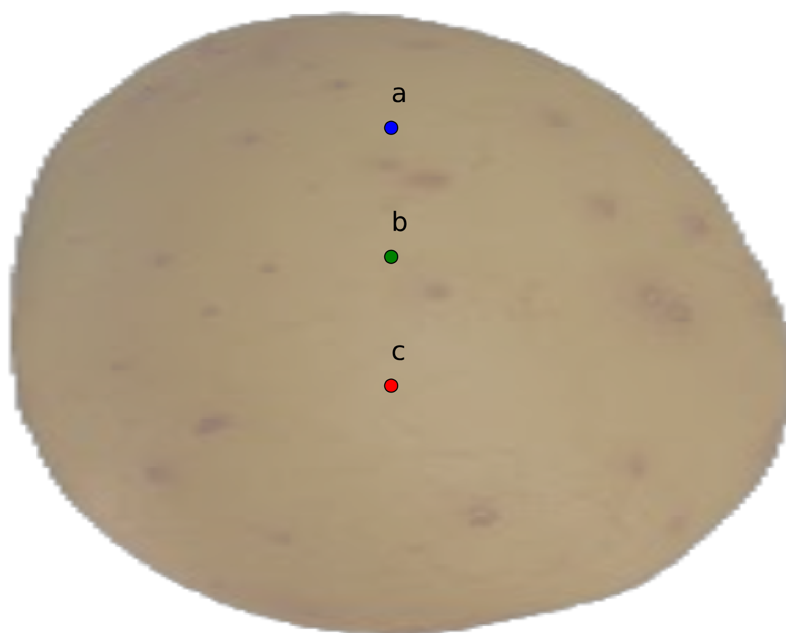
Théorie de Scott et Strachey

La théorie élaborée en 1971 par Christopher Strachey[1] et Dana Scott[2] avait pour but de formaliser les preuves de programmes[3] en créant une sémantique[4] des programmes. Elle nous intéresse ici parce que pour *étudier* du point de vue mathématique les programmes informatique, il a fallu préalablement que Scott et Strachey *définissent* un programme informatique.

1. Ordinateur

Un ordinateur (ou « machine ») est un ensemble de variables, désignées par leur nom, et en nombre fini. Ici ce seront trois variables numériques appelées respectivement a, b et c :



1 Auteur du premier programme de jeu de dames et du premier programme de musique par ordinateur, puis membre du groupe Algol, ce camarade de promotion d'Alan Turing était une immense figure de l'informatique, et pas seulement théorique.

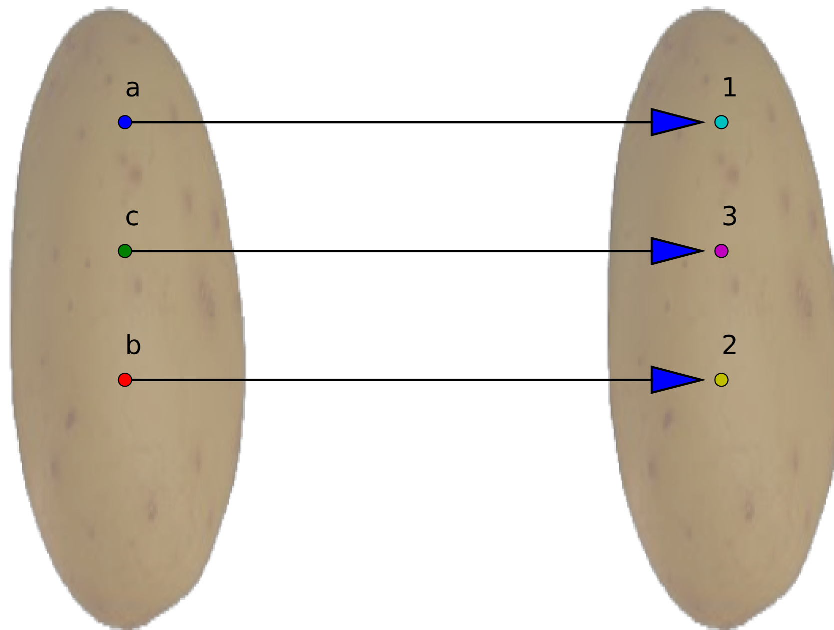
2 Logicien de formation, il est connu pour ses travaux avec Rabin sur la théorie des langages Regex), avec Lemmon sur la logique temporelle (logique modale), avec Montague en linguistique (λ-calcul) et avec Strachey sur la sémantique des programmes.

3 Les preuves de programmes sont des démonstrations mathématiques, qu'un programme fait effectivement ce pour quoi il a été programmé. Par exemple, on peut se servir de la logique pour prouver que l'algorithme d'Euclide calcule bel et bien le pgcd de deux nombres, et pas autre chose. Elles sont à distinguer de l'algorithmique (qui s'intéresse au temps mis par le programme pour accomplir son travail) et de la théorie de la calculabilité (qui se demande si ce temps est fini).

4 La syntaxe des programmes est l'objet de la théorie des langages, elle étudie à quelles conditions un programme est correctement rédigé. La sémantique des programmes est nécessaire pour prouver des assertions sur le programme, en étudiant la signification des opérations élémentaires du programme. En logique, la sémantique étudie la valeur de vérité d'une proposition et la cohérence d'une théorie, la syntaxe étudie la démontrabilité.

2. États de la machine

Un état de l'ordinateur est une fonction qui, à chaque variable, associe une valeur (ici, numérique). Par exemple, si la variable a contient 1, la variable b contient 2 et la variable c contient 3, on a l'état suivant :



Remarque : On peut obtenir cet état avec l'algorithme suivant (« initialisation ») :

```
a ← 1
b ← 2
c ← 3
```

Scott utilise aussi un autre mot que « état » : Il appelle parfois ces fonctions des « affectations » : Une affectation associe à chaque variable une valeur. Mais il est plus habituel aujourd'hui d'appeler affectation l'opération qui aboutit à un nouvel état, plutôt que l'état final lui-même.

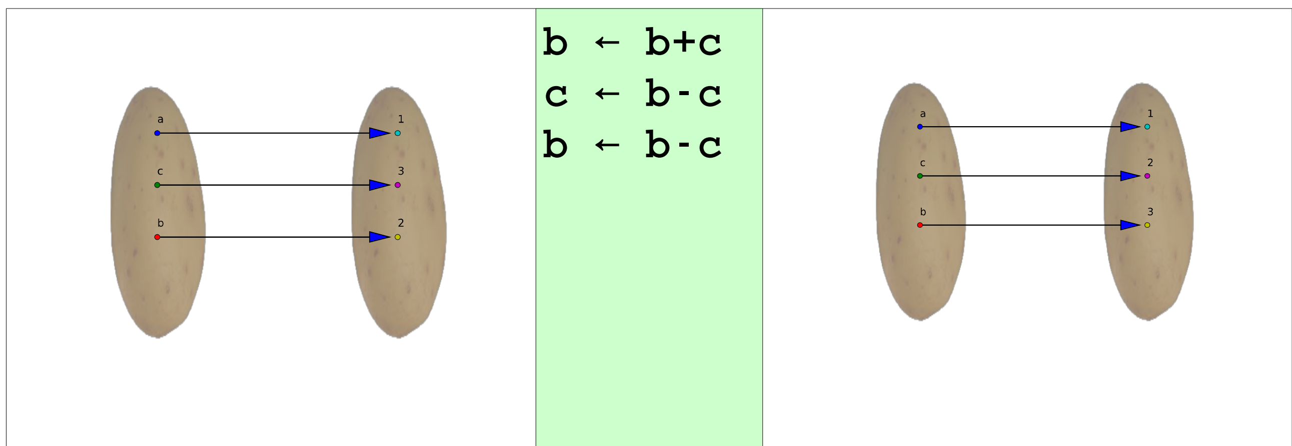
On retient pour la suite qu'un état est une fonction, par exemple de V dans **R** (V désignant l'ensemble des variables, c'est-à-dire l'ordinateur). En Python l'état de la machine peut être connu grâce à la fonction `globals()` qui renvoie un dictionnaire permettant, pour chaque nom de variable, de connaître sa valeur.

3. Effets de bord et instructions

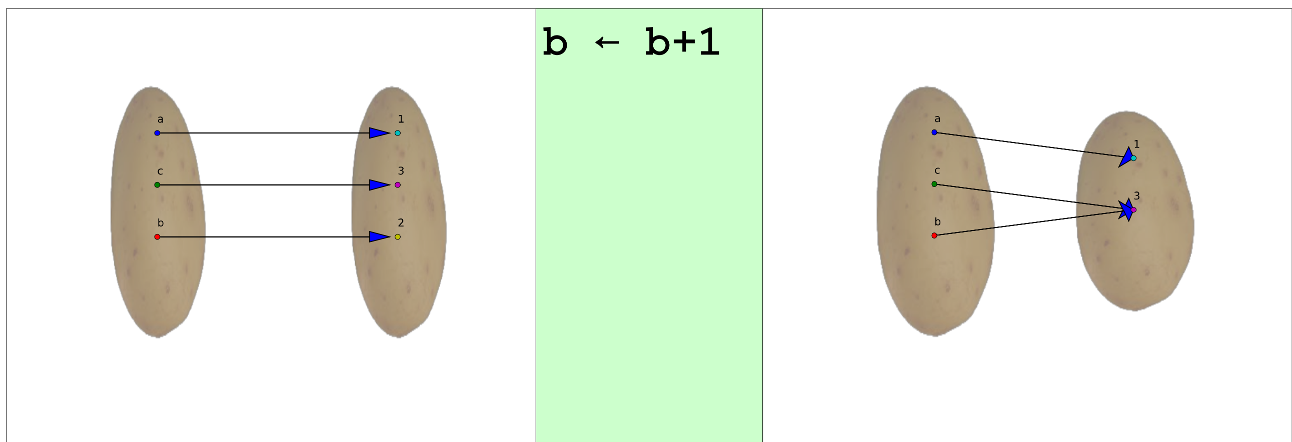
Scott et Strachey appellent ***effet de bord***, une fonction, qui à un état (l'ancien état de la machine) associe un autre état (le nouvel état de la machine). Comme un état est lui-même une fonction, un effet de bord est une fonction entre fonctions, par exemple de (V → **R**) dans (V → **R**).

Voici des exemples (ancien état, algorithme, nouvel état)[5] :

Échange de valeurs :



Incrémentation de b :

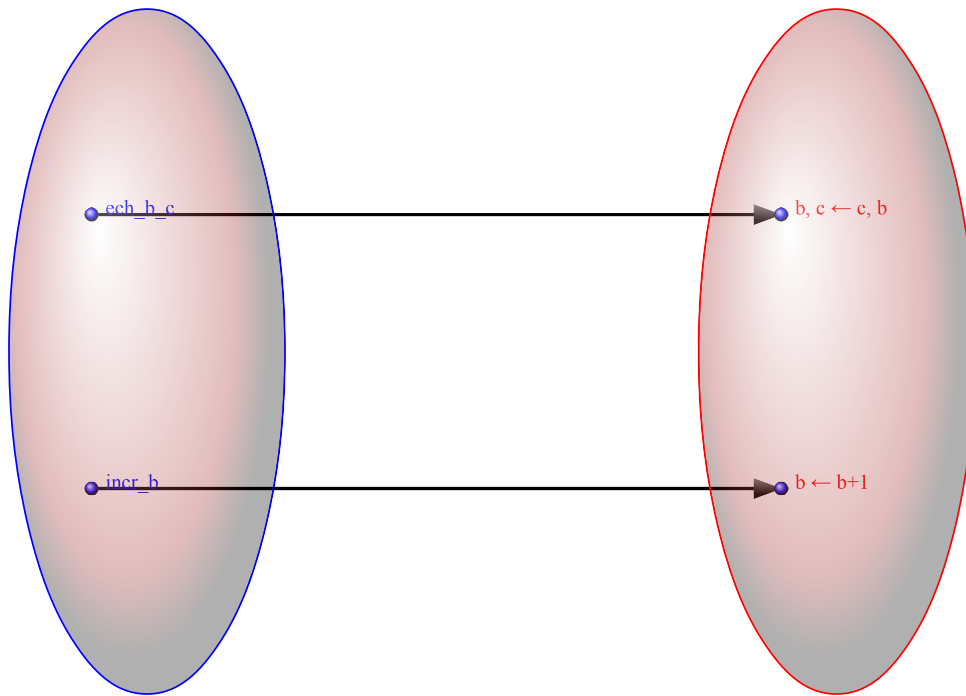


Pour Scott, une **instruction** est la requête d'un effet de bord ; mais ce sont bien les *instructions* qu'il modélise par les fonctionnelles ci-dessus.

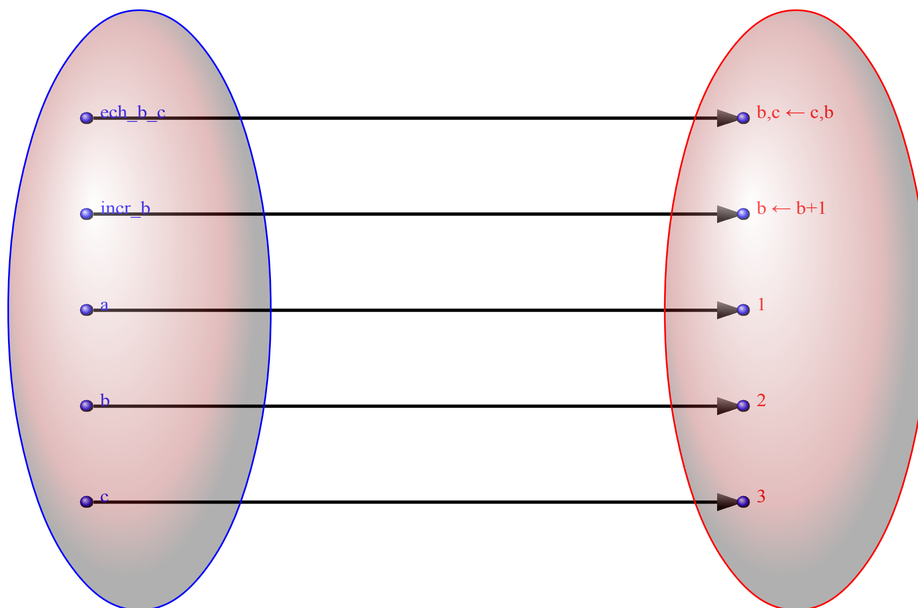
La possibilité de faire du calcul récursif est due à ce que les instructions aussi peuvent être stockées. Par exemple on peut appeler « ech_b_c » la première des instructions ci-dessus (elle échange b et c) et « incr_b » la seconde (elle incrémente b), on peut avoir cet aperçu de la mémoire programme de l'ordinateur :

---

5 La présentation en trois étapes (état,algo,état) s'appelle un triplet de Hoare : C'est une notation introduite par C.A.R. Hoare, auteur d'algorithmes de tri et de recherche rapides, vers le milieu des années 1960.



On peut même, puisque les variables peuvent être de plusieurs types différents, avoir une vue d'ensemble de la machine :



Donc un état peut très bien relier une de ses valeurs de départ à une instruction, qui est elle-même une fonction associant un état (par exemple cet état-là) à un état !

4. Programme

Un programme est une suite d'instructions. C'est-à-dire une suite[6] de fonctions allant de l'ensemble des états dans lui-même (un état étant lui-même une fonction allant de la machine vers elle-même) …

Par exemple, les deux instructions « incr_b » et « ech_b_c » ci-dessus permettent de faire rapidement quelques programmes courts :

- `incr_b; incr_b;` a pour effet d'augmenter b de 2 unités
- `incr_b; ech_b_c;` a pour effet de placer b+1[7] dans c et c dans b
- `ech_b_c; incr_b;` a pour effet de placer c+1 dans b et b dans c
- `ech_b_c; ech_b_c;` n'a aucun effet : Il laisse la machine dans le même état qu'avant son exécution.

La *succession* d'instructions qui constitue un programme informatique est modélisée par la *composition* des fonctions qui les représentent (parce qu'une instruction est une fonction, voir plus haut) mais dans l'ordre inverse. Ainsi si on note e et i les fonctions modélisant les instructions ci-dessus,

- `incr_b; incr_b;` est modélisé par $i^2=i\circ i$
- `incr_b; ech_b_c;` est modélisé par $b\circ i$ (et non $i\circ b$)
- `ech_b_c; incr_b;` est modélisé par $i\circ b$
- `ech_b_c; ech_b_c;` est modélisé par $e\circ e=Id$

Le fait que l'ordre d'exécution des instructions a un effet sur le résultat obtenu illustre la non-commutativité de la composition des fonctions.

5. Récursivité

Par composition des fonctions, un programme informatique, même formé de plusieurs instructions, est modélisé comme s'il était une instruction, par une fonction associant chaque état (l'état de la machine avant l'exécution du programme) à un autre état (celui de la machine après l'exécution du programme).

Et comme l'image de certains noms de variables par l'état peut très bien être elle-même un programme, on peut concevoir, dans la théorie de Scott et Strachey, des programmes agissant sur des programmes[8].

Bien entendu, un programme étant rédigé dans un langage de programmation, il est difficile d'incrémenter un programme. Mais on peut mélanger ses instructions ou en concaténer. En Python la concaténation se note additivement, on va garder cette notation pour ce programme construisant des programmes :

---

6 On rappelle qu'une suite d'éléments d'un ensemble E, est une fonction de N dans E. Un programme est donc une fonction dont les valeurs sont des fonctions liant des fonctions à des fonctions. À ce stade on commence à voir l'intérêt du λ-calcul, dont l'objet d'étude est justement les fonctions portant sur des fonctions.

7 Ce n'est pas vraiment b+1 puisque b est le nom de la variable et ne saurait être additionné : On a commis ici un abus de langage, l'énoncé complet étant « 1 de plus que la valeur de la variable de nom b ».

8 Et même des programmes agissant sur eux-mêmes (se modifiant eux-mêmes, en cours d'exécution) ; ce qui permet de formaliser la récursivité.

```
P ← "x←0;"

pour n allant de 0 à 7
    si randrange(2)=1
        P ← P + "x←x+1;"
    sinon
        P ← P + "x←2×x;"
P ← P + "afficher (x)"
```

La fonction randrange(2) de Python crée un entier égal à 0 ou 1. L'algorithme ci-dessus fabrique un programme P, lequel fabrique un entier en binaire, par l'algorithme de Horner. Voici un programme P parmi les 256 qu'on peut obtenir en exécutant ce programme de programmes :

```
x←0
x←x+1
x←2×x
x←2×x
x←x+1
x←2×x
x←x+1
x←2×x
x←x+1
```

Et pour finir, la version Python (c'est `exec(P)` pour exécuter le programme P) :

```
from random import *

P = "x=0;"

for n in range(8):
    if randrange(2)==1:
        P = P + "x=x+1;"
    else:
        P = P + "x=2*x;"
P = P + "print(x)"

exec(P)
```