Théorie des langages rationnels
TP 2 - Expressions régulières

Adrien Pommellet

18 janvier 2023

L'objectif de ce TP est d'apprendre à utiliser des expressions régulières et à se familiariser avec l'implémentation en Python que nous utiliserons dans le cadre de la théorie des automates.

Téléchargez au préalable les fichiers `tp2_exo1_q1`, `tp2_exo1_q2`, `tp2_exo2_q1`, `tp2_exo2_q2` et `thlr_regex.py` sur Moodle. Ce TP nécessite une installation préalable de la bibliothèque `graphviz` ; si elle ne s'avère pas disponible sur votre poste, utilisez la suite d'instructions suivante :

```
python -m venv ~/envthlr
. ~/envthlr/bin/activate
pip install graphviz
```

Les questions marquées d'une étoile `(*)` seront **notées** et font l'objet d'un rendu dont **la rédaction doit obligatoirement respecter les règles suivantes** :

— Les réponses au TP doivent être regroupées dans un unique fichier nommé `thlr_2_code.py` (tout autre fichier sera ignoré). Inutile d'uploader les bibliothèques externes que l'on vous fournit telle que `thlr_automata.py`.

— Il est impératif que le fichier soit interprétable, sans la moindre erreur de syntaxe. Si ce n'est pas le cas, le rendu entier sera ignoré. Testez votre code sous Linux directement depuis la console avec l'instruction `python3 thlr_2_code.py` depuis un répertoire contenant ses éventuelles dépendances, sans passer par un IDE.

— Les noms de fonctions donnés dans l'énoncé doivent être respectés.

— La réponse à la question numéro `i` et les éventuelles sous-fonctions utilisées doivent être placées entre des balises de la forme `#[Qi]` et `#[/Qi]` (sans espace après `#`, avec un `/` et non un \). Ainsi, si l'on vous demande d'écrire une fonction `f`, votre réponse doit être de la forme :

```
#[Qi]
def f():
  # Insert your code here
#[/Qi]
```

— Testez vos fonctions sur des exemples pour vous assurer de leur correction ; n'incluez toutefois pas ces tests et ces exemples entre les tags `#[Qi]` et `#[/Qi]`.

— Certaines fonctions sont interdépendantes : une erreur sur la première fonction peut compromettre les suivantes, soyez donc vigilants.

Le non-respect de ces critères peut mettre en défaut le système de notation automatique et conduire à une note de 0 pour le rendu dans son intégralité, sans contestation possible.

1 Utiliser les expressions régulières en Perl

Nous utiliserons l'outil *Perl*, installé par défaut sur la plupart des systèmes d'exploitation. La commande `man perlretut` affiche un tutoriel sur l'usage des expressions régulières dans Perl. La syntaxe de Perl pour les expressions régulières diffère de celle que nous utilisons en TD. Dans le tableau qui suit $c_1, c_2, \ldots, c_n$ désignent des lettres $l \in \Sigma$, et $L(e)$ est le langage dénoté par l'expression régulière e.

Langage	**Expression régulière Perl**
Σ	`.`
$\{c\}$	`c`
$\{c_1, c_2\}$	`[`c_1c_2`]`
$\{c_1, c_2, \ldots, c_n\}$	`[`c_1`-`c_n`]`
$\Sigma \setminus \{c_1, c_2\}$	`[^`c_1c_2`]`
$L(e)L(f)$	ef
$L(e) \cup L(f)$	`(`e`|`f`)`
$L(e) \cup \{\varepsilon\}$	e`?`
$L(e)^\star$	e`*`
$L(e)^+$	e`+`

Notez qu'il est possible d'assembler des classes de caractères entre elles. Ainsi, `[a-zA-Z]` représente une lettre minuscule ou majuscule. Enfin, pensez à échapper les caractères spéciaux par un '\'. Par exemple, si vous avez besoin de reconnaître le caractère '.' (un point), il faut écrire '`\.`'. Ces caractères spéciaux sont `\ . ? * [ ] | ( )`.

Pour tester vos expressions régulières avec Perl, entrez **(sans la copier-coller depuis ce fichier)** une commande de la forme suivante dans un terminal :

`perl -0777 -pe 's/`*`expression-régulière`*`/`*`texte-à-substituer`*`/gsm'` *`fichier`*

`texte-à-substituer` permet de spécifier par quoi remplacer le texte reconnu. Par exemple, si l'on applique `perl -0777 -pe 's/[a-z]/@/gsm'` sur un fichier contenant `azertyAZERTYuiop`, on obtient comme affichage `@@@@@@AZERTY@@@@` : chaque lettre minuscule reconnue a été remplacé par `@`. Le fichier original n'a pas été modifié par cette opération : le résultat de la substitution est uniquement envoyé à l'écran.

1.1 Commentaires du langage Pascal

On cherche à reconnaître les commentaires en Pascal qui sont de la forme `{`*`texte`*`}`, où *`texte`* peut être composé de n'importe quelle suite de caractères.

Question 1. Dans un premier temps, on suppose que l'on a un seul commentaire par fichier. Utilisez Perl dans le fichier `tp2_exo1_q1` pour remplacer les commentaires reconnus par le texte `COMMENTAIRE`.

Question 2. Si l'on a désormais plusieurs commentaires dans un fichier, que se passe-t-il ? Écrivez une nouvelle expression dans le fichier `tp2_exo1_q2` permettant de pallier à ce problème, toujours en remplaçant les commentaires reconnus par `COMMENTAIRE`.

1.2 Chaînes de caractères du langage C

On cherche cette fois à trouver les chaînes de caractères du langage C. Celles-ci sont comprises entre guillemets et contiennent un nombre indéfini de caractères : "*texte*", où *texte* est encore une suite de n'importe quel caractère.

Question 3. Écrivez une expression régulière permettant de reconnaître de telles chaînes (il peut y en avoir plusieurs) dans le fichier `tp2_exo2_q1` : elle devra remplacer les motifs reconnus par le texte `CHAINE`.

Question 4. Il est possible dans ces chaînes *d'échapper* (ou *déspécialiser*) les caractères spéciaux en les faisant précéder du caractère \. Par exemple, si on veut pouvoir afficher le caractère ", on doit écrire `"Texte \" suite du texte"` (sinon il aurait été reconnu comme le guillemet de fin de chaîne). En fait, tous les caractères peuvent être échappés de cette manière. Ajoutez cette possibilité à votre précédente expression régulière et appliquez-la au fichier `tp2_exo2_q2`.

2 Une implémentation des expressions régulières en Python

Importez au préalable le fichier `thlr_regex.py` en utilisant en préambule l'instruction :

```
from thlr_regex import *
```

La première ligne permet de n'effectuer l'importation que quand le fichier est directement exécuté ; ce ne sera pas le cas lors de l'évaluation automatique du TP, l'environnement se chargeant de gérer les bibliothèques, d'où cette importation conditionnelle.

2.1 Définition

Une expression régulière s'écrit habituellement sous la forme d'une chaîne de caractères. Toutefois, une structure d'*arbre* semblable à celle décrite dans la figure 1 est bien plus intéressante pour le programmeur : elle permet de savoir immédiatement à quels éléments chaque opérateur s'applique, sans avoir à interpréter la chaîne. Formellement, un arbre est une structure de données constituée de nœuds, étiquetés et reliés hiérarchiquement par des arêtes orientées selon une relation parent-enfant : un nœud (dit *enfant*) ne peut être pointé que par au plus un autre nœud (dit *parent*), situé immédiatement au dessus de lui. Il ne peut y avoir qu'un seul nœud au sommet de cette hiérarchie, appelé la *racine*. Les nœuds à la base de l'arbre sont appelés les *feuilles*.

Notre implémentation en Python de cette structure d'arbre sera une *classe* `RegEx`. Une classe est une manière précise de regrouper différents objets (appelés *attributs*) et fonctions (appelées *méthodes*). Un tel regroupement est appelé une *instance* de la classe. Si `X` est une instance d'une classe ayant un attribut `a`, on peut obtenir la valeur de cet attribut `a` pour l'instance `X` grâce à l'instruction `X.a` ; de même, si `f()` est une méthode, on l'applique grâce à l'instruction `X.f()`. On ne vous demandera pas de créer de classes, mais vous aurez à utiliser des attributs et des méthodes de classes existantes.

Pour créer l'arbre d'une expression régulière, on utilise la fonction `new_regex(regex_string)` qui construit à partir d'une chaîne de caractères `regex_string` décrivant une expression régulière une représentation sous forme d'arbre de cette dernière. Cet arbre peut être visualisé dans un fichier `file.pdf` en appliquant à une instance de la classe `RegEx` la méthode `.export("file")`. Ainsi, la suite d'instructions :

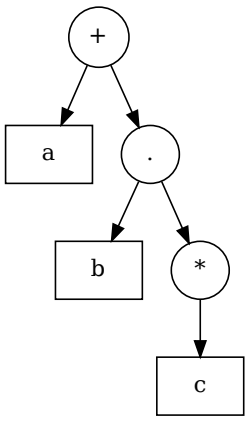

FIGURE 1 – Une représentation graphique de l'expression régulière `a+b.c*`.

```
e = new_regex("a+b.c*")
e.export("tp2_tree")
```

crée dans le répertoire par défaut un fichier `tp2_tree.pdf` contenant l'arbre de la figure 1 associé à l'expression régulière $e = a + bc^*$. Notez que dans cette représentation, l'opérateur `.` doit être explicité dans un souci de clarté. Les règles de priorité usuelles sur les opérateurs s'appliquent.

Question 5. Créez les arbres `r1` et `r2` associés aux expressions régulières $r_1 = (ab + c)^* + b$ et $r_2 = a(b + c + \varepsilon)^*$, puis affichez-les. Copiez-collez le symbole ε depuis cet énoncé si besoin est.

Question 6. Exécutez la suite d'instructions :

```
print(r1.root)
print(len(r1.children))
print(r1.children[1].root)
```

Quel sens pouvez-vous donner à ces différentes opérations ?

2.2 Manipulation d'arbres

Plus formellement, les attributs de la classe `RegEx` sont la chaîne `root` qui décrit l'étiquette de la racine de l'arbre, qui peut être un opérateur `+`, `*`, `.` ou une lettre de l'alphabet Σ, et la liste `children` (dont la longueur dépend de l'arité de l'opérateur à la racine) des `RegEx` enfants de la racine.

On souhaite désormais écrire une fonction `regex_to_string(regex)` qui renvoie une représentation entièrement parenthésée sous forme de chaîne d'un arbre `regex` de type `RegEx` : toute opération binaire ou unaire est mise entre parenthèses, seules les lettres en sont exemptées. Ainsi, plutôt que d'écrire `a.b+c*`, on écrira `((a.b)+(c)*)`. Pour reprendre l'exemple de la figure 1 :

```
>> print(regex_to_string(e))
(a+(b.(c)*))
```

Intuitivement, `regex_to_string(regex)` doit être une fonction récurrente ; par exemple, si un arbre `a` de racine `"."` a pour enfants deux arbres `a1` et `a2`, la chaîne que l'on souhaite renvoyer sera :

```
"(" + regex_to_string(a1) + "." + regex_to_string(a2) + ")"
```

L'opérateur `+` permet de concaténer des chaînes en Python.

Question 7 ($\star$). Écrire une fonction `regex_to_string(regex)` renvoyant une représentation entièrement parenthésée sous forme de chaîne d'un arbre `regex` de type `RegEx`.