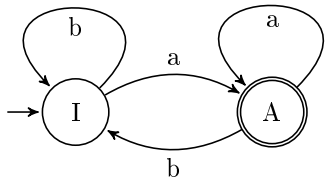
Calculabilité

Automates finis

Question 1 Quels sont les mots acceptés par l'automate suivant ?



On précise que l'état initial est I (marqué par la flèche entrante) et que le seul état acceptant est A (le trait du cercle est doublé).

Question 2 Trouvez un automate qui reconnait les mots de la forme a^{2n} (c'est-à-dire constitués d'un nombre pair de a).

Question 3 En Python on peut coder la fonction de transition d'un automate comme un dictionnaire dont les clés sont les états, et les valeurs sont elles-mêmes des dictionnaires associant à chaque symbole le nouvel état obtenu.

Par exemple pour l'automate de la question 1, on obtient le dictionnaire:

```
{
'I': { 'a': 'A', 'b': 'I' },
'A': { 'a': 'A', 'b': 'I' },
}
```

Écrivez une fonction :

```
def simule_automate(transitions, initial, mot):
    ...
```

qui lit un mot sur un automate défini par un dictionnaire de transitions en partant d'un état initial et retourne l'état obtenu.

Question 4 Décrivez un automate qui accepte exactement les mots de la forme $a^n b^n$ pour $n \leq 10$ (évidemment, si on sait faire pour 10, on sait faire pour n'importe quel $k \in \mathbf{N}$).

Question 5 Fixons un automate fini $\mathcal{A}$, avec n états. Montrez que si $a^n b^n$ est accepté par $\mathcal{A}$, alors il y a un $k < n$ tel que $a^k b^n$ soit aussi accepté. Déduisez-en qu'il n'y a pas d'automate fini qui reconnaisse exactement l'ensemble des mots de la forme $a^n b^n$. (Cette technique se généralise: c'est le lemme d'itération ou lemme de l'étoile.)

Calculer avec une machine de Turing

On va utiliser un simulateur en ligne : http://morphett.info/turing/turing.html.

Le format utilisé pour définir une machine est simple, en codant chaque transition sur une ligne. Plus précisément, si $\delta(e, c) = (e', c', m)$ alors on met une ligne

```
e c c' m e'
```

(en utilisant `l` pour gauche et `r` pour droite).

Question 1 Implémentez dans ce simulateur la machine qui calcule le successeur en binaire. (Attention, le simulateur démarre sur le premier caractère, donc le bit de poids fort!)

Question 2 Implémentez une machine qui accepte exactement le langage des mots de la forme $a^n b^n$.

Implémentation des machines de Turing

On veut réaliser un simulateur de machines de Turing en Python 3. Partant de la définition formelle, l'essentiel du travail consiste à:

- fixer le codage d'un ruban,
- fixer le codage des données qui définissent une machine,
- implémenter la fonction Δ qui passe d'une configuration à une autre.

On suit la même idée que pour les automates: pour une machine $M = (E, I, F, \delta)$, on va représenter la fonction de transition δ par un dictionnaire dont les clés sont les états et les valeurs sont elles-mêmes des dictionnaires, qui donnent la fonction *caractère lu* $\mapsto$ *(état,caractère écrit,décalage)*.

Question 1 L'étape la plus technique est de représenter un ruban bi-infini: on veut pouvoir initialiser le ruban en fournissant un mot, déplacer la tête, lire le caractère courant, et le remplacer.

Une idée est de maintenir deux listes de symboles: celle des symboles à droite de la tête (dans l'ordre inverse) et celle des symboles à gauche de la tête (dans l'ordre normal), en incluant par exemple la tête dans la liste de gauche. Par convention, en dehors de ces listes, toutes les cases sont vides. En inversant l'ordre des éléments dans la liste de droite, un déplacement de tête revient à utiliser la bonne combinaison de `pop()` et `append()`.

Définissez une classe `Ruban` suivant le prototype suivant:

```
class Ruban:
    """Ruban de machine de Turing."""
    def __init__(self, caracteres="", blanc="."):
        """Initialise un ruban.

        Par défaut, le ruban est vierge. On peut spécifier le caractère
        spécial `blanc` qui représente les cases non écrites (on utilise un
        point par défaut). On peut également fournir des `caracteres`
        initialement écrits sur le ruban."""
        ...

    def gauche(self):
        """Décale la tête du ruban vers la gauche."""
        ...

    def droite(self):
        """Décale la tête du ruban vers la droite."""
        ...

    def ecrit(self, caractère):
        """Écrit un `caractere` à l'emplacement courant de la tête de lecture."""
        ...

    def lit(self):
        """Retourne le caractère courant."""
        ...
```

Question 2 Programmez maintenant un simulateur de machines de Turing, soit comme une classe, soit comme une fonction

```
def simule_turing(transitions,initial,mot):
    ...
```

Testez votre code, par exemple avec la fonction de transition:

```
successeur = {
    'I': {
        '0': ('R','1',-1),
        '1': ('I','0',1),
        '.': ('R','1',-1),
        },
    'R': {
        '0': ('R','0',-1),
        '1': ('R','1',-1),
        '.': ('F','.',1),
        }
```

```
}
```

Question 3 Si la question précédente était facile, prenez le temps de soigner votre code, puis étendez-le pour:

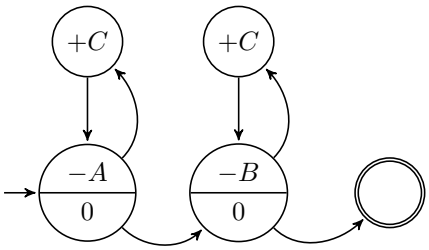
- prendre en compte les jokers comme sur http://morphett.info/turing/turing.html;
- gérer plusieurs rubans.

Machines à compteurs

Les machines à compteurs sont un modèle de calcul extrêment simple, dû à Marvin Minsky. On peut les voir comme des automates agissant non pas sur des mots mais sur un nombre fini de compteurs A, B, C, ... Dans un état donné on peut:

- ou bien incrémenter un compteur et passer dans un état q';
- ou bien tester si un certain compteur est nul:
  - si oui, on change d'état;
  - si non, on décrémente et on change d'état (possiblement différent de celui du cas nul).

On peut démontrer que les fonctions calculables sur les entiers par machines de Turing le sont par machines à compteurs (il faut au moins deux compteurs). Mais c'est surtout un modèle de calcul avec lequel on peut jouer, de la maternelle (sans aller bien loin) à l'université (également testé en stage Hippocampe), en les représentant graphiquement:



Question 1 Que calcule la machine à compteurs ci-dessus dans le compteur C, en fonction des valeurs initiales des compteurs A et B (en démarrant avec $C = 0$)?

Question 2 Modifiez la machine précédente, pour que les valeurs données dans les compteurs A et B ne soient pas perdues (un compteur supplémentaire sera utile).

Question 3 Définissez une machine à compteurs pour la multiplication.

Question 4 Implémentez un simulateur de machines à compteurs: ce devrait être facile, maintenant que vous êtes rôdés.

Question 5 On peut représenter une machine à compteurs par un listing comme celui-ci (qui correspond à la machine de la question 1):

```
A_vers_C     -A  B_vers_C A_vers_C_bis
A_vers_C_bis +C  A_vers_C
B_vers_C     -B  fini B_vers_C_bis
B_vers_C_bis +C  B_vers_C
```

avec la convention que l'état initial est celui de la première ligne. Écrivez une fonction (ou une méthode) qui transforme un tel script en une machine à compteurs au sens de votre implémentation de la question 4.

Vous avez écrit votre premier (ou pas) compilateur!

Problèmes indécidables: réduction

Étant données deux fonctions Python `f` et `g` et un objet `x`, on dit que `f` et `g` coïncident en `x` si:

- ou bien les expressions `f(x)` et `g(x)` s'évaluent en le même résultat (au sens où, par exemple, `f(x) == g(x)` s'évalue en `True`);
- ou bien ni `f(x)` ni `g(x)` ne produisent un résultat (ça boucle indéfiniment).

Question 1 Supposons qu'on vous donne une fonction Python `cestpareil` telle que `cestpareil(f,g,x)` renvoie `True` si `f` et `g` coïncident en `x` et `False` sinon.

À l'aide de `cestpareil`, définissez une fonction qui résout le problème de l'arrêt. Déduisez-en que le problème consistant à savoir si deux fonctions Python coïncident en une certaine valeur n'est pas décidable.

Question 2 On dit que `f` est totale si `f(x)` s'évalue toujours en un objet. Montrez qu'il n'y a pas de fonction Python qui décide si une fonction est totale.

Ces résultats se généralisent: toute propriété non triviale qui concerne la fonction (au sens mathématique) calculée par une machine de Turing (ou un programme Python) est indécidable. C'est le *théorème de Rice*.

Question 3 Fixons une fonction Python `f` (par exemple la fonction constante égale à 0: `lambda x: 0`). On dit que `g` implémente `f` si pour tout objet `x`, `f` et `g` coïncident en `x`. Montrez qu'il n'y a pas de fonction Python qui décide si une fonction `g` implémente `f`.