INF 424

TP 1 de logique

Yannis Haralambous (Télécom Bretagne)

Nous allons utiliser le package Python NLTK. Pour l'installer sans être root, on écrira

```
pip install nltk --user
```

Attention, dans ce TP on utilise la version 3 de NLTK, si vous avez la version 2, il faudra écrire

```
pip install --upgrade nltk --user
```

1 Modélisation de la langue naturelle par les grammaires formelles

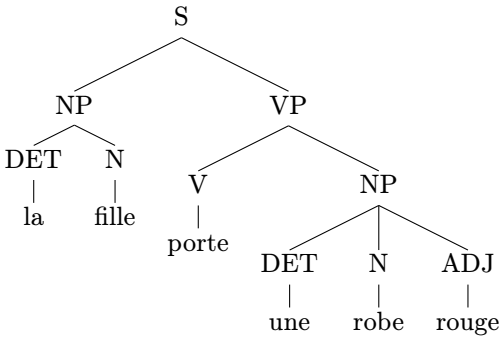
Dans cette section nous aurons besoin de deux outils non vus en cours : les *grammaires formelles à structures de traits* et le *λ-calcul*.

Commençons par voir comment les grammaires formelles, telles qu'on les a vues en cours, sont implémentées sous Python.

1.1 Grammaires formelles standard

Prenons la phrase *la fille porte une robe rouge*.

On peut l'analyser syntaxiquement comme



Cette phrase peut être générée par la grammaire

```
% start S
S -> NP VP
NP -> DET N ADJ | DET N
VP -> V NP
DET -> 'la' | 'une'
N -> 'fille' | 'robe'
V -> 'porte'
ADJ -> 'rouge'
```

Pour vérifier que la phrase peut être générée par la grammaire, nous allons écrire le code ci-dessus dans un fichier `grammaire.cfg` et utiliser la classe `load_parser` comme suit :

```
sent = "la fille porte une robe rouge".split()
parser = nltk.load_parser("file:grammaire.cfg")
for tree in parser.parse(sent):
    print tree
    tree.draw()
```

et le résultat est bien

```
(S
  (NP (DET la) (N fille))
  (VP (V porte) (NP (DET une) (N robe) (ADJ rouge))))
```

comme attendu (la boucle existe car il peut y avoir plusieurs arbres syntaxiques pour la même phrase).

D'ailleurs le parseur nous décrit toutes les dérivations dans la sortie :

```
|.  la  .fille .porte . une  . robe .rouge .|
|[------]      .      .      .      .      .| [0:1] 'la'
|.      [------]      .      .      .      .| [1:2] 'fille'
|.      .      [------]      .      .      .| [2:3] 'porte'
|.      .      .      [------]      .      .| [3:4] 'une'
|.      .      .      .      [------]      .| [4:5] 'robe'
|.      .      .      .      .      [------]| [5:6] 'rouge'
|[------]      .      .      .      .      .| [0:1] DET -> 'la' *
|[------>      .      .      .      .      .| [0:1] NP -> DET * N ADJ
|[------>      .      .      .      .      .| [0:1] NP -> DET * N
|.      [------]      .      .      .      .| [1:2] N  -> 'fille' *
|[------------->      .      .      .      .| [0:2] NP -> DET N * ADJ
|[-------------]      .      .      .      .| [0:2] NP -> DET N *
|[------------->      .      .      .      .| [0:2] S  -> NP * VP
|.      .      [------]      .      .      .| [2:3] V  -> 'porte' *
|.      .      .      [------]      .      .| [3:4] DET -> 'une' *
|.      .      .      [------>      .      .| [3:4] NP -> DET * N ADJ
|.      .      .      [------>      .      .| [3:4] NP -> DET * N
|.      .      .      .      [------]      .| [4:5] N  -> 'robe' *
|.      .      .      [------------->      .| [3:5] NP -> DET N * ADJ
|.      .      .      [-------------]      .| [3:5] NP -> DET N *
|.      .      .      [------------->      .| [3:5] S  -> NP * VP
|.      .      .      .      .      [------]| [5:6] ADJ -> 'rouge' *
|.      .      .      [--------------------]| [3:6] NP -> DET N ADJ *
|.      .      .      [--------------------->| [3:6] S  -> NP * VP
```

1.1.1 Exercice

Étendre la grammaire ci-dessus pour inclure :

1. les verbes intransitifs (par exemple : *la fille dort*),
2. les compléments d'objet indirect (par exemple : *la fille parle à un ami*).

Attention : la grammaire doit produire *la fille dort* mais pas **la fille dort la robe* ou **la fille dort à la fille*. De même, on doit obtenir *la fille parle à un ami* mais pas **la fille parle une robe*. De même on ne doit pas obtenir *la fille porte*. Par contre *la fille parle* est autorisé.

Attention : mettre au début du fichier Python la ligne

```
# -*- coding: utf-8 -*
```

sinon le mot «*à*» vous causera des problèmes de codage...

1.2 Grammaires à structures de traits

La grammaire que l'on a écrite va produire aussi bien la phrase *la fille parle à un ami* que la phrase **la fille parle à un robe*.

Ce n'est pas raisonnable. Qu'est-ce qui ne va pas avec cette phrase ? Ce n'est évidemment pas le fait qu'une fille parle à une robe (après tout, pourquoi pas ?) mais plutôt l'*accord* entre l'adjectif et le nom (**un robe*).

Comment faire alors pour que les différentes parties du discours s'accordent ? (il y a le nombre, la personne, le genre, le cas, etc.). On ne va pas s'amuser à créer des symboles pour toutes les combinaisons : le russe, par exemple, a 3 nombres, 3 genres et 6 cas, ce qui ferait 54 symboles pour chaque nom ou d'adjectif, sans parler des cas où certains informations ne seraient pas connues...

C'est là où les structures de traits viennent à la rescousse. Il s'agit d'attacher à chaque symbole de la grammaire des paires trait-valeur qui serviront à imposer un accord en se limitant à certaines productions.

Ainsi, par exemple, au lieu d'écrire

```
DET -> 'la' | 'une' | 'un'
```

on écrira

```
DET[GENRE=fem] -> 'la' | 'une'
DET[GENRE=mas] -> 'un'
```

ce qui permettra déjà de distinguer «un» et «une». À cela s'ajoute l'utilisation de variables pour spécifier que dans une dérivation, certains symboles doivent avoir la même valeur d'un trait donné. Ici, par exemple :

```
NP -> DET[GENRE=?g] N[GENRE=?g] ADJ[GENRE=?g] | DET[GENRE=?g] N[GENRE=?g]
```

1.2.1 Exercice

Faire en sorte que la grammaire produise les phrases
- *les filles parlent à un ami*
- *les filles portent des robes rouges*

mais pas les phrases
- **les filles parle à une ami*
- **les filles parlent à une ami*
- **la fille portent un robes rouge*

Petit détail technique : *pour que notre grammaire soit reconnue comme une grammaire à traits, il faut changer l'extension du fichier* `grammaire.cfg` *en* `grammaire.fcfg`.

Rappel : le verbe s'accorde avec le sujet, mais pas avec le COD ou le COI.

1.3 Le λ-calcul

Essayons maintenant d'utiliser un formalisme de logique du premier ordre. On pourrait, par exemple, obtenir à partir de la phrase *la fille dort*, la formule logique `dort(fille)`. Sachant que l'arbre syntaxique de cette phrase est `(S (NP (DET la) (N fille)) (VP dort))` et que `NP` correspond à *la fille* et `VP` à *dort*, on voudrait avoir un moyen de combiner le verbe et le nom pour obtenir la formule logique souhaitée.

Ne pouvant pas écrire, en tant que formule logique, «dort» tout seul [1] ou «dort(.)», on utilise la notation $\lambda x.\text{dort}(x)$ (ou `\x.dort(x)` sous Python) pour dire «la fonction qui à x associe dort(x)» (en notation mathématique, ceci correspond à $x \mapsto \text{dort}(x)$). On appelle λx (`\x.` sous Python) un λ-*opérateur* et l'opération une λ-*abstraction*. Attention : l'antislash `\` étant un caractère spécial sous Python, il faut écrire des «chaînes brutes», du type

```
r'\x.dort(x)'
```

Notons que l'antislash utilisé ici n'est que la manière de Python de représenter la lettre grecque λ par un symbole de la table ASCII. On lira donc «lambda x dort x».

Puisque `\x.dort(x)` est une fonction, on peut l'appliquer à un objet. On obtient donc que

```
\x.dort(x) (gerald)
```

est la même chose que

```
dort(gerald)
```

et on appelle cette opération, une β-réduction.

Le λ-opérateur peut être appliqué à un prédicat, par exemple :

```
\P.(P(x) & dort(x)) (\x.fille(x))
```

1. Comme en mathématiques, où on écrira sin pour la fonction sinus et sin(x) pour sa valeur au point x.

(où & représente la conjonction $\wedge$) est la même chose que

```
fille(x) & dort(x)
```

où on a donc «remplacé le prédicat `P` par le prédicat `fille`».

On peut également combiner plusieurs λ-opérateurs :

```
\x y.(homme(x) & femme(y) & aime(x,y)) (roméo) (juliette)
```

est la même chose que

```
homme(roméo) & femme(juliette) & aime(roméo,juliette)
```

1.4 Dernière ligne droite : on combine tout !

Nous allons maintenant combiner les grammaires formelles à structures de traits avec le λ-calcul pour convertir des phrases en des formules logiques.

L'astuce consiste à utiliser un trait appelé `SEM` (comme SÉMantique). Chaque symbole de la grammaire (sauf les symboles terminaux qui sont des simples mots) portera l'attribut `SEM` correspondant. En partant des feuilles et en allant vers la racine de l'arbre syntaxique, ces sémantiques vont se combiner grâce aux β-réductions et former une seule formule logique, qui représentera la phrase de départ.

Exemple : prenons la phrase *Alice dort*. Pour obtenir `dort(alice)`, une première tentative serait d'écrire

```
% start S
S[SEM=<?vrb(?suj)>] -> NP[SEM=?suj] VP[SEM=?vrb]
NP[SEM=?suj] -> N[SEM=?suj]
N[SEM=<alice>] -> 'Alice'
VP[SEM=?vrb] -> IV[SEM=?vrb]
IV[SEM=<\x.dort(x)>] -> 'dort'
```

En lançant le programme Python

```
# -*- coding: utf-8 -*
import nltk

parser = nltk.load_parser("file:test.fcfg",trace=1)
for tree in parser.parse("Alice dort".split()):
    print tree
```

on obtient le retour

```
|.Alice. dort.|
|[-----]     .| [0:1] 'Alice'
|.     [-----]| [1:2] 'dort'
|[-----]     .| [0:1] N[SEM=<alice>] -> 'Alice' *
|[-----]     .| [0:1] NP[SEM=<alice>] -> N[SEM=<alice>] *
|[------>    .| [0:1] S[SEM=<?suj(?vrb)>] -> NP[SEM=?suj] * VP[SEM=?vrb]
                {?suj: <ConstantExpression alice>}
|.     [-----]| [1:2] IV[SEM=<\x.dort(x)>] -> 'dort' *
|.     [-----]| [1:2] VP[SEM=<\x.dort(x)>] -> IV[SEM=<\x.dort(x)>] *
|[==========]| [0:2] S[SEM=<alice(\x.dort(x))>] -> NP[SEM=<alice>] VP[SEM=<\x.dort(x)>] *
(S[SEM=<alice(\x.dort(x))>]
  (NP[SEM=<alice>] (N[SEM=<alice>] Alice))
  (VP[SEM=<\x.dort(x)>] (IV[SEM=<\x.dort(x)>] dort)))
```

(Expliquer !)

Autrement dit : ce n'est pas dort(alice) que l'on a obtenu, mais alice(λx.dort(x)), ce qui peut s'expliquer par le fait que la grammaire française met le sujet avant le verbe. Comment modéliser «Alice» pour que, tout en étant à gauche du prédicat λx.dort(x), il nous permette d'obtenir la formule dort(alice) ?

Faire de même pour les phrases : *Alice aime Gérald* et *Gérald aime Alice*. Attention : Alice et Gérald ne pourront rester des constantes mais vont devenir des λ-expressions.

Dans le TP 2 on ira plus loin dans la modélisation de la langue naturelle.