IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UE TAL

TP 2 : Expressions régulières / Automates de type fini (mercredi 12 février 2020)

Yannis Haralambous (IMT Atlantique)

1 Expressions régulières

1.1 Introduction

Pour utiliser les expressions régulières sous Python il faut importer le module `re` :

```
import re
```

Une expression régulière est d'abord compilée et le résultat est stocké dans un objet `RegexObject`. On écrit une expression régulière dans une «chaîne brute Python» : une chaîne délimitée par `r"` et `"`. Exemple : `r"[a-z]+"` est l'expression régulière qui correspond aux chaînes formées d'une ou plusieurs lettres entre `a` et `z`.

Pour compliler cette expression et créer un objet `RegexObject` on écrit :

```
r = re.compile(r"[a-z]+")
```

L'objet `r` a plusieurs méthodes, nous allons en utiliser trois :

1. `findall` qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme de liste Python ;
2. `finditer` qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme d'itérateur Python ;
3. `sub` qui sert à appliquer une expression régulière, à remplacer les sous-chaînes trouvées par d'autres chaînes.

Il s'agit donc de fonctionnalités similaires aux «chercher» et «chercher / remplacer» des éditeurs de texte.

Une méthode plus simple, du nom de `match()` va simplement tester si une chaîne satisfait les contraintes imposées par l'objet expression régulière auquel on l'applique.

1.2 Syntaxe des expressions régulières «à la Perl»

Avant de voir l'utilisation des expressions régulières sous Python, un rappel de la syntaxe des expressions régulières «à la Perl» :

- `toto` va trouver les sous-chaînes `toto` ;

- `.` est un caractère quelconque, mis à part le passage à la ligne `\n` et le retour chariot `\r` ;
- `[ax123Z]` signifie : «un caractère quelconque parim `a`, `x`, `1`, `2`, `3` et `Z`» ;
- `[A-Z]` signifie : «un caractère quelconque dans l'intervalle de `A` à `Z`» ;
- le trait d'union sert à indiquer les intervalles mais peut faire partie des caractères recherchés s'il est placé à la fin : `[AZ-]` signifie : «un caractère quelconque parmi `A`, `Z` et `-`» ;
- on peut combiner à volonté les caractères énumérés et les intervalles : par exemple `[A-Za-z0-9.:?]` signifie «une lettre minuscule ou majuscule, un chiffre, un point, un deux-points, ou un point d'interrogation» ;
- les caractères `(`, `)`, `\`, `[`, `]` peuvent être recherchés, à condition de les protéger par un antislash : `\(`, `\)`, `\\`, `\[`, `\]` ;
- le symbole `^` placé après le crochet ouvrant indique que l'on va chercher le complémentaire de ce qui est placé entre les crochets. Exemple : `[^a-z]` va trouver un caractère quelconque *qui ne soit pas* une lettre entre `a` et `z` ;
- on dispose des quantificateurs suivants :
  1. `*` (zéro, une ou plusieurs fois),
  2. `+` (une ou plusieurs fois),
  3. `?` (zéro ou une fois),
  4. `{n,m}` (entre `n` et `m` fois),
  5. `{n,}` (plus de `n` fois) ;
- on dispose également des quantificateurs «non gourmands» suivants :
  1. `*?` (zéro, une ou plusieurs fois),
  2. `+?` (une ou plusieurs fois),
  3. `??` (zéro ou une fois),
  4. `{n,m}?` (entre `n` et `m` fois),
  5. `{n,}?` (plus de `n` fois) ;

  La différence entre quantificateurs «gourmands» et «non gourmands» provient du fait que les premiers vont trouver la sous-chaîne la plus longue respectant les contraintes alors que les deuxièmes vont trouver la chaîne la plus courte.

  Exemple : l'expression `[a-z]+` appliquée à «`mon ami Pierrot`» va trouver `mon` alors que `[a-z]+?` va trouver `m` (ce qui n'a que peu d'intérêt). Autre exemple (qui montre l'utilité des quantificateurs non gourmands) : l'expression `\(.+\)` appliquée à «`Brest (29) et Aix (13)`» va retourner `29) et Aix (13` puisque c'est la plus longue sous-chaîne délimitée par une parenthèse ouvrante et une parenthèse fermante. Par contre `\(.+?\)` va retourner d'abord `29` et ensuite `13` ;
- les symboles `^` et `$` servent à indiquer le début et la fin d'une chaîne. Par exemple : `^a.+` va trouver toutes les chaînes qui commencent par un `a`, `toto$` va trouver toutes les chaînes qui finissent par `toto`, `^ $` va trouver toutes les chaînes égales à un blanc ;
- l'opérateur «ou» `|` sert à indiquer un choix entre deux expressions ;
- on peut utiliser les parenthèses pour deux raisons :
  1. pour délimiter une expression qui sera utilisée par l'opérateur «ou» ou à laquelle on va appliquer un quantificateur (exemple : `abc(toto)+` signifie «`abc` suivi d'un ou plusieurs `toto`») ;

2. pour délimiter une sous-chaîne que l'on va récupérer par la suite. On appelle cette sous-chaîne, un «groupe».

Ce double usage des parenthèses peut être gênant : en écrivant `abc(toto)+` on fait de `toto` un groupe, même si on n'a pas l'intention de le récupérer par la suite. En écrivant `abc(?:toto)+` les parenthèses ne servent qu'au premier usage, aucun groupe n'est formé.

1.3 Utilisation des expressions régulières sous Python

1.3.1 Recherche

Supposons que l'on veuille trouver tous les mots de la chaîne «Le bon chasseur sachant chasser sait chasser sans son chien» contenant un «s». On peut trouver un tel mot en écrivant `[a-rt-z]*s[a-z]*`. On peut donc déjà compiler une expression régulière :

```
import re
r = re.compile(r"[a-rt-z]*s[a-z]*")
```

Pour l'appliquer à la chaîne on écrira :

```
m = r.findall("Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

Le résultat sera une liste Python contenant tous les mots trouvés :

```
['chasseur', 'sachant', 'chasser', 'sait', 'chasser', 'sans', 'son']
```

On peut ré-écrire le code précédent en utilisant un *itérateur* Python :

```
import re
r = re.compile(r"[a-rt-z]*s[a-z]*")
for m in r.finditer("Le bon chasseur sachant chasser sait chasser sans son chien"):
    print(m.group())
```

L'avantage de cette écriture est que l'on récupère non pas des simples chaînes de caractères mais des objets `MatchObject` qui ont leurs propres méthodes et attributs.

1.3.2 Recherche / remplacement

Maintenant nous allons essayer de rendre la phrase «Le bon chasseur sachant chasser sait chasser sans son chien» conforme au dialecte chti-mi. On peut commencer par remplacer tous les «s» par des «ch» :

```
import re
r = re.compile(r"s")
m = r.sub(r"ch","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

Le résultat est «Le bon chachcheur chachant chachcher chait chachcher chanch chon chien», qui est relativement imprononçable. On peut rectifier le tir en évitant les doubles «ch». On va donc remplacer un *nombre quelconque de lettres s consécutives* par un seul «ch» :

```
import re
r = re.compile(r"s+")
m = r.sub(r"ch","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

Le résultat «Le bon chacheur chachant chacher chait chacher chanch chon chien» est nettement plus chti-mi, mais il reste un cas problématique : le «s» muet du mot «sans» est devenu un «ch» prononcé dans «chanch». Il faut donc éviter de convertir les «s» en fin de mot :

```
import re
r = re.compile(r"s+([a-z]+)")
m = r.sub(r"ch\1","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

Pour ce faire, on a créé un groupe `([a-z]+)` que l'on retrouve dans la chaîne de remplacement (`\1`). Le réchultat «Le bon chacheur chachant chacher chait chacher chans chon chien» est chans contechte parfaitement chti-mi.

1.3.3 Recherche / remplacement avec utilisation de fonction

Lorsqu'on remplace une sous-chaîne par une autre il peut être utile d'intercaler un traitement entre lecture de la sous-chaîne et écriture dans la nouvelle chaîne. Python nous permet d'appliquer une fonction à chacune des sous-chaînes trouvées. Imaginons que dans la chaîne `toto 123 blabla 456 titi` on veut représenter les nombres en hexadécimal. Ce calcul est trop compliqué pour être fait uniquement par des expressions régulières, on utilisera donc une fonction

```
def ecrire_en_hexa ( entree ):
    return hex( int( entree.group() ) )
```

et on écrira :

```
import re
r = re.compile(r"[0-9]+")
m = r.sub( ecrire_en_hexa,"toto 123 blabla 456 titi" )
print(m)
```

Le résultat est bien `toto 0x7b blabla 0x1c8 titi`. L'argument de la fonction est un objet de type `MatchObject`. La méthode `group()` fournit la chaîne tout entière, alors que `group(`n`)` fournira le n-ième groupe de la sous-chaîne.

1.4 Exercices

Récupérer sur Moodle le fichier `gen1551.csv`. Pour le lire ligne par ligne, utiliser le code Python suivant :

```
f = open("gen1551.csv",'r')
for ligne in f:
    #faire qqch avec la ligne ligne
f.close()
```

1.4.1 Exercice 1

Que fait le code suivant ?

```
import re
r = re.compile(r"^([0-9]+);[^;]*;PAUL;")
f = open("gen1551.csv",'r')
for ligne in f:
    for m in r.finditer(ligne):
        print(m.group(1)+" OK")
f.close()
```

SOLUTION Il trouve les gens dont le prénom est PAUL.

1.4.2 Exercice 2

Complétez ce programme afin qu'il sorte les identifiants des gens nés dans un village dont le nom commence par PLOU.

SOLUTION

```
import re
r = re.compile(r"^([0-9]+);[^;]*;PAUL;[^;]*;PLOU")
f = open("Awk/gen1551.csv",'r')
for ligne in f:
    for m in r.finditer(ligne):
        print(m.group(1)+" OK")
f.close()
```

1.4.3 Exercice 3

Remplacez les lieux de naissance des personnes trouvées dans l'exercice 2 par des lieux qui commencent par LOC (par exemple : PLOUNEVEZ devient LOCNEVEZ).

Rappel : pour écrire dans un fichier on utilise le code suivant :

```
o = open("fichier_sortie",'w')
o.write("texte à écrire")
o.close()
```

SOLUTION

```
import re
r = re.compile(r"^([0-9]+;[^;]*;PAUL;[^;]*;)PLOU")
f = open("Awk/gen1551.csv",'r')
o = open("tmp","w")
for ligne in f:
    o.write(r.sub(r"\1LOC",ligne))
f.close()
o.close()
```

1.4.4 Exercice 4

Incrémentez les dates de naissance des personnes trouvées dans l'exercice 2 de 10 ans.

Tuyaux :

1. définir une fonction `traiterdate` qui va gérer le remplacement de la chaîne qui nous intéresse ;
2. en Python on passe d'un objet nombre entier à un objet chaîne en utilisant `str`, pour l'opération inverse on dispose de la fonction `int`.

SOLUTION

```
import re
def traiterdate( entree ):
    return entree.group(1)+"/"+str(int(entree.group(2))+10)+";PLOU"

r = re.compile(r"^([0-9]+;[^;]*;PAUL;[0-9][0-9]/[0-9][0-9])/([0-9]{4});PLOU")
f = open("Awk/gen1551.csv",'r')
o = open("tmp","w")
for ligne in f:
    o.write(r.sub(traiterdate,ligne))
f.close()
o.close()
```

Compter le nombre de fiches où le nom de la personne est ABALAIN.

SOLUTION

```
compteur=0
r = re.compile(r"^([0-9]+);ABALAIN;")
f = open("Awk/gen1551.csv",'r')
for ligne in f:
    if r.findall(ligne):
        compteur=compteur+1
f.close()

print(compteur)
```

La réponse est 13.

1.4.5 Exercice 5

Calculez l'âge moyen lors des mariages, en considérant que tous les mois ont 30 jours. À noter que les dates de naissance et de mariage sont données par les champs DN et DM.

Tuyau : si a_m, m_m, d_m sont resp. l'année, le mois et le jour de mariage et a_n, m_n, d_n de même pour la naissance, l'âge d'une personne lors du mariage peut être exprimé par la formule

$$A = \left(a_m + \frac{m_m - 1}{12} + \frac{d_m - 1}{360}\right) - \left(a_n + \frac{m_n - 1}{12} + \frac{d_n - 1}{360}\right).$$

SOLUTION

```
compteur=0
total=0.
r = re.compile(r"^[0-9]+;[^;]*;[^;]*;([0-9][0-9])/([0-9][0-9])/([0-9]{4});[^;]*;([0-9][0-9])/\
([0-9][0-9])/([0-9]{4})")
f = open("Awk/gen1551.csv",'r')
for ligne in f:
    for m in r.finditer(ligne):
        datenaissance=int(m.group(3))+((int(m.group(2))-1)/12)+((int(m.group(1))-1)/360)
        datemariage=int(m.group(6))+((int(m.group(5))-1)/12)+((int(m.group(4))-1)/360)
        total = total + datemariage - datenaissance
        compteur = compteur+1
f.close()

print(total/compteur)
```

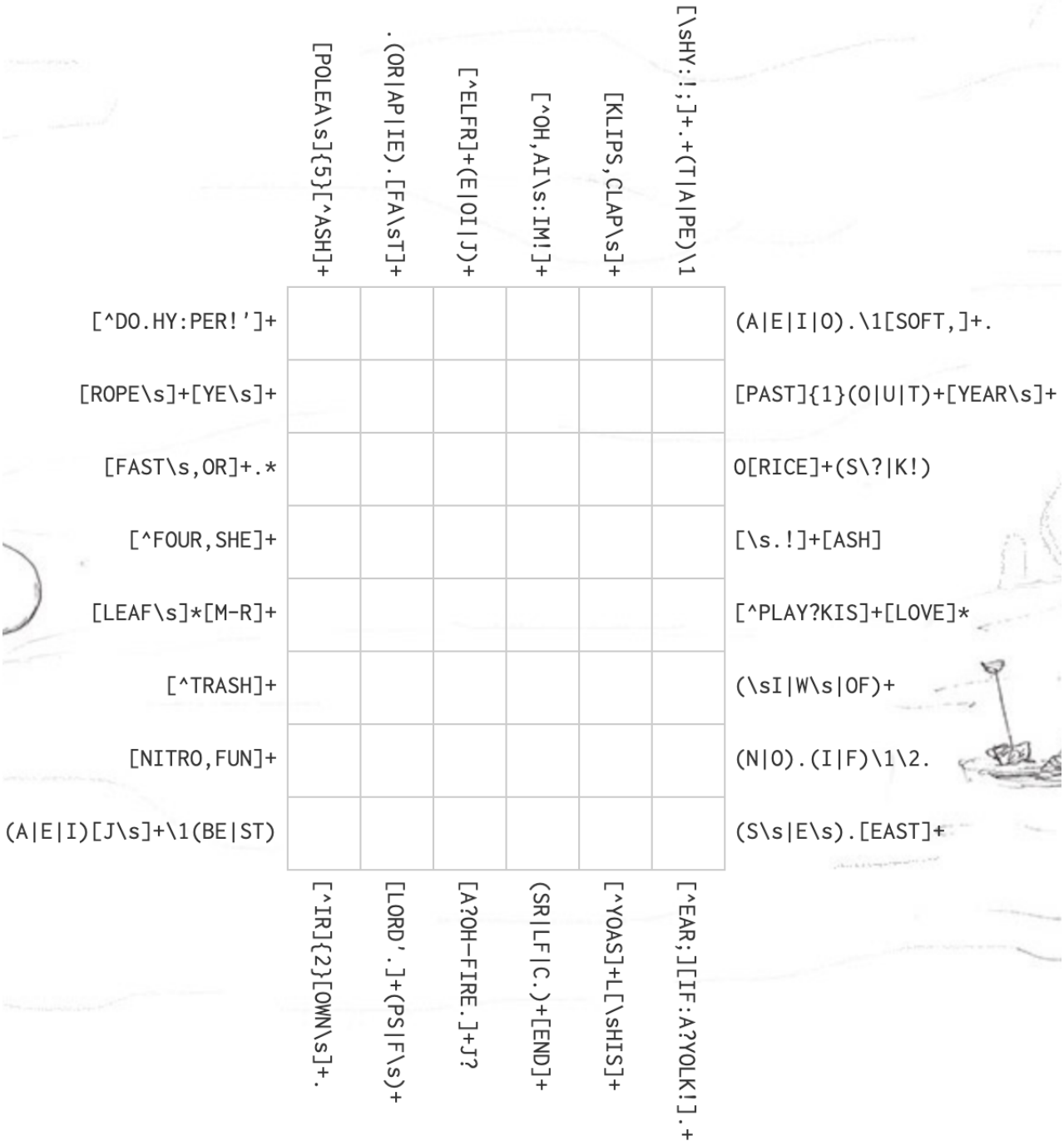
La solution est 25.0805717999. Attention : il faut initialiser `total=0.` sinon le résultat est un nombre entier.

1.5 Puzzles

Résolvez les puzzles suivants (ils proviennent de `www.regexcrossword.com`, un site dédié à ce jeu, à bientôt les championnats !).

	(FI|A)+	(YE|OT)K	(.)[IF]+	[NODE]+	(FY|F|RG)+
(Y|F)(.)\2[DAF]\1					
(U|O|I)*T[FRO]+					
[KANE]*[GIN]*					

	[RUTH]*(OE|EO)[RB]*	(BG|ON|KK)+[RIF]+	(MN|BO|FI)[EU]{2,}	(KT|AL|ET)+G	[OH](PR|AX|TR)+
[IT](O)*(BE|AD)*\1					
[NORMAL]+T{2}					
.*(XA|BE).*					
(EG|UL){2}[ALF]*					
[REQ]*(G|P)(.)+					



2 Automates de type fini

Nous allons modéliser la conjugaison de quelques verbes français.

Commençons par le verbe «parler».

Installer le module `pysimpleautomata` en écrivant

```
pip3 install pysimpleautomata --user
```

Lire la doc sur `https://pysimpleautomata.readthedocs.io/en/latest/tutorial.html`

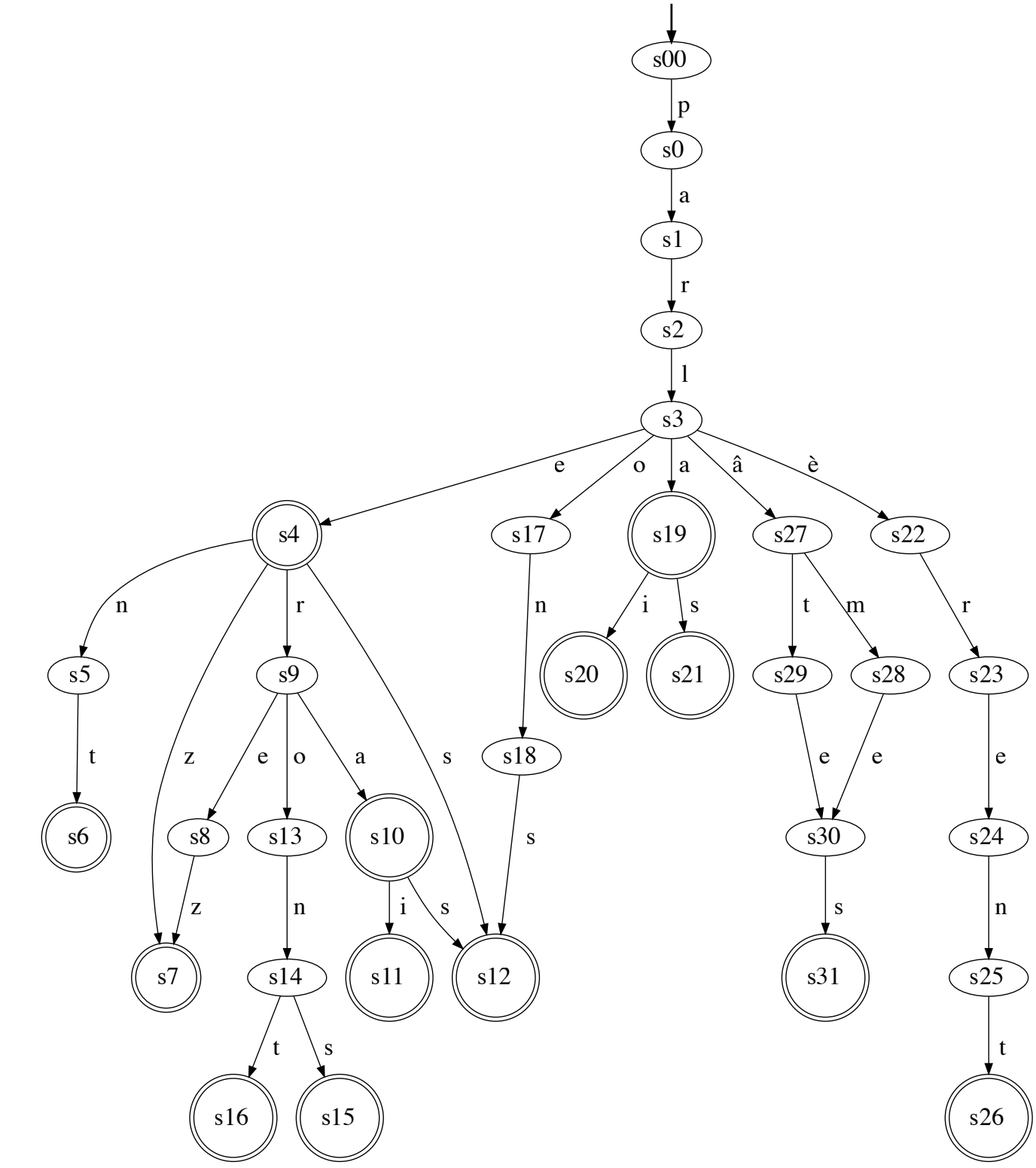
2.1 Exercice de conjugaison

Écrire un automate qui modélise la conjugaison du verbe «parler» à l'indicatif, au présent, futur et passe simple.

Cet automate doit reconnaître les formes correctes et refuser toute erreur.

SOLUTION

```
from PySimpleAutomata import DFA,automata_IO

automate={
'alphabet': {'a','b','c','d','e','f','g','h','i','j','o','p','q','r','s','t','u','v','w','x','y','z'},
'states': {'s00','s0','s1','s2','s3','s4','s5','s6','s7','s8','s9','s10','s11','s12','s13','s14',\
's15','s16','s17','s18','s19','s20','s21','s22','s23','s24','s25','s26','s27','s28',\
's29','s30','s31'},
'initial_state': 's00',
'accepting_states': {'s4','s6','s7','s10','s11','s12','s15','s16','s19','s20','s21','s26','s31'},
'transitions': {
('s00','p'): 's0',
('s0','a'): 's1',
('s1','r'): 's2',
('s2','l'): 's3',
('s3','e'): 's4',
('s4','n'): 's5',
('s5','t'): 's6',
('s4','z'): 's7',
('s4','r'): 's9',
('s4','s'): 's12',
('s9','e'): 's8',
('s8','z'): 's7',
('s9','a'): 's10',
('s10','i'): 's11',
('s10','s'): 's12',
('s9','o'): 's13',
('s13','n'): 's14',
('s14','s'): 's15',
('s14','t'): 's16',
('s3','o'): 's17',
('s17','n'): 's18',
('s18','s'): 's12',
('s3','a'): 's19',
('s19','i'): 's20',
('s19','s'): 's21',
('s3','è'): 's22',
('s22','r'): 's23',
('s23','e'): 's24',
('s24','n'): 's25',
('s25','t'): 's26',
('s3','â'): 's27',
('s27','m'): 's28',
('s27','t'): 's29',
('s28','e'): 's30',
('s29','e'): 's30',
('s30','s'): 's31',
}
}

print(DFA.dfa_word_acceptance(automate,('p','a','r','l','e','r','o','n','t')))
print(DFA.dfa_word_acceptance(automate,('p','a','r','l','e','r','e')))

automata_IO.dfa_to_dot(automate, "automata", "/hom/yannis/texmf/cours/ue-tlft/")

DFA.dfa_completion(automate)
new_dfa=DFA.dfa_minimization(automate)

automata_IO.dfa_to_dot(new_dfa, 'automata2', '/hom/yannis/texmf/cours/ue-tlft/')
```



2.2 Optimisation

Est-ce que votre automate est complet ? Sinon, complétez-le.

Est-ce que votre automate est déterministe ? Sinon, déterminisez-le.

Minimisez votre automate en utilisant la méthode du package. Comparez l'automate initial et l'automate minimal.

2.3 Extension à d'autres verbes

Étendre l'automate aux verbes «aimer» (moins de lettres), «placer» (attention aux cédilles), «aller» (attention, verbe irrégulier).