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

BY NC ND

UE LALOG

TP Logique et langage naturel (mercredi 20 mai 2020)

Yannis Haralambous (IMT Atlantique)

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

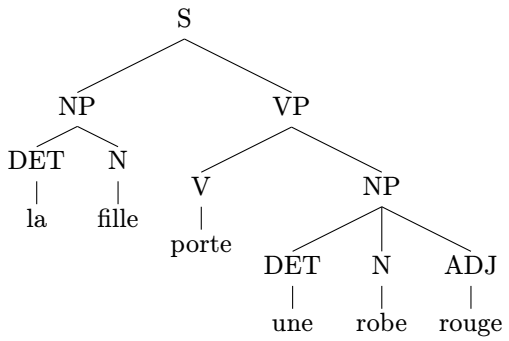
Dans cette section nous aurons besoin de deux outils non vus en cours : les *grammaires formelles à structures de traits* et le *λ-calcul*.

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 :

```
# -*- coding: utf-8 -*
import nltk
sent = "la fille porte une robe rouge".split()
parser = nltk.load_parser("file:grammaire.cfg",trace=1)
```

```
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...

SOLUTION

```
% start S
S -> NP VP
NP -> DET N ADJ | DET N
VP -> IV | ATV | TV NP | ATV PREP NP
```

```
DET -> 'la' | 'une' | 'un'
TV -> 'porte'
IV -> 'dort'
ATV -> 'parle'
N -> 'fille' | 'robe' | 'ami'
ADJ -> 'rouge'
PREP -> 'à'
```

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.

SOLUTION

```
% start S
S -> NP[NOMBRE=?n] VP[NOMBRE=?n]
NP[NOMBRE=?n,GENRE=?g] -> DET[NOMBRE=?n,GENRE=?g] N[NOMBRE=?n,GENRE=?g] ADJ[NOMBRE=?n,GENRE=?g]
    | DET[NOMBRE=?n,GENRE=?g] N[NOMBRE=?n,GENRE=?g]
VP[NOMBRE=?n] -> IV[NOMBRE=?n] | ATV[NOMBRE=?n] PREP NP | ATV[NOMBRE=?n] | TV[NOMBRE=?n] NP
DET[NOMBRE=sng,GENRE=fem] -> 'la' | 'une'
DET[NOMBRE=sng,GENRE=mas] -> 'un'
DET[NOMBRE=plu] -> 'les' | 'des'
TV[NOMBRE=sng] -> 'porte'
IV[NOMBRE=sng] -> 'dort'
ATV[NOMBRE=sng] -> 'parle'
N[NOMBRE=sng,GENRE=frm] -> 'fille' | 'robe'
N[NOMBRE=sng,GENRE=mas] -> 'ami'
ADJ[NOMBRE=sng] -> 'rouge'
TV[NOMBRE=plu] -> 'portent'
IV[NOMBRE=plu] -> 'dorment'
ATV[NOMBRE=plu] -> 'parlent'
N[NOMBRE=plu] -> 'filles' | 'robes' | 'amis'
ADJ[NOMBRE=plu] -> 'rouges'
PREP -> 'à'
```

(Suite du TP «Logique et langage naturel»)

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 $\lambda x.\mathrm{dort}(x)$ (ou `\x.dort(x)` sous Python) pour dire «la fonction qui à x associe $\mathrm{dort}(x)$» (en notation mathématique, ceci correspond à $x \mapsto \mathrm{dort}(x)$). 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))
```

(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 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=<?suj(?vrb)>] -> 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'
```

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

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) ?

SOLUTION La solution est de modéliser Alice par $\lambda Q.Q(\text{alice})$. Ainsi, avec

```
% start S
S[SEM=<?suj(?vrb)>] -> NP[SEM=?suj] VP[SEM=?vrb]
NP[SEM=?suj] -> N[SEM=?suj]
N[SEM=<\Q.Q(alice)>] -> 'Alice'
N[SEM=<\Q.Q(alice)>] -> 'Gérald'
VP[SEM=?vrb] -> IV[SEM=?vrb]
VP[SEM=<?vrb(?nom)>] -> TV[SEM=?vrb] NP[SEM=?nom]
IV[SEM=<\x.dort(x)>] -> 'dort'
TV[SEM=<\R x.R(\y.aime(x,y))>] -> 'aime'
```

on obtient

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

qui correspond.

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.

SOLUTION Il faut d'abord régler la transitivité du verbe, par l'ajout d'une règle

```
VP[SEM=<?vrb(?nom)>] -> TV[SEM=?vrb] NP[SEM=?nom]
```

Il faut donc que «aime» puisse récupérer Gérald (son COD). Comment l'écrire ?

Pour répondre à cette question, voyons ce qu'on veut obtenir et faisons du «reverse engineering». Quand Gérald sera récupéré par «aime», on n'aura plus qu'un prédicat à une place, c'est-à-dire

$$\lambda x.\mathrm{aime}(x, \mathrm{gerald}).$$

On cherche donc une expression Ψ pour «aime», qui, placée à gauche de $\lambda Q.Q(\mathrm{gerald})$ donne

$$\lambda x.\mathrm{aime}(x, \mathrm{gerald}).$$

Si Ψ est placé à gauche, alors forcément Ψ doit commencer par un λ-opérateur de prédicat pour pouvoir être appliqué à $\lambda Q.Q(\mathrm{gerald})$. Et comme on veut récupérér un $\lambda x.\mathrm{aime}(x, \mathrm{gerald})$, cet opérateur sera suivi d'un λx, qui va rester. Ensuite on écrira le prédicat, ce qui aura comme effet qu'on appliquera $\lambda Q.Q(\mathrm{gerald})$ au prédicat «aime» vu comme fonction de son deuxième argument, pour y placer Gérald :

$$\lambda Q.Q(\mathrm{gerald})\ (\lambda y.\mathrm{aime}(x, y)).$$

Donc on peut en conclure que

$$\Psi \equiv \lambda R.\lambda x.R(\lambda y.\mathrm{aime}(x, y)).$$

Preuve :

$$\begin{aligned}\lambda R.\lambda x.R(\lambda y.\mathrm{aime}(x, y))\ (\lambda Q.Q(\mathrm{gerald})) &\equiv \lambda x.[\lambda Q.Q(\mathrm{gerald})\ (\lambda y.\mathrm{aime}(x, y))] \\ &\equiv \lambda x.[(\lambda y.\mathrm{aime}(x, y))\ \mathrm{gerald}] \\ &\equiv \lambda x.\mathrm{aime}(x, \mathrm{gerald}).\end{aligned}$$

Donc

```
TV[SEM=<\R x.R(\y.aime(x,y))>] -> 'aime'
```

Solution finale :

```
% start S
S[SEM=<?suj(?vrb)>] -> NP[SEM=?suj] VP[SEM=?vrb]
NP[SEM=?suj] -> N[SEM=?suj]
N[SEM=<\Q.Q(alice)>] -> 'Alice'
N[SEM=<\Q.Q(gerald)>] -> 'Gérald'
VP[SEM=?vrb] -> IV[SEM=?vrb]
VP[SEM=<?vrb(?nom)>] -> TV[SEM=?vrb] NP[SEM=?nom]
IV[SEM=<\x.dort(x)>] -> 'dort'
TV[SEM=<\R x.R(\y.aime(x,y))>] -> 'aime'
```

Retour :

```
|.Alic.aime.Gér.|
|[----]    .    .| [0:1] 'Alice'
|.    [----]    .| [1:2] 'aime'
|.    .    [----]| [2:3] 'G\xc3\xa9rald'
|[----]    .    .| [0:1] N[SEM=<\Q.Q(alice)>] -> 'Alice' *
|[----]    .    .| [0:1] NP[SEM=<\Q.Q(alice)>] -> N[SEM=<\Q.Q(alice)>] *
|[---->    .    .| [0:1] S[SEM=<?suj(?vrb)>] -> NP[SEM=?suj] * VP[SEM=?vrb]
                   {?suj: <LambdaExpression \Q.Q(alice)>}
|.    [----]    .| [1:2] TV[SEM=<\R x.R(\y.aime(x,y))>] -> 'aime' *
|.    [---->    .| [1:2] VP[SEM=<?vrb(?nom)>] -> TV[SEM=?vrb] * NP[SEM=?nom]
                   {?vrb: <LambdaExpression \R x.R(\y.aime(x,y))>}
|.    .    [----]| [2:3] N[SEM=<\Q.Q(gerald)>] -> 'G\xc3\xa9rald' *
```

```
|.    .    [----]| [2:3] NP[SEM=<\Q.Q(gerald)>] -> N[SEM=<\Q.Q(gerald)>] *
|.    .    [---->| [2:3] S[SEM=<?suj(?vrb)>] -> NP[SEM=?suj] * VP[SEM=?vrb]
                   {?suj: <LambdaExpression \Q.Q(gerald)>}
|.    [---------]| [1:3] VP[SEM=<\x.aime(x,gerald)>] -> TV[SEM=<\R x.R(\y.aime(x,y))>] NP[SEM=<\Q.Q(gerald)>] *
|[==============]| [0:3] S[SEM=<aime(alice,gerald)>] -> NP[SEM=<\Q.Q(alice)>] VP[SEM=<\x.aime(x,gerald)>] *
aime(alice,gerald)
```

1.5 Théorie des types

Astuce : pour savoir comment formaliser les différents mots d'une phrase, voyons ce qu'ils deviennent lorsqu'on interprète la formule (dans le sens d'«interprétation de formule logique» comme on l'a vu en cours).

Un exemple : dans la phrase *Gérard aime Alice*, *Gérard* est naturellement interprété par un élément «Gérard» du domaine D, et de même pour *Alice*. L'interprétation du prédicat binaire *aime* peut être considérée comme une application $D^2 \to \{\text{vrai}, \text{faux}\}$ (elle envoie la paire d'entités (Gérard, Alice) vers la valeur vrai si Gérard aime Alice et vers faux sinon).

Autre exemple : dans la phrase *Gérard aime Alice et Paul déteste Virginie*, la particule de coordination «*et*» va combiner deux phrases pour en produire une nouvelle, son interprétation sera donc une application $\{\text{vrai}, \text{faux}\} \times \{\text{vrai}, \text{faux}\} \to \{\text{vrai}, \text{faux}\}$ dont la table de valeurs correspond à celle du connecter $\wedge$.

La situation se complique encore plus dans le cas des adverbes : dans *Gérard aime beaucoup Alice*, l'adverbe *beaucoup* agit sur le verbe, donc on peut considérer qu'il transforme une application $D^2 \to \{\text{vrai}, \text{faux}\}$ en une autre application $D^2 \to \{\text{vrai}, \text{faux}\}$.

Et que dire alors des modificateurs d'adverbe comme *vraiment beaucoup* : en effet, *vraiment* agit sur *beaucoup* et est donc un transformateur de transformateur d'application $D^2 \to \{\text{vrai}, \text{faux}\}$...

Comment gérer cette complexité qui semble croître inexorablement ?

Voici une modélisation mathématique qui nous délivre du cauchemar décrit ci-dessus : considérons qu'il n'existe que deux *types sémantiques primitifs* : celui de «formule» (dont l'interprétation dans D sera «vrai» ou «faux») et celui de «valeur» (dont l'interprétation sera un élément du domaine D). Notons t les formules et e les valeurs. Ensuite prenons le monoïde libre $\{e, t\}^*$ dont nous notons la loi comme un produit scalaire $\langle , \rangle$[2].

Appelons les éléments de ce monoïde des *types sémantiques complexes*. Nous allons faire *correspondre chaque sommet de l'arbre syntaxique à un type sémantique complexe*, et ceci de la manière suivante : l'élément de gauche de $\langle , \rangle$ est la «donnée d'entrée» du type, son élément de droite est sa «donnée de sortie».

Exemple : un prédicat unaire, comme *dort*, s'applique à une constante, son interprétation fournit une valeur de $\{\text{vrai}, \text{faux}\}$. On dira donc qu'il est de type $\langle e, t \rangle$ (= «il prend un e et il nous rend un t»).

Mais attention : on ne prend qu'un seul élément d'entrée à la fois ! Un prédicat *binaire* ne sera donc pas de type $\langle \{e, e\}, t \rangle$ (cette syntaxe n'est pas valide) mais sera décrit de *manière récursive* comme $\langle e, \langle e, t \rangle \rangle$.

Si on y réfléchit un peu, c'est parfaitement logique : un prédicat binaire auquel on fournit une valeur devient prédicat unaire, donc pour une entrée e, la sortie est $\langle e, t \rangle$. De même, le type d'un prédicat ternaire sera $\langle e, \langle e, \langle e, t \rangle \rangle \rangle$ et ainsi de suite...

De la même manière, la particule de conjonction *et* sera de type $\langle t, \langle t, t \rangle \rangle$, l'adverbe *beaucoup* de type $\langle \langle e, t \rangle, \langle e, t \rangle \rangle$, et le modificateur d'adverbe *vraiment*, de type $\langle \langle \langle e, t \rangle, \langle e, t \rangle \rangle, \langle \langle e, t \rangle, \langle e, t \rangle \rangle \rangle$. Arrivé à ce stade, le lecteur/la lectrice doit normalement se sentir ébloui(e) devant l'époustouflante beauté de ce modèle : en effet, *quelque soit* le type sémantique d'un mot, aussi complexe soit-il, on peut le décrire simplement par un élément du monoïde libre $\{e, t\}^*$... c'est simple et efficace.

Sur la fig. 1 le lecteur peut constater que

- à chaque fois qu'un sommet n'a qu'un seul enfant, le type sémantique ne change pas, et

[2] Attention : cette loi *n'est pas* associative, donc pas question de faire des «simplifications» : $\langle e, \langle e, t \rangle \rangle$ n'est pas la même chose que $\langle \langle e, e \rangle, t \rangle$!

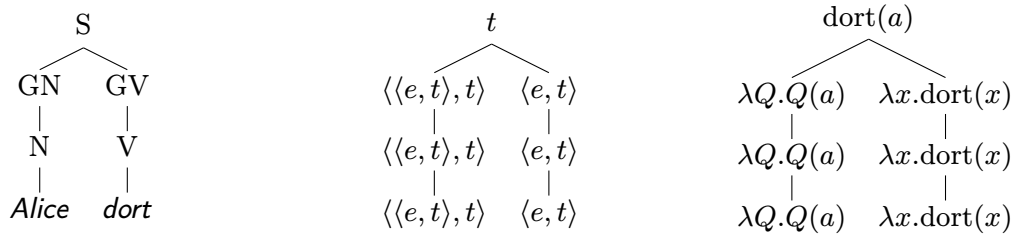


Figure 1: Arbre syntaxique de la phrase *Alice dort*, arbre de ses types sémantiques, arbre de la traduction en formalisme logique.

- à chaque fois qu'un sommet a plusieurs enfants, leurs types sémantiques se *composent* : ainsi, $\langle\langle e,t\rangle,t\rangle$ appliqué à $\langle e,t\rangle$ donne t. On note $\times$ cette composition : $\langle e,t\rangle \times e = t$. Pour qu'une composition $T_1 \times T_2$ puisse avoir lieu, il faut que T_1 soit un type complexe et que sa première composante soit égale à T_2.

La cohérence sémantique d'une phrase provient du fait que les types sémantiques des sommets de son arbre syntaxique se composent correctement, pour arriver au type de S qui est, invariablement, t (comme on peut le constater sur le graphe de la fig. 1).

NLTK peut nous calculer les types à partir des formules logiques. En écrivant

```
from nltk.sem.logic import *
tlp = LogicParser(True)
print(tlp.parse(r'\Q.Q(alice)').type)
print(tlp.parse(r'\x.dort(x)').type)
print(tlp.parse(r'\Q.Q(alice) (\x.dort(x))').type)
```

on a le retour

```
<<e,?>,?>
<e,?>
?
```

Le point d'interrogation vient du fait que le parseur ne sait pas si «Q» et «dort» sont des prédicats (et donc de type t) ou des fonctions (type e). Autre fonctionnalité intéressante : NLTK peut faire des β-réductions pour nous. Ainsi, si on se demande que peut bien signifier $\lambda Q.Q(\mathrm{alice})(\lambda x.\mathrm{dort}(x))$, il suffit d'écrire

```
from nltk.sem.logic import *
lexpr = Expression.fromstring
print(lexpr(r'\Q.Q(alice) (\x.dort(x))').simplify())
```

et on a la réponse :

```
dort(alice)
```

On peut ainsi vérifier et nos types et nos réductions.

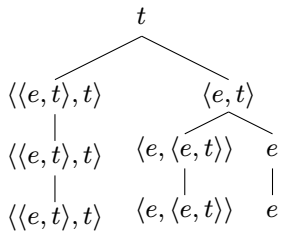
1.5.1 Exercice sur l'amour

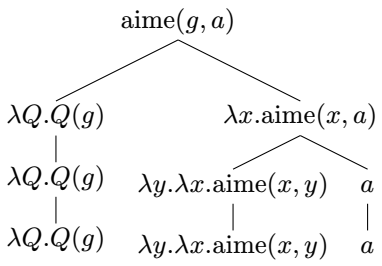
En se basant sur les résultat de la première partie du TP, calculer l'arbre des types de la phrase *Gérard aime Alice* (fig. 2) de la manière la plus simple possible. Vérifier sous Python NLTK.

On remarquera qu'alors *Gérard* et *Alice* ne seront pas modélisés de la même manière. Reprendre les calculs en considérant que *Alice* est modélisée de la même manière que *Gérard*. Cela donnera une formule plus complexe pour *aime*, mais permet un traitement plus uniforme.

SOLUTION

Premier cas



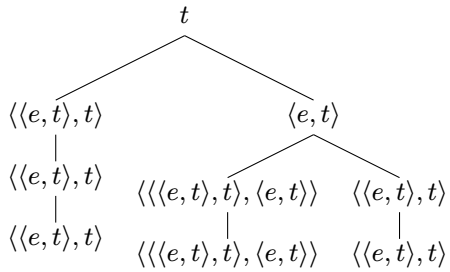


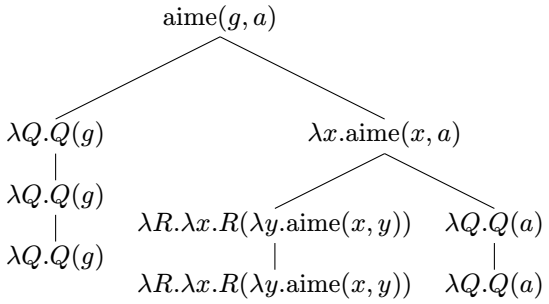

```
from nltk.sem.logic import *
lexpr = Expression.fromstring
print(lexpr(r'\y (\x.aime(x,y)) (alice)').simplify())
print(lexpr(r'\Q.Q(gerald) (\y (\x.aime(x,y)) (alice))').simplify())
```

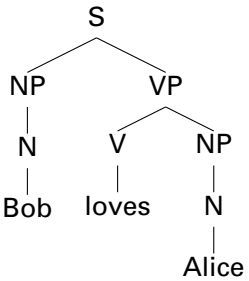
donne bien

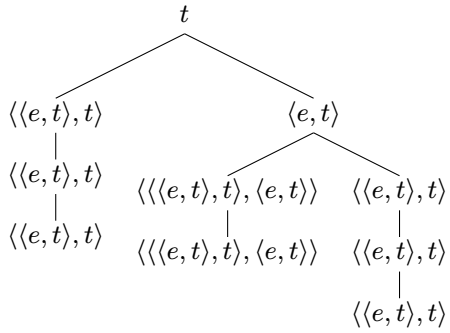
```
\x.aime(x,alice)
aime(gerald,alice)
```

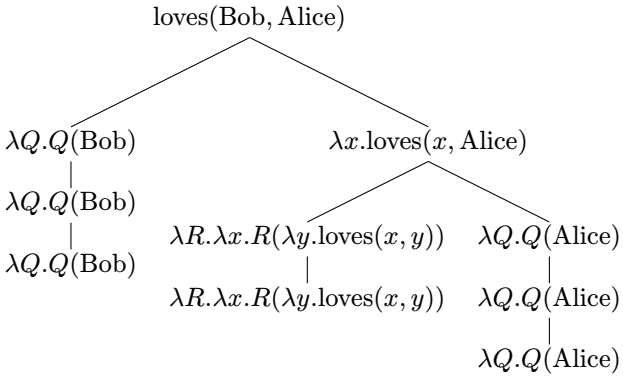
Deuxième cas












```
from nltk.sem.logic import *
lexpr = Expression.fromstring
print(lexpr(r'\R x.R(\y.aime(x,y)) (\Q.Q(alice))').simplify())
print(lexpr(r'\Q.Q(gerald) (\R x.R(\y.aime(x,y)) (\Q.Q(alice)))').simplify())
```

donne bien

```
\x.aime(x,alice)
aime(gerald,alice)
```

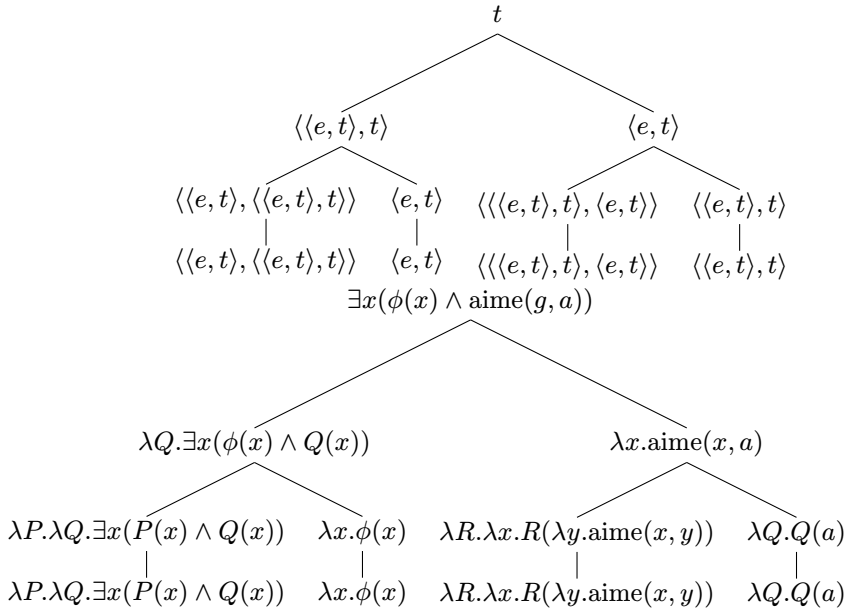
Figure 2: Arbres syntaxiques des phrases *Gérard aime Alice* et *un philosophe aime Alice*.

1.6 L'article indéfini

Prenons la phrase *un philosophe aime Alice* (fig. 2). Notons tout de suite que *philosophe* ne peut être traduit par une constante, comme, par exemple *Gérard*, puisque «être philosophe» est une propriété, et les propriétés sont naturellement traduites par des prédicats unaires. Ainsi, on écrira philosophe(g) pour dire que g est philosophe. De même, $\exists x\,\text{philosophe}(x)$ signifie qu'il existe un philosophe. Et donc, sa traduction en formalisme logique est $\exists x\,(\text{philosophe}(x) \wedge \text{aime}(x, a))$.

En donner l'arbre des types et l'arbre des formalisations logiques. Vérifier sous NLTK.

SOLUTION




```
from nltk.sem.logic import *
tlp = LogicParser(True)
print(tlp.parse(r'(\Q.exists x(phi(x) & Q(x))) (\y.aime(y,a))').type)
print(tlp.parse(r'(\P.\Q.exists x.(P(x) & Q(x))) (\x.phi(x))').type)
lexpr = Expression.fromstring
print(lexpr(r'(\Q.exists x(phi(x) & Q(x))) (\y.aime(y,a))').simplify())
print(lexpr(r'(\P.\Q.exists x.(P(x) & Q(x))) (\x.phi(x))').simplify())
```

donne

```
t
<<e,t>,t>
exists x.(phi(x) & aime(x,a))
\Q.exists x.(phi(x) & Q(x))
```

1.7 L'article défini

Mais comment traduire alors l'article défini *le*, dans la phrase *le philosophe aime Alice* ?

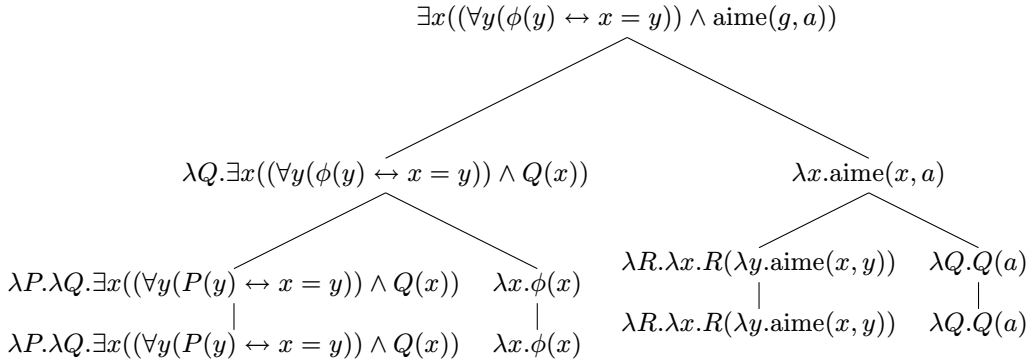
Formulons la question autrement : comment indiquer qu'il n'y a qu'*un seul* philosophe, et que quand on dit *le philosophe* on parle justement de lui ?

Voici comment décrire l'*unicité* : il n'y a qu'un seul philosophe x si et seulement si pour tout individu y tel que y soit philosophe, on ait $x = y$.

La phrase *le philosophe aime Alice* se traduira donc par $\exists x\,(\forall y\,(\text{philosophe}(y) \leftrightarrow x = y) \wedge \text{aime}(x, a))$.

En donner l'arbre des types et l'arbre des formalisations logiques. Vérifier sous NLTK. Comment justifier le fait qu'un mot aussi simple et aussi élémentaire a une traduction en formalisme logique aussi complexe ?

SOLUTION L'arbre des types ne change pas




```
from nltk.sem.logic import *
tlp = LogicParser(True)
print(tlp.parse(r'(\Q.exists x((forall y.(phi(y) <-> x=y)) & Q(x))) (\y.aime(y,a))').type)
print(tlp.parse(r'(\P.\Q.exists x.((forall y.(P(y) <-> x=y)) & Q(x))) (\x.phi(x))').type)
lexpr = Expression.fromstring
print(lexpr(r'(\Q.exists x((forall y.(phi(y) <-> x=y)) & Q(x))) (\y.aime(y,a))').simplify())
print(lexpr(r'(\P.\Q.exists x.((forall y.(P(y) <-> x=y)) & Q(x))) (\x.phi(x))').simplify())
```

donne

```
t
<<e,t>,t>
exists x.(all y.(phi(y) <-> (x = y)) & aime(x,a))
\Q.exists x.(all y.(phi(y) <-> (x = y)) & Q(x))
```

La complexité de la formalisation se justifie par le potentiel de ce mot...