1 Programme en tant que donnée

Les théories de la calculabilité et de la complexité sont deux pans de la théorie du calcul :

Qu'est-ce que veut dire calculer ? Un ordinateur peut-il tout calculer ?

C'est en 1936 que Alan Turing (1912-1954) a apporté des réponses à ces questions, et la conclusion de ses travaux... va peut-être vous étonner :)

Nous allons tout d'abord expliciter un point important qui sera le fondement de la théorie de la calculabilité : **un programme est aussi une donnée**.

Cela peut paraître étonnant à première vue puisqu'on est habitué à traiter :
- les programmes dans des fonctions.
- les données dans des variables.

Fonctions et variables sont des objets de nature différente en apparence : si on se raccroche à ce que l'on connaît en Python, une fonction se déclare avec le mot clé `def` et une variable s'initialise avec l'opérateur d'affectation =.

Prenons en exemple l'algorithme d'Euclide, un fameux algorithme vieux de plus de 2500 ans permettant de calculer le PGCD de 2 nombres. On peut l'écrire à l'aide d'une fonction Python :

```
def euclide(a, b):
    while b:              # s'arrête quand b vaut 0
        a, b = b, a%b
    return a
```

Dans ce programme Python, `euclide` est une fonction et `a` et `b` sont des données. Ils ne semblent pas être de nature comparable :

```
>>> euclide(35, 49)
7
```

Et pourtant, à y regarder de plus près, l'algorithme programmé dans la fonction `euclide` n'est rien d'autre qu'une succession de caractères. On peut même pousser la réflexion jusqu'à créer une chaîne de caractères contenant ce programme :

```
programme = """
def euclide(a,b):\n\tif a < b: a,b=b,a\n\twhile b: a,b=b,a%b\n\treturn a
"""
```

Maintenant l'algorithme est devenu une variable. On peut alors construire une machine universelle capable d'évaluer n'importe quelle donnée contenant un algorithme formalisé dans le langage Python :

```
def universel(algo, *args):          # *args = arguments optionnels
                                     # args est alors un tuple d'arguments
    exec(algo)                       # exécution dynamique de code Python
    nom = algo.split('(')[0][4:]     # extrait le nom de la fonction
    return eval(f"{nom}{args}")      # évaluation de l'expression Python
```

À présent on peut invoquer cette machine universelle en lui passant en données :
- la variable contenant l'algorithme
- les arguments sur lequel celui-ci va travailler, et obtenir la réponse

```
>>> universel(programme, 35, 49)
7
```

Dans l'exemple ci-dessus, vous pouvez constater que le programme et les données sur lesquelles il agit sont de même nature : ce sont 3 variables passées en paramètres à la fonction `universel`.

Conclusion :

Tout programme est aussi une donnée.

 Citer d'autres exemples où un programme est considéré comme donnée.

Quelques réponses possibles : Les programmes qui manipulent d'autres programmes en tant que données sont courants. On peut citer :
- Les **compilateurs** qui prennent en entrée un texte et le transforment en une suite de 0 et 1 exécutable par le microprocesseur de l'ordinateur.
- L'**interpréteur** Python fait de même, c'est d'ailleurs lui que nous avons mis à contribution dans la machine universelle.
- Le **navigateur** Web : il reçoit un lot de données d'internet via le protocole HTTP et interprète certaines d'entre elles comme programme (JavaScript) et d'autres comme données (HTML) etc...
- Lors du **téléchargement d'un logiciel** en ligne, celui-ci est traité comme une simple donnée.
- Le **système d'exploitation** utilise les logiciels comme des données.

2 Calculabilité

La **calculabilité** est la branche de l'algorithmique qui s'intéresse aux questions suivantes :
- Peut-on tout calculer à l'aide d'un ordinateur ?
- Que signifie calculer à l'aide d'un ordinateur ?

La théorie de la calculabilité décrit et caractérise les problèmes qui peuvent être résolus par un algorithme (on parle de problème décidable ou calculable), et répertorie ceux qui ne le peuvent pas.

Puisqu'un programme est aussi une donnée, il est donc manipulable par des algorithmes.
Peut-on concevoir un algorithme permettant de savoir si un programme passé en paramètre :
- va se terminer ? c'est à dire renvoyer un résultat.
- va provoquer une erreur ?
- est correct et ne contient pas de bugs ?

Ce sont des questions fondamentales au cœur de l'algorithmique et de l'informatique en général. Imaginez un programme capable de dire si un autre programme est correct ou bugué ! Vous commencez à sentir que cela ne va pas être possible, hélas ...

Théorème fondamental de la calculabilité :

> Il existe des problèmes non calculables par des fonctions.

> La non calculabilité de certains problèmes est un résultat **fondamental** et reste totalement indépendant d'un quelconque langage de programmation.

3 Le problème de l'arrêt

3.1 Description du problème

Le premier problème explicite non calculable a été décrit par Turing en 1936 : il s'agit du **problème de l'arrêt** qui s'énonce ainsi :
Étant donné un algorithme A et prenant en entrée un paramètre m, existe t-il un algorithme permettant de décider si $A(m)$ s'arrête ou pas ?

C'est une question cruciale et pourtant, elle est **indécidable**.

> Il n'existe aucun moyen de savoir en général si un algorithme quelconque va s'arrêter ou non.

3.2 Preuve de Turing

Pour prouver que le problème de l'arrêt n'est pas calculable, Turing a proposé ce raisonnement par l'absurde :
Supposons qu'il existe une fonction **arrêt** qui prend en paramètres un algorithme **A** et des arguments **m** (on se rappelle qu'un algorithme est une donnée).
Cela pourrait prendre cette forme en Python :

```
def arret(A, m):
    """ Renvoie True si l'algorithme A s'arrête avec la donnée m """
    ... # Tout est ici !
    if ...:
        return True
    else:
        return False
```

Le vrai problème est bien sûr de compléter les pointillés de cette fonction `arret`, mais on suppose ici que quelqu'un a su le faire.
On peut alors créer un paradoxe à l'aide de l'algorithme suivant :

```
def paradoxe(truc):
    if arret(truc, truc): # s'il y a bien arrêt,
        while True:
            pass          # on crée une boucle infinie ...
    else:
        return True       # sinon, on arrête cette fonction
```

Que donne l'appel `paradoxe(paradoxe)` (on cherche à savoir si paradoxe s'arrête) ?

- Si l'algorithme paradoxe s'arrête avec la donnée paradoxe : `arret(paradoxe, paradoxe)` vaut True, et alors `paradoxe(paradoxe)` ne s'arrête pas car on rentre dans la boucle infinie => contradiction !
- Si au contraire l'algorithme paradoxe ne s'arrête pas avec la donnée paradoxe : `arret(paradoxe, paradoxe)` vaut False, et dans ce cas `paradoxe(paradoxe)` s'arrête car le return True met fin à la fonction => contradiction !

Cette contradiction montre donc que l'hypothèse de départ « *il existe une fonction arret* » est impossible. **le problème de l'arrêt est indécidable**.

Cette indécidabilité se retrouve dans la majorité des problèmes en algorithmique.

Retrouvez cette preuve dans cette vidéo (en anglais, facile à comprendre) :
`https://youtu.be/92WHN-pAFCs`

3.3 Conclusion

La détection de bugs (comme des boucles infinies par exemple) est une activité cruciale en informatique. Il existe des programmes capables de détecter des erreurs dans d'autres programmes : les environnements de développement modernes comme VScode ou Spyder sont capables de souligner vos erreurs en Python alors même que vous tapez le programme, et cela constitue un grand gain de temps pour le développeur.

En France, depuis l'accident du vol 501 d'Ariane 5 en 1996 dû à une erreur de programmation, le développement de programmes de **preuves de correction** a connu une forte croissance.

Un bel exemple de progrès réalisé grâce à ces programmes est le logiciel de la ligne de métro automatique 14 (Météor) à Paris : cette ligne a été mise en service sans test en condition réelles : **son programme a été mathématiquement prouvé sans fautes**. Depuis sa mise en service, aucun bug n'a été à déplorer.

Malheureusement, un algorithme général permettant de prouver qu'un programme fonctionne n'existe pas, cela fait partie des très nombreux problèmes indécidables. Il n'y aura donc jamais de système permettant de s'assurer que n'importe quel programme est fiable, même si c'est réalisable pour quelques exemples particuliers comme le Météor.