countdown(10.8)` ?

Comme la variable `n` ne sera jamais égale à `0`, le programme va rentrer dans une **boucle infinie**, il ne s'arrêtera jamais ! On a pu prévoir ceci en analysant attentivement le code du programme. On a ainsi remarqué qu'une variable `n` non entière provoquerait une boucle infinie.

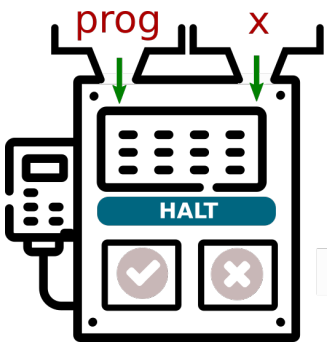
Question : Est-ce qu'un programme d'analyse de programmes aurait pu faire cela à ma place ?

3.2 Une machine pour prédire l'arrêt ou non d'un programme

Après tout, un programme est une suite d'instructions (le code source), et peut donc être le paramètre d'entrée d'un autre programme qui l'analyserait. Un tel programme (appelons-le `halt`) prendrait en entrées :

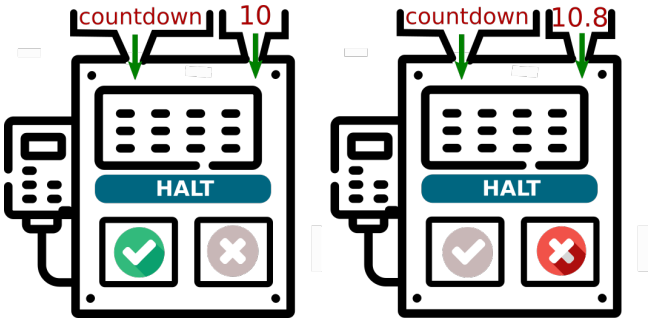
- un paramètre `prog` (le code-source du programme)
- un paramètre `x`, qui serait le paramètre d'entrée de `prog`.

L'instruction `halt(prog,x)` renverrait `True` si `prog(x)` s'arrête, et `False` si `prog(x)` ne s'arrête pas.



Exemples :

- `halt(countdown,10)` renverrait `True`.
- `halt(countdown,10.8)` renverrait `False`.



Tentative d'écriture de `halt` en Python :

```
def halt(prog, x):
    if "prog(x) s'arrete":    # code a determiner
        return True
    else :
        return False
```

Nous en resterons là pour l'instant dans l'écriture de ce programme. Nous allons nous en servir pour construire d'autres programmes.

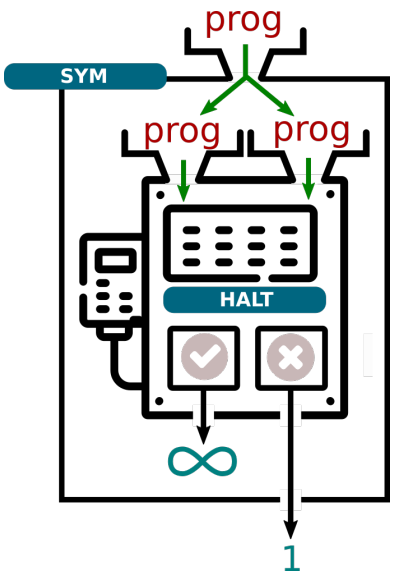
3.3 Le problème de l'arrêt

Considérons le programme suivant :

```
def sym(prog):

    if halt(prog, prog) == True:
        while True:
            print("vers l'infini et au-dela !")
    else:
        return 1
```

On peut remarquer que le programme `halt` est appelé avec comme paramètres `(prog,prog)`, ce qui signifie que `prog` se prend lui-même en paramètre. On rappelle que ce n'est pas choquant, un code source étant une donnée comme une autre.



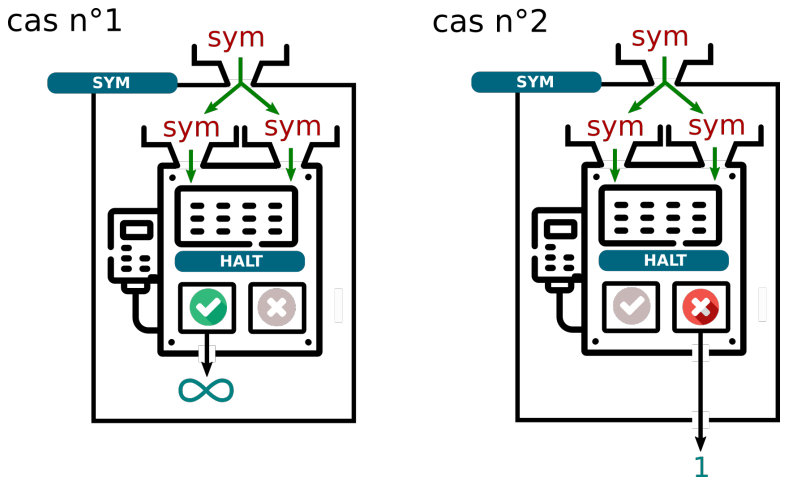
Ce programme `sym` reçoit donc en paramètre un programme `prog`, et :

- va rentrer dans une boucle infinie si `prog(prog)` s'arrête.
- va renvoyer `1` si `prog(prog)` ne s'arrête pas.

Problème :

Puisqu'un programme peut prendre en paramètre son propre code-source, que donnerait l'appel à `sym(sym)` ?

Deux cas peuvent se présenter, suivant si `halt(sym, sym)` renvoie `True` ou `False`.



— **cas n°1 :** `halt(sym, sym)` renvoie `True`, ce qui signifie que `sym(sym)` devrait s'arrêter. Mais dans ce cas-là, l'exécution de `sym(sym)` rentre dans une boucle infinie. C'est une **contradiction** !

— **cas n°2 :** `halt(sym, sym)` renvoie `False`, ce qui signifie que `sym(sym)` rentre dans une boucle infinie. Mais dans ce cas-là, l'exécution de `sym(sym)` se termine correctement et renvoie la valeur `1`. C'est une **contradiction** !

3.4 Conclusion

Nous venons de prouver que notre programme `halt`, censé prédire si un programme `prog` peut s'arrêter sur une entrée `x`, **ne peut pas exister !**

Ce résultat théorique, d'une importance cruciale, s'appelle le **problème de l'arrêt**.

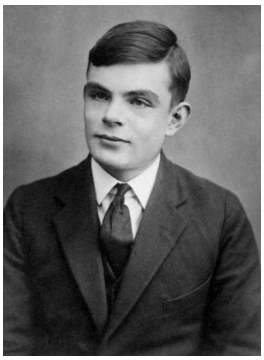
Problème de l'arrêt

Il ne peut pas exister de programme universel qui prendrait en entrées :

— un programme `P`

— une entrée `E` de ce programme `P`

et qui déterminerait si ce programme `P`, lancé avec l'entrée `E`, va s'arrêter ou non.



Ce résultat a été démontré par *Alan Turing* en 1936, dans un article intitulé *«On computable numbers, with an application to the Entscheidungsproblem»*.

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means.

Pour sa démonstration, il présente un modèle théorique de machine capable d'exécuter des instructions basiques sur un ruban infini, les **machines de Turing**.

À la même époque, le mathématicien *Alonzo Church* démontre lui aussi ce **théorème de l'arrêt**, mais par un moyen totalement différent, en inventant le **lambda-calcul**.

Tous deux mettent ainsi un terme au rêve du mathématicien allemand *David Hilbert*, qui avait en 1928 posé la question de l'existence d'un algorithme capable de répondre «oui» ou «non» à n'importe quel énoncé mathématique posé sous forme décisionnelle (*«un triangle rectangle peut-il être isocèle ?»*, *«existe-t-il un nombre premier pair ?»*).

Cette question, appelée **problème de la décision**, ou *Entscheidungsproblem* en allemand, est définitivement tranchée par le **problème de l'arrêt** : un tel théorème ne peut pas exister, puisque par exemple, aucun algorithme ne peut répondre «oui» ou «non» à la question *«ce programme va-t-il s'arrêter ?»*.

Définition

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.

4 Calculabilité

Le problème de l'arrêt est dit **indécidable** car la fonction qui le résout (notre programme `halt`) n'est pas **calculable**.

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.

Propriété

On dira qu'une fonction est **calculable** si elle peut être programmée dans l'un ou l'autre des langages de programmation usuels.

En *Première* et *Terminale*, nous utilisons le langage Python comme témoin : une fonction est calculable si on peut la programmer en Python.

Il existe d'autres modèles de calcul, comme le λ-calcul, les fonctions récursives, les machines de Turing, que nous ne développerons pas ici, et qui ne font pas partie des attendus du programme.

La thèse de *Church* postule que tous ces modèles de calcul sont équivalents : une fonction calculable pour un modèle l'est pour un autre. Cela nous permet d'utiliser le modèle des fonctions programmables en Python sans perdre de généralité.

On peut calculer beaucoup de choses avec un ordinateur comme le nombre π, les nombres rationnels, $\sqrt{2}$, $\sqrt{3}$, ..., *etc.*

Cependant, 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**.

En résumé, voir la vidéo suivante

5 Exercices

Exercice 1 : *Vocabulaire*

1. Donner deux exemples qui montrent pourquoi un programme est aussi une donnée.
2. Quelle est la différence entre la calculabilité et la décidabilité d'un problème ?
3. Faire des recherches et donner des exemples de problèmes **décidables**.
4. Faire des recherches e dtonner des exemples de problèmes **indécidables**.

Exercice 2 : *Racine carrée*

On définit ci-dessous une fonction `racine_carree` :

```
1
2  def racine_carree(n, precision=1E-2):
3      """ Recherche d'une racine carree par une methode dichotomique
4      Parametres :
5      ------------
6      n: float
7      le nombre dont on recherche la racine
8      precision: float 0.01 par defaut
9      precision du calcul du carre
10
11     Retour :
12     --------
13     float : la racine carree de n
14     """
15     gauche, droite, milieu = 0, n, n
16     while abs(milieu ** 2 - n) > precision :
17         milieu = (gauche + droite) / 2
18         if milieu ** 2 - n > 0:
19             droite = milieu - precision
20         else:
21             gauche = milieu + precision
22     return milieu
```

1. Expliquer pourquoi il est nécessaire d'utiliser une précision dans ce calcul ?
2. Expliquer la ligne : `while abs(milieu ** 2 - n) > precision :`
3. Pourquoi peut-on qualifier cet algorithme de dichotomique ?

Exercice 3 : *Entiers*

1. Implémenter en Python deux fonctions :
   - ▷ `est_pair(n)` : renvoie `True` si le nombre entier n est pair, `False` sinon.
   - ▷ `est_premier(n)` : renvoie `True` si le nombre entier n est premier, `False` sinon.
2. Tester ces fonctions avec un entier négatif et un nombre décimal. Que se passe-t-il ?
3. Tester ces fonctions avec des petites entrées, puis avec des grandes pour voir si ces algorithmes seraient utilisables en pratique.

Exercice 4 : *Non-décidabilité de l'arrêt*

1. Pourquoi dit-on que le problème de l'arrêt est indécidable ?

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

   — une fonction,
   — et des données d'entrée pour cette fonction

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

On définit les deux fonctions suivantes :

```
def fonction1(n):
    if n % 3 != 0:
        return True
    else:
        return False

def fonction2(n):
    while n % 3 != 0:
        print("True")
    print("False")
```

2. Que renverront les appels ci-dessous ?

```
termine(fonction1,7)
termine(fonction1,9)
termine(fonction2,7)
termine(fonction2,9)
```

Justifier vos réponses.

3. On définit une fonction `test_sur_soi`.

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

Que se passe-t-il si on appelle `test_sur_soi` sur elle-même : `test_sur_soi(test_sur_soi)` ?