Pablo Rauzy
pr@up8.edu
https://pablo.rauzy.name/teaching/mdlp

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Méthodologie de la programmation

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / L1 informatique

Séance d (TP) : Programmation sur machines de Turing

N'oubliez pas :

- Les TPs doivent être rendus par courriel au plus tard la veille de la séance suivante avec "[mdlp]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[mdlp] TPd Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tpd").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :

- Raisonner différemment et apprendre à décomposer un problème via la programmation sur machines de Turing universelles.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/mdlp/sd-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/mdlp/seance-d_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-d_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv mdlp_seance-d_files` *`votre-nom`*). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Simulateur de machine de Turing

1. Dans ce TP, votre but va être de programmer des machines de Turing. En plus d'être amusant, cela vous confronte à un autre paradigme de programmation.

   Seulement, il vous faut pour cela un simulateur de machines de Turing, qui interprétera vos descriptions de machines de Turing, tout comme l'interpréteur `python3` interprète vos programmes écrit en Python.

   Le fichier `turingmachine.py` implémente presque tout le mécanisme nécessaire, sauf sa partie "calculatoire" (ligne 41).

   → Décrivez superficiellement (c'est à dire sans entrer dans les détails) le contenu du fichier.

2. La classe `TuringMachine` est constituée de quatre méthodes :
   - La méthode `__init__` prend en argument un fichier source qui décrit une machine de Turing (i.e. notre programme).
     Ensuite elle initialise des attributs de la classe :
     - l'attribut `_tape` correspond au ruban, son contenu initial est lu sur la première ligne du fichier source, chaque caractère correspondant à un symbole dans une case ;
     - l'attribut `_head` correspond à la position de la tête sur le ruban, sa valeur initiale est lue sur la seconde ligne du fichier source, où elle est indiquée avec le symbole `^` pointant vers la case voulue dans la première ligne ;
     - l'attribut `_state` correspond à l'état de la machine, sa valeur initiale est lue sur la troisième ligne du fichier source ;
     - l'attribut `_rules` correspond aux règles de transition de la machine, ces règles sont lues dans la suite du fichier avec une règle par ligne sous la forme suivante :
       `état_courant symbole_lu : nouvel_état symbole_écrit direction`

où les symboles doivent ne faire qu'une seule lettre, et la direction est forcément `<` (gauche) ou `>` (droite). En plus de cela il est possible de laisser des lignes vides ou de mettre des commentaires avec le symbole `;`.

- La méthode `parse_rule` sert à convertir les lignes de règles écrites au format qu'on vient de voir en structure utilisable en Python.
- La méthode `print_tape` affiche l'état du ruban, la position de la tête et l'état courant de la machine.
- La méthode `step` tente d'avancer d'une étape dans l'exécution de la machine, et retourne `True` si elle a réussi, `False` sinon.

Un exemple de machine très simple (et totalement inutile) pourrait être le fichier `exemple.tm` :

```
0
^
not

not 0 : end 1 > ; si 0, écrire 1, aller à droite et s'arrêter
not 1 : end 0 < ; si 1, écrire 0, aller à gauche et s'arrêter
```

Dans cette description de machine, on initialise le ruban avec juste un symbole, `0`, on place la tête de la machine sur ce symbole, et on lui dit qu'on commence dans l'état `not`.

Ensuite deux règles sont données :

- Si on est dans l'état `not` et qu'on lit un `0`, alors on va dans l'état `end`, on écrit un `1`, et la tête de lecture/écriture bouge vers la droite.
- Si on est dans l'état `not` et qu'on lit un `1`, alors on va dans l'état `end`, on écrit un `0`, et la tête de lecture/écriture bouge vers la gauche.

Après une seule étape la machine s'arrête car elle n'a aucune règle pour l'état `end` dans laquelle elle se trouve.

→ Bien qu'on ait pas encore implémenté la méthode `step`, lancez l'exécution de la machine décrite dans `exemple.tm` avec la commande `python3 turingmachine.py exemple.tm`. Que constatez-vous ?

3. Pour écrire la fonction de transition, il faut d'abord comprendre comment sont stockées les règles de transition.
   (a) → Dans la méthode `parse_rule`, avec quelle structure de données sont retournées les règles quand elles sont bien formées ?
   (b) → Donnez comme exemple les valeurs Python correspondantes aux deux règles de `exemple.tm`.
   (c) → Comment sont stockées ces règles dans le dictionnaire `_rules` ?
4. On peut maintenant écrire la méthode `step`.
   Plusieurs informations utiles :
   - Pour vérifier si il existe une entrée dans une dictionnaire pour une clef donnée, on peut utiliser `in`. Par exemple `'foo' in d` renverra `True` si la valeur `d['foo']` existe, et `False` sinon.
   - Lorsque vous déplacez la tête de lecture/écriture sur une case qui n'existe pas, vous la remplirez par convention du symbole `_` que l'on va utiliser comme "symbole blanc".

   → Écrivez la méthode `step`.
5. → Testez maintenant votre code en relançant le programme d'exemple. Vérifiez qu'il se comporte bien comme attendu, y compris si vous remplacez le 0 sur la bande par un 1.

Exercice 2.

Petits programmes.

1. → Écrire une machine de Turing `inverse.tm` qui a sur son ruban un nombre écrit en binaire, qui commence sur le premier chiffre de ce nombre, et qui le parcours entièrement en inversant ses bits au passage.
2. → Écrire une machine de Turing `incr.tm` qui incrémente un nombre écrire en binaire sur sa bande, sachant que la tête est initialement sur le bit de poids faible du nombre.
3. → Écrire une machine de Turing `find.tm` qui a quelque part sur son ruban un `@` (toutes les autres cases contiennent des symboles blancs), et qui s'arrête sur ce symbole dans l'état `found` où qu'il soit sur le ruban.

Exercice 3.

Programme un peu plus gros.

1. → Écrire une machine de Turing `findincr.tm` qui trouve un nombre écrire en binaire n'importe où sur sa bande et l'incrémente.
2. → Écrire une machine de Turing `un2bin.tm` qui convertit en binaire un nombre écrire en unaire.
   Indice : en partant de zéro on peut compter jusqu'à n en incrémentant n fois.
   La position initiale de la tête de lecture/écriture est celle qui vous arrange le mieux. Quand la machine s'arrête, la position de la tête de lecture/écriture n'importe pas mais il ne doit rester sur le ruban que l'écriture binaire du nombre.

Exercice 4.

(Bonus) Programmes plus compliqués. Ne faire que si vous avez fini tout le reste.

1. → Écrire une machine de Turing `bool-pi.tm` qui évalue une formule booléenne écrite sur sa bande en notation polonaise inversée avec des `0`, des `1`, les opérateurs `&` (and), `|` (or), `!` (not).

   Cette notation évite d'avoir à gérer les parenthèses : on traite chaque opérateur dans l'ordre d'arrivée et chaque fois ses opérandes sont les plus proche à sa gauche.

   Exemples :
   — `!0` s'écrit `0!`,
   — `1&0` s'écrit `10&`,
   — `1&(1|0)` s'écrit `110|&`,
   — `1&0|!(!1|0)` s'écrit `10&1!0|!|`.

   La tête de lecture sera placée au début de la formule. Il sera probablement utile d'avoir un symbole indiquant la fin de la formule. Par exemple en utilisant `#`, les deux premières lignes de `bool-pi.tm` pourraient être :

```
0!!1!&01|!1|01&1&!!||#
^
```

2. → Plus difficile : même exercice que le précédent mais cette fois-ci en notation infixe (celle à laquelle on est habitué), sans toutefois considérer que certaines opérations sont prioritaires sur les autres.

   La tête de lecture sera placée au début de la formule. Il sera probablement utile d'avoir un symbole indiquant les limites de la formule. Par exemple en utilisant `#`, les deux premières lignes de `bool.tm` pourraient être :

```
#!(!0)&(!1)|(!((0|1))|1|(!(!(1&(0&1)))))#
 ^
```

   (Il s'agit de la même formule que dans la question précédente.)