Option Informatique - MPSI

Option Informatique
Introduction historique

Florent Pompigne
pompigne@crans.org

Lycée Buffon

année 2025/2026

Différents langages de programmation

Langage Python

- **Paradigme dominant** : impératif / itératif (boucles)
- **Points forts** :
  - Simplicité de programmation
  - bibliothèques scientifiques optimisées

Langage Ocaml

- **Paradigme dominant** : fonctionnel / récursif
- **Points forts** :
  - exécution rapide pour un langage de haut niveau
  - typage statique renforçant la sûreté du code
  - création de types adaptée à l'algorithmique avancée

Factorielle en Python et en Ocaml

Langage Python

```
def fact(n):
    r = 1
    for i in range(1,n+1):
        r *= i
    return r
```

décrit comment calculer la factorielle

Langage Ocaml

```
let rec fact n =
    if n = 0 then 1 else n * (fact (n-1))
;;
```

décrit ce qu'est la factorielle

Origines de la calculabilité

Au début du vingtième siècle, David Hilbert pousse pour surmonter la crise des fondements des Mathématiques ("Wir mussen wissen. Wir werden wissen."). Objectifs :

- Démontrer la cohérence des théories infinies dans l'arithmétique élémentaire (réfuté par Gödel en 1931).
- Résoudre le problème de la décision (Entscheidungsproblem, 1928) : déterminer algorithmiquement si un énoncé est démontrable ou non.

La théorie des cardinaux de Cantor permet d'observer qu'il existe des fonctions entières non calculables par un algorithme, mais pour savoir si c'est le cas du problème de la décision, il faut définir formellement ces notions d'algorithme et de fonction calculable.
Alan Turing et Alonzo Church vont donner deux définitions très différentes de la calculabilité pendant les années 1930.

Machines de Turing

Objet mathématique inspiré d'une machine mécanique :

- Un ensemble fini d'états, dont un état initial et un état final de la machine ;
- un ruban infini à gauche et à droite formé de cases contenant des symboles ;
- une tête de lecture pointant sur une case du ruban ;
- une table de transition indiquant pour chaque état et chaque symbole dans la case pointée, dans quel nouvel état passer, quel nouveau symbole écrire dans la case pointée, et dans quelle direction déplacer d'un cran la tête.

Pour exécuter une machine de Turing sur un ruban contenant l'entrée de l'algorithme, on applique la table de transition jusqu'à tomber sur l'état final. Le ruban contient alors le résultat.

Exemples

On utilise les symboles _ et I, et on note les entiers en unaire, par exemple 5 serait noté ...__IIIII__.... On suppose que la tête de lecture est initialement sur le premier I et qu'on commence dans l'état 0.
La table de transition suivante calcule le successeur d'un entier :

	0
I	0,I, +
_	F,I, -

Déterminer une table de transition pour calculer la somme de deux entiers *n* et *m* séparés par un _ , puis une table pour calculer leur produit. On pourra utiliser autant d'états et de symboles que nécessaire.

Exemples

On utilise les symboles _ et I, et on note les entiers en unaire, par exemple 5 serait noté . . . __IIIII__. . . . On suppose que la tête de lecture est initialement sur le premier I et qu'on commence dans l'état 0.
La table de transition suivante calcule le successeur d'un entier :

	0
I	0,I, +
_	F,I, -

Déterminer une table de transition pour calculer la somme de deux entiers *n* et *m* séparés par un _ , puis une table pour calculer leur produit. On pourra utiliser autant d'états et de symboles que nécessaire.
Indication : Faire apparaître une boucle respectant l'invariant "À l'étape k, on a $n - k$ I, puis un _, puis mk ○, puis m I, avec la tête de lecture tout à gauche".

Solution du produit

	0	1	2	3	4	5	6	7	8
I	1,_,+	2,I,+	2,I,+	4,○,+	4,I,+	6,○,+	6,I,+	7,I,-	8,I,-
_		F,_,+	3,_,+		5,I,-	8,_,-	7,I,-		0,_,+
○				3,○,+		5,○,-	6,○,+	5,○,-	

Il resterait à la fin à remplacer chaque ○ restant par un I, étape facile qu'on a omise pour ne pas alourdir la table.

Lambda-calcul de Church

Un terme du λ-calcul est défini récursivement comme :

- une variable, par exemple $x, y, z, \ldots$;
- ou une application tu, où t et u sont eux-mêmes des termes (intuitivement : le résultat de l'application de la fonction t sur l'argument u) ;
- ou une λ-abstraction $\lambda x.t$, où x est une variable et t est un terme (intuitivement : la fonction qui à x associe $t(x)$).

La seule règle de calcul est la β-réduction : si on applique une λ-abstraction sur un terme, on peut remplacer l'argument formel par l'argument effectif :

$$(\lambda x.t)u \ \rightarrow_\beta \ t[x \leftarrow u]$$

où $t[x \leftarrow u]$ est le terme obtenu en remplaçant chaque variable x par le terme u dans t.

Exemples

$$(\lambda x.\lambda z.xz)(\lambda x.yx)z \;\to_\beta\; (\lambda z.(\lambda x.yx)z)z \;\to_\beta\; (\lambda z.yz)z \;\to_\beta\; yz$$

On code un entier n comme $[n] := \lambda f.\lambda x.f(f(\ldots fx)\ldots)$, avec n applications de f.
Par exemple, $[3] = \lambda f.\lambda x.f(f(fx))$, et $[0] = \lambda f.\lambda x.x$.
La fonction successeur peut alors se définir comme :

$$s := \lambda n.\lambda f.\lambda x.nf(fx)$$

On a bien $s[n]$ qui donne $[n+1]$ par β-réductions :
$s[2] = (\lambda n.\lambda f.\lambda x.nf(fx))[2] \to_\beta \lambda f.\lambda x.[2]f(fx)$
$= \lambda f.\lambda x.(\lambda f.\lambda x.f(fx))f(fx) \to_\beta \lambda f.\lambda x.(\lambda x.f(fx))(fx)$
$\to_\beta \lambda f.\lambda x.f(f(fx)) = [3]$
Déterminer sur le même principe une fonction réalisant la somme et une fonction réalisant le produit de deux entiers.

Solutions pour la somme et le produit

$$+ := \lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$$

$$* := \lambda n.\lambda m.\lambda f.\lambda x.n(mf)x$$

Alternatives :

$$+ := \lambda n.\lambda m.nsm$$

$$* := \lambda n.\lambda m.n(+m)[0]$$

Le fichier `church.py` contient des solutions pour coder la puissance, le prédécesseur et la soustraction.

Comparaison des modèles de calcul

- Dans les machines de Turing, il y a une différence de nature entre les **instructions** de la table de transition, qui modifient l'état du système, et les **données** écrites sur le ruban. Dans le λ-calcul, il n'y a que des termes fonctionnels.
- Dans les machines de Turing, si on veut répéter des étapes, il faut faire apparaître des boucles. Dans le λ-calcul, il n'y a pas de notion de boucle, on peut en revanche créer de la récursivité.
- L'approche des machines de Turing, d'inspiration mécanique, inspirera les premiers ordinateurs (WWII) et leurs langages de programmation (paradigme impératif / itératif). l'approche du λ-calcul inspirera l'aspect fonctionnel des langages de programmation plus modernes, permettant la récursivité et la manipulation de fonctions.
- Python et Ocaml sont multiparadigmes. On peut écrire des instructions et des boucles en Ocaml, et des fonctions et des appels récursifs en Python !

Thèse de Church-Turing

Church et Turing montrent chacun l'indécidabilité du problème de la décision de Hilbert.
De façon remarquable, leurs modèles de calcul définissent exactement le même ensemble de fonctions calculables.
Aujourd'hui encore, tout langage Turing-complet, comme Python ou Ocaml, permet de programmer les mêmes fonctions que l'on pourrait coder en machines de Turing ou en λ-calcul. D'où :

Thèse de Church-Turing (1937)

La définition formelle complète de la notion d'algorithme est celle donnée indifféremment par les machines de Turing, le λ-calcul ou tout langage Turing-complet.
Formulation physique : tout processus physique peut être exprimé dans un tel langage.