Résumé

Programme en tant que donnée

Un programme (du texte binaire que la machine peut exécuter) est le produit d'un compilateur (qui transforme le code source en fichier exécutable).

On peut ensuite télécharger ce programme, le déplacer etc. avec différents logiciels.

Ce programme est donc *une donnée* de ces logiciels (compilateur, navigateur, explorateur de fichiers etc.).

Langage Turing complet

Un langage est *Turing-Complet* s'il peut *calculer* la même chose qu'une machine de Turing universelle.

Tous les langages de programmation modernes sont Turing-Complets. Python est donc Turing-Complet.

Ainsi, on peut *tout* programmer en Python. Ce n'est pas toujours un bonne idée... parce que Python est lent et n'est pas adapté à certains problèmes (pilotes matériel notamment).

Décidabilité

Un problème de décision est un problème pour lequel on peut répondre par Oui ou Non selon une donnée en entrée.

Exemple : un entier n est-il premier ?

Exemple : cette chaîne de caractères est-elle un code python sans erreur de syntaxe ?

Un problème de décision est *décidable* si on peut programmer une fonction Python qui réponde pour chaque donnée d'entrée.

Pour les deux exemples précédent c'est le cas, ils sont décidables.

Calculabilité

Une fonction est *calculable* si on peut la programmer dans un langage Turing-Complet comme Python.

Cette notion est similaire à celle de problème de décision décidable :

Un problème de décision est décidable si son prédicat est calculable.

Problème de l'arrêt

Peut-on écrire une fonction Python qui, pour n'importe quelle fonction en entrée et n'importe quelle donnée de cette fonction, réponde Vrai si le programme termine (=s'arrête ou plante) ou Faux si elle ne termine pas ?

La réponse est non, le problème de l'arrêt est *indécidable*.

Preuve par l'absurde

Supposons que quelqu'un ait réussi à programmer cette fonction `arret`.

```
def arret(f, x) -> bool:
    """
    Prend une fonction f et ses paramètres x en entrée.
    * Termine toujours
    * Renvoie `True` si `f(x)` termine
    * Renvoie `False` si `f(x)` ne termine pas.
    """
    ...
```

La fonction `strange`

Nous pouvons alors l'utiliser pour programmer la fonction `strange` suivante :

```
def strange(f, x):
  if arret(f, x):
    while True:
      pass
```

Cette fonction teste si le calcul de `f(x)` termine.

- si `f(x)` termine, elle rentre dans une boucle infinie dans laquelle elle ne fait rien.
- Sinon, elle ne fait rien et termine.

Autrement dit, `strange(f, x)` termine si et seulement si `f(x)` ne termine pas.

La fonction `paradox`

La précédente fonction nous permet d'en définir une nouvelle, la fonction `paradox`.

```
def paradox(f):
  strange(f, f)
```

Par construction, le calcul de `paradox(f)` termine si et seulement si le calcul de `f(f)` ne termine pas. Appelons (1) cette propriété.

> (1) *le calcul de `paradox(f)` termine si et seulement si le calcul de `f(f)` ne termine pas*

`paradox(paradox)` ?

Maintenant, demandons-nous si le calcul de `paradox(paradox)` termine.

Pour cela, dans la propriété (1), remplaçons `f` par `paradox`.

La propriété devient :

> *Le calcul de paradox(paradox) termine si et seulement si le calcul de paradox(paradox) ne termine pas.*

Cette dernière propriété est évidemment absurde (elle se contredit elle même).

Nous pouvons donc conclure notre raisonnement par l'absurde et affirmer qu'il est impossible de programmer la fonction `arret`.

Machine de Turing

Une machine de Turing est un modèle théorique d'ordinateur.

Dans sa version la plus simple on dispose :

- d'un ruban infini contenant des cases où sont écrits des symboles 0 ou 1 ou rien,
- d'une tête de lecture / écriture qui peut se déplacer à gauche ou à droite d'une case à la fois,
- d'un automate qui selon un *état* (parmi un nombre fini) et ce qu'on vient de lire décide : d'écrire un symbole, d'aller à droite ou à gauche et de changer d'état.

Cette construction est suffisante pour *tout programmer*.

Historique et conséquences

Turing démontre en 1936 qu'il n'existe pas de machine de Turing résolvant le problème de l'arrêt. Au passage il fournit un des premiers modèles qui servira de base aux futurs ordinateurs.

Ce résultat est la conséquence de milliers d'années de recherche mais surtout des développement de Hilbert vers 1900, Gödel en 1929-1930 puis Church, Turing et Kleene à la fin des années 1930.

Les mathématiciens sont maintenant confrontés à un problème insoluble : cette conjecture est-elle vraie (et on peut fournir une preuve), fausse (un contre exemple) ou indécidable (on cherchera indéfiniment pour rien).

Les informaticiens sont aussi bien heureux d'avoir des ordinateurs (merci Turing et les autres)... mais ils savent que certains problèmes n'auront jamais de solution (problème de l'arrêt, assistant de preuve parfait, ramasse miette parfait etc.).