Formation LIESSE 2021
Langages formels et logique du premier ordre sous Python

Yannis Haralambous (IMT Atlantique)

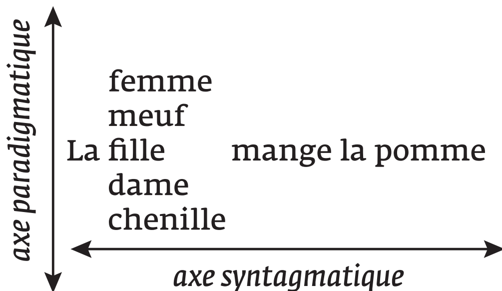
12 mai 2021

partie

Approches formelles

Structuralisme de Jacobson

- Un ingrédient fondamental du structuralisme de Jacobson est l'opposition entre *axe syntagmatique* et *axe paradigmatique*.
- On définit les *unités élémentaires* d'un domaine et ensuite on les place dans ces deux axes :



- Cela suppose que l'on a, outre les unités élémentaires, aussi le *moyen de les combiner*.

Section 1

Langages formels

Formalisation

- Pour *formaliser* des situations, nous allons donc examiner :
  1. les *éléments* d'un système,
  2. la manière de les *combiner*,
  3. les *combinaisons* permises par la situation.
- Exemples :
  - l'étude de l'affichage d'une calculatrice
    1. éléments : graphèmes `0`, ..., `9`, `+`, `-`, ×, etc.,
    2. manière de combiner : concaténation,
    3. combinaisons permises : `5+3`, non permises : `7+`×`3`, etc.) ;
  - l'étude de l'ADN...
  - l'étude morphosyntaxique d'une phrase...
  - l'étude du jeu d'échecs...
  - l'interprétation d'une image de route par un véhicule autonome...

Langages formels

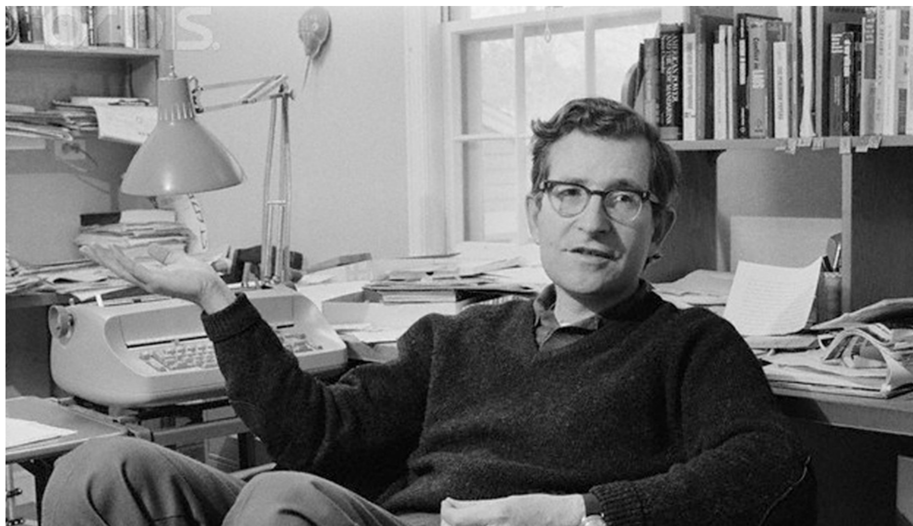
- On met les unités élémentaires du système que l'on souhaite décrire dans un ensemble que l'on appelle (de manière très eurocentrique) *alphabet (formel)*.
- On appelle les membres de cet ensemble, des *lettres (formelles)*.
- Pour les combiner on utilise un·des *opérateur·s de concaténation*.
- Un ensemble de lettres formelles concaténées est appelé un *mot (formel)*.
- Un *langage formel* est un ensemble de mots formels.

- Mathématiquement, on note, par exemple, $A = \{a, b, c, \ldots\}$ un alphabet, $\cdot$ (ou vide) l'opérateur de concaténation (quand il n'y a qu'un seul), ab, $bbac$, etc., les mots (le mot vide est noté ε), et L un langage sur A.

Exemples de langages formels

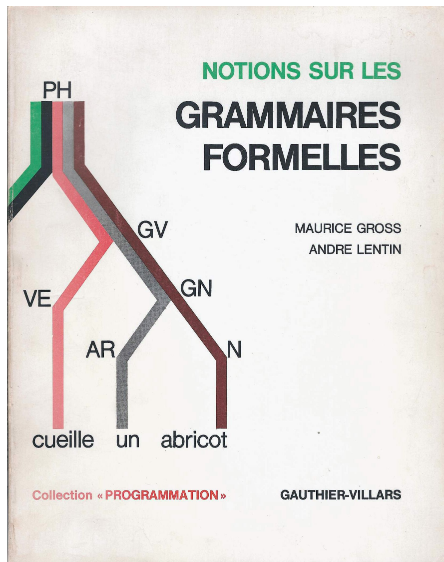
- Dans la théorie des langages formels, toute la difficulté consiste dans la description d'un langage.
- Un langage fini peut être défini de manière *extensionnelle*, c'est-à-dire en énumérant tous ses mots, e.g. $L = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$;
- Quand il est infini ou très grand, on utilise des définitions *intensionnelles*, c'est-à-dire l'énumération de ses propriétés : « le langage de tous les mots qui commencent par n lettres a et se poursuivent par le même nombre de lettres b », on se sert aussi de motifs de régularité : $L = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$.
- Les descriptions intensionnelles conviennent plutôt à l'humain qu'à la machine. Comment faire alors ?

Noam Chomsky



Maurice Gross et André Lentin

Les langages formels font depuis longtemps partie du quotidien (et de l'imaginaire) des informaticiens, bien avant la démocratisation de l'informatique.
La preuve : en France, un ouvrage fondamental, *Notions de grammaires formelles* de Maurice Gross et André Lentin est sorti déjà en 1967. Gross était prof à l'X et il a fréquenté le MIT (et donc, Chomsky), Lentin était prof à Paris Vet sous-directeur de l'Institut Blaise Pascal.



Les grammaires formelles de Chomsky

- Inspiré par le langage naturel, Chomsky introduit un outil de description des langages, appelé *grammaire formelle*.
- Une grammaire formelle comporte quatre types d'ingrédients :
  1. des *symboles terminaux* (= les membres de l'alphabet du langage à décrire) qu'on notera, par exemple, $a, b, c, \ldots$,
  2. des *symboles non-terminaux* qu'on notera, par exemple, $A, B, C, \ldots$,
  3. des *règles de production* qui envoient des mots (qui comportent au moins un non-terminal) vers d'autres mots,
  4. le choix d'un non-terminal que l'on appelle *axiome de départ* (noté S).
- Selon la liberté qu'on laisse aux règles de production, on a une hiérarchie de grammaires formelles.

Hiérarchie de Chomsky

- Une règle de production s'écrit $\alpha \to \omega$ (on note les mots par des lettres grecques), en sachant que dans α il doit y avoir au moins un non-terminal.
- On peut donc toujours écrire α comme $\beta A\gamma$, où A est un non-terminal, et c'est ce non-terminal qui est ré-écrit. Une règle quelconque s'écrit donc $\boxed{\beta A\gamma \to \omega}$.

Hiérarchie de Chomsky

- On dira que la grammaire est de *type 0* (ou *récursivement énumérable*) s'il n'y a aucune condition sur β, γ, ω.
- On dira qu'elle est de *type 1* (ou *contextuelle*) si ω est de la forme $\beta\psi\gamma$ (donc : « même contexte des deux côtés » puisque la règle s'écrit $\beta A\gamma \to \beta\psi\gamma$).
- On dira qu'elle est de *type 2* (ou *hors-contexte*) si β et γ sont toujours vides (donc : pas de « contexte »).
- On dira qu'elle est de *type 3* (ou *régulière*) si β et γ sont vides et ω est de type a ou aB ou Ba (où B est non-terminal et a est terminal).

Lien entre grammaire et langage

- Une grammaire *reconnaît* les mots d'un langage de la manière suivante :
  si on peut appliquer un certain nombre de règles de production en commençant par l'axiome de départ et aboutir à un certain mot (qui ne contient donc que des terminaux), alors on dira que le mot est *reconnu* par la grammaire.
- Exemple : montrons que le mot ab est reconnu par la grammaire $(\{a, b\}, \{S\}, \{S \to aS, S \to b\}, S)$:
  en appliquant d'abord $S \to aS$ à S on obtient aS, en appliquant $S \to b$ au S qui se trouve dans aS, on obtient ab, CQFD.
  Est-ce que aab et abb sont reconnus par cette grammaire ?
- Une série d'applications successives de règles de production est appelée une *dérivation*.

ATTENTION

- ATTENTION : la grammaire formelle propose des règles, l'humain (ou la machine) choisit quelles règles appliquer et à quelle partie du mot.
- Chaque règle de production est appliquée à *un seul non-terminal* (ce n'est pas un « chercher-remplacer global »). Si je veux obtenir $BBBB$ à partir de $AAAA$ à l'aide de la règle de production $A \to B$, je dois l'appliquer quatre fois et c'est moi qui choisis à quelles instances de A je l'applique, e.g. :

$$AAAA \xrightarrow{S} ABAA \xrightarrow{S} ABAB \xrightarrow{S} BBAB \xrightarrow{S} BBBB$$

ATTENTION

- ATTENTION : la difficulté ne réside pas seulement dans le fait de reconnaître les mots du langage souhaité, mais aussi dans celui de *ne pas reconnaître les mots qui n'appartiennent pas au langage*.
- Ainsi, avec les règles $S \to aS$, $S \to Sb$, $S \to \varepsilon$, on aura bien sûr tous les mots

$$\underbrace{a \cdots a}_{n \text{ fois}} \underbrace{b \cdots b}_{m \text{ fois}},$$

mais *impossible* d'avoir *uniquement* les mots

$$\underbrace{a \cdots a}_{n \text{ fois}} \underbrace{b \cdots b}_{n \text{ fois}}.$$

- Pour cela il faut une grammaire hors-contexte $S \to aSb, S \to \varepsilon$.

Langages réguliers et hors-contexte

- Dans la plupart des applications (en particulier, en TAL) on utilisera des langages réguliers ou hors-contexte.
- Les langages réguliers sont les plus simples, ils peuvent également être décrits par des *automates de type fini* et par des *expressions régulières*.
- Les langage hors-contexte son moins contraignants que les langages réguliers. Ils peuvent être décrits également par des *automates à pile*.
- Un exemple de langage qui est hors-contexte mais régulier est $L = \{a^n b^n \mid n \geq 0\}$. Ses règles de production sont $\{S \to aSb, S \to \varepsilon\}$, la première n'est pas régulière.

Expressions régulières

- Les *expressions régulières* sont un formalisme pour décrire des langages réguliers.
- Dans cette notation on trouve : les lettres et les mots du langage formel, mais aussi des *quantificateurs*, un *opérateur de disjonction* `|`, des *parenthèses* `( )`.
- Les quantificateurs sont `?` (zéro ou une fois), `+` (une ou plusieurs fois, `*` (zéro, une ou plusieurs fois).
- Le langage $\{b, ab, aab, aaab, \ldots\}$ est représenté par `a*b`. Le langage {*pure*, *pire*, *pore*} est représenté par `p(u|i|o)re`.
- L'expression `(p|m)(u|û)re?` représente-t-elle le langage {*pur,mur,pure,mûre*} ?
- Les langages de programmation proposent des expressions régulières un peu plus évoluées (POSIX).

Expressions régulières POSIX

- On considère que l'alphabet est le codage Unicode.
- On note `\r` le retour-chariot, `\n` le newline, `\t` la tabulation, `\uabcd` le caractère Unicode de position `0xABCD`, `\U0001f60e` le caractère Unicode de position `0x1F60E` (c'est-à-dire le 😎).
- On utilise le backslash `\` pour protéger des caractères, y compris le backslash lui-même.
- Sur l'axe paradigmatique on utilise des *intervalles* : `[0-9]`, `[A-EG-Za-k]`, `[^a-zA-Z]`, `.` (équivalent à `[^\r\n]`), `[a-z-]`, `[\[\](){}]`, etc.

Expressions régulières POSIX

- Sur l'axe syntagmatique, on concatène avec ou sans parenthèses, et on sert de *quantificateurs* pour indiquer des quantités : `?` (0 ou 1 fois), `*` (0 ou 1 ou plusieurs fois), `+` (1 ou plusieurs fois), `{n}` (*n* fois), `{p,q}` (entre *p* et *q* fois, inclus), `{p,}` (*p* fois ou plus).
- Les quantificateurs `?`, `+`, etc., sont *gourmands*, dans le sens qu'ils vont reconnaître la chaîne la plus longue. Si on les fait suivre par `?` ils deviennent anorexiques : il reconnaissent la chaîne la plus courte. Exemple : `<b>(.*?)</b>`.
- On peut utiliser l'opérateur de disjonction `|` : `(bla|bli)`, `M(on|a) (frère|sœur) adorée?`, etc.

Expressions régulières POSIX

- On peut réutiliser des expressions parenthésées à l'aide de backslash suivis de numéro : `(bla|bli)\1`, `([a-z]{2})([a-z]{2})\2\1`, etc.

  Uniquement pour les *grands-maîtres ès expressions régulières* :

- Si une expression parenthésée ne doit pas être mémorisée, on écrira `(?:` à la place de la parenthèse ouvrante.
- Le *lookbehind* : pour ne reconnaître `abc` que s'il est précédé de `def` on écrira `(?<=def)abc`. En négatif : `(?<!def)abc`.
- Le *lookahead* : pour ne reconnaître `abc` que s'il est suivi de `def` on écrira `abc(?=def)`. En négatif : `abc(?!def)`.

TP sur les expression régulières I

Expressions régulières

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 complier 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 :

TP sur les expression régulières II

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.

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 » :

TP sur les expression régulières III

- `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 » ;

TP sur les expression régulières IV

- 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),

TP sur les expression régulières V

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` ;

TP sur les expression régulières VI

- 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.

Utilisation des expressions régulières sous Python

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 :

TP sur les expression régulières VIII

```
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())
```

TP sur les expression régulières IX

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.

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)
```

TP sur les expression régulières X

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 :

TP sur les expression régulières XI

```
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.

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.

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()
```

Exercice 1

Que fait le code suivant ?

TP sur les expression régulières XIV

```
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.

Exercice 2

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

SOLUTION

TP sur les expression régulières XV

```
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()
```

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()
```

TP sur les expression régulières XVI

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()
```

Exercice 4

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

TP sur les expression régulières XVII

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")
```

TP sur les expression régulières XVIII

```
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)
```

TP sur les expression régulières XIX

La réponse est 13.

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

TP sur les expression régulières XX

```
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)
```

TP sur les expression régulières XXI

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

Expressions régulières ludiques

Le site `regexcrossword.com` propose des mots-croisés en expressions régulières, de difficulté croissant :

	[ABC]*(.)\1(ME|UO)	(.)T*E*\1	[HAS]*(SN|PA)	(WE|GA|AL)T*O+	(EG|BEEE)[WIQ]*
[QA].[WEST]*					
(HE|RT|TK)*.					
(RE|QR)[QUART]*					
[EUW]*S[RITE]*					
(.)(.)\2\1[WE]					

C'est extrêmement ludique et addictif. Malheureusement il n'y a ni certificat après résolution de tous les exercices, ni tournoi, ni championnat mondial. Pour le moment.

Expressions régulières poétiques

Les expressions régulières ont aussi servi à la poésie moderne (Waber, 2008) :

```
I need /t(w?o1,2) w?r(i|a|ough)te?/.
```

Expressions régulières poétiques

À l'aide des expressions régulières on peut ré-écrire de manière plus concise certains chefs d'œuvres poétiques contemporains qui ont bercé notre jeunesse :

Isabelle a les yeux bleus
Isabelle a les yeux bleus
Isabelle a les yeux bleus
Bleus les yeux Isabelle a

Isabelle a les yeux bleus
Isabelle a les yeux bleus
Bleus les yeux Isabelle a
Isa bleus belle a les yeux

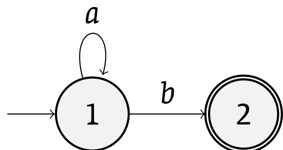
s'écrirait (modulo la casse) :

```
((Isa)(belle) a ((les yeux) (bleus)))\n\1\n\1\n\6 \5
\2\3 a\n\1\n\1\n\6 \5 \2\3 a\n\2 \6 \3 a \5
```

(Codage de Huffman)

Automates de type fini

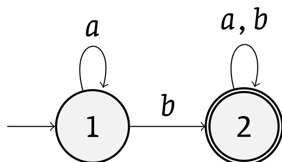
- Un *automate de type fini* (ATF) sur un langage formel L, est un quadruplet :
  1. un ensemble d'*états*,
  2. un ensemble de *transitions* entre les états, déclenchées par des lettres du langage
     (autrement dit : lorsqu'on se trouve en un état donné, une lettre donnée déclenche une ou plusieurs transition·s vers lui-même et/ou vers d'autres états),
  3. une sélection d'au moins un état *initial*,
  4. une sélection d'au moins un état *terminal*.
- Exemple : un automate qui reconnaît tous les mots du langage $\{b, ab, aab, aaab, \ldots\}$ est :



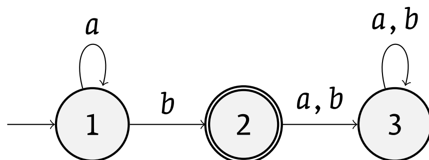
Mais reconnaît-il *uniquement* les mots de ce langage ?

Automates complets

- Un ATF est dit *complet* si toutes les combinaisons d'états et de lettres apparaissent en tant que transitions.
  Ainsi, en écrivant

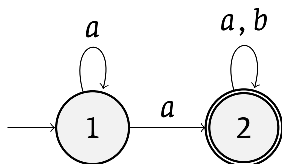


on reconnaît les mots de $L = \{b, ab, aab, aaab, \ldots\}$ mais aussi d'autres mots, comme *aababab*. Pour représenter *exactement* le langage L il faut ajouter un autre état, non terminal :



Automates déterministes complet

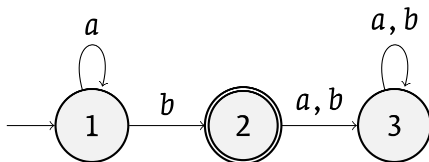
- Un automate de type fini est dit *déterministe complet* si (a) il n'a qu'un seul état initial, et (b) pour un état et une lettre donnés, il n'existe qu'une seule transition possible.
- Exemple d'automate non-detérministe :



qui reconnaît les mots qui commencent par au moins un a.
- Le non-déterminisme complique l'implémentation des automates.
- Heureusement il existe un algorithme (dit de *déterminisation*) qui permet de créer un automate déterministe à partir de tout automate non-déterministe.

Table de transitions

- Tout automate de type fini est décrit par une *table de transitions* : les lignes représentent les états, les colonnes les lettres, une flèche dénote les états initiaux, un souligné les terminaux.



- Exemple :
  est entièrement décrit par

	a	b
→1	1	2
$\underline{2}$	3	3
3	3	3

TP sur les automates I

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

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

TP sur les automates II

```
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',
```

TP sur les automates III

```
('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',
```

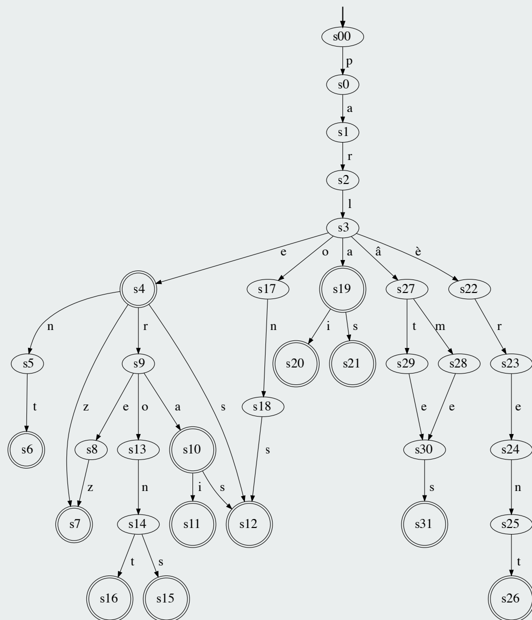
TP sur les automates IV

```
('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')))
```

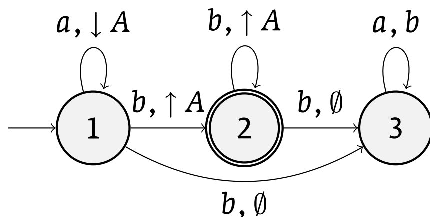
TP sur les automates V

```
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/')
```

TP sur les automates VI



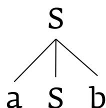
- Exemple : pour reconnaître un mot de $\{a^n b^n \mid n \geq 1\}$ les règles de transition vont empiler des symboles A pour chaque a et les dépiler pour chaque B :



où l'acceptation se fait quand à la fin du mot on est sur 2 avec une pile vide. Les symboles $\downarrow A$, $\uparrow A$ et $\emptyset$ signifient : « empiler A », « dépiler A », « si pile vide ».

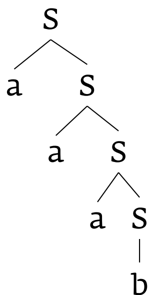
Arbres syntaxiques

- Revenons aux grammaires formelles, et plus particulièrement aux grammaires hors-contexte.
- Comme les règles de production n'ont qu'un seul élément à gauche (un non-terminal), on peut représenter leur application par une branche d'arbre.
  Par exemple, appliquer la règle $S \to aSb$ à S peut être représenté par



Arbres syntaxiques

- Autre exemple : la dérivation qui à travers les règles $\{S \to aS, S \to b\}$ reconnaît *aaab* s'écrira



- Les feuilles de cet arbre, lues dans l'ordre induit par les règles, forment le mot reconnu.
- On l'appelle *arbre syntaxique* du mot.

Caveat canem

- Attention : une règle du type $A \to aAa$ stipule que A peut être réécrit entouré de deux éléments a, qui sont des instances de la même lettre a.
- Mais une règle $A \to NAN$ avec derrière $\mathrm{N} \to a$ et $\mathrm{N} \to b$ *ne garantit pas* que A sera entouré par les mêmes terminaux : on peut très bien réécrire N par a la première fois et par b la deuxième.
- Attention aussi : si on ne fait pas attention, une grammaire peut *ne générer aucun mot* (fini). Par exemple dans le cas de la grammaire $(\{a, b\}, \{S\}, \{S \to aS, S \to bS\}, S)$ on ne peut se débarrasser du S.

  (Si on ajoute à celle-ci la règle $S \to \varepsilon$, on tombe dans l'autre extrême : on obtient la *totalité* des mots sur $\{a, b\}$.)

Lex/Yacc

- Deux outils légendaires servent à créer des compilateurs : *Lex* (Lexical Analyzer) et *Yacc* (Yet Another Compiler Compiler).
- Lex se sert d'expressions régulières et d'un automate à états fini pour détecter des tokens de certains types, fait des calculs de valeur et renvoie le tout (type et valeur) à Yacc.
- Yacc reçoit les données de Lex, vérifie qu'elles sont syntaxiquement valides et ensuite fait des calculs à partir des valeurs reçues, et ceci pour chaque règle production.
- Lex et Yacc ont été développés dans les années 70 en langage C. Il en existe des versions dans d'autres langages et, en particulier, en Python (e.g., package PLY).

Exemple : la grammaire d'une calculette

EXPR $\rightarrow$ EXPR (PLUS|MINUS) EXPR
EXPR $\rightarrow$ TERM
TERM $\rightarrow$ TERM (TIMES|DIVIDE) FACTOR
TERM $\rightarrow$ FACTOR
FACTOR $\rightarrow$ NUMBER
FACTOR $\rightarrow$ LPAREN EXPR RPAREN

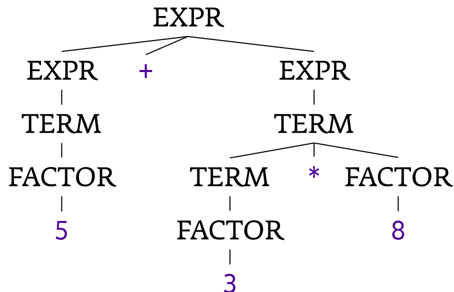
NUMBER $\rightarrow$ `[0-9]+`
PLUS $\rightarrow$ `+`
MINUS $\rightarrow$ `-`
TIMESS $\rightarrow$ `*`
DIVIDE $\rightarrow$ `/`
LPAREN $\rightarrow$ `(`
RPAREN $\rightarrow$ `)`

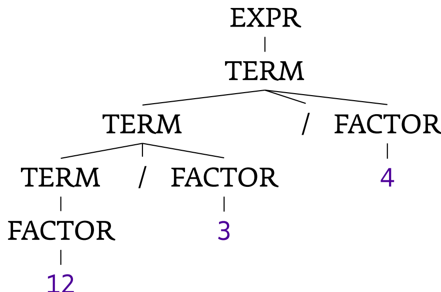
Pourquoi avoir des symboles EXPR, TERM et FACTOR ?

Exemple : la grammaire d'une calculette

Pourquoi avoir des symboles EXPR, TERM et FACTOR ?

Réponse : pour respecter la priorité des opérations.
Exemples : 5+3*8 et 12/3/4.





La dite calculette sous Python I

Le fichier `calclex.py` :

```
import ply.lex as lex

tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

t_PLUS    = r'\+'
t_MINUS   = r'-'
```

La dite calculette sous Python II

```
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

t_ignore  = ' \t'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

La dite calculette sous Python III

```
lexer = lex.lex()
```

Le fichier `calcyacc.py` :

```
import ply.yacc as yacc

from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

La dite calculette sous Python IV

```
def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]
```

La dite calculette sous Python V

```
def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

def p_error(p):
    print("Syntax error in input!")

parser = yacc.yacc()

while True:
   try:
       s = input('calc > ')
```

La dite calculette sous Python VI

```
    except EOFError:
        break
    if not s: continue
    result = parser.parse(s)
    print(result)
```

Pour l’utiliser, lancer

```
python calcyacc.py
```

Pour l’arrêter, taper CTRL-D.

Un exemple de langage formel contextuel

- Le *langage de copie* : la première moitié du mot est identique à la deuxième.
- Voici sa grammaire, sur l'alphabet $\{\mathbf{a}, \mathbf{b}\}$:

$$S \xrightarrow{(1)} D_1D_2T \qquad A_2E \xrightarrow{(4)} E\mathbf{a} \qquad A_1F \xrightarrow{(7)} F\mathbf{a}$$

$$T \xrightarrow{(2)} A_1A_2T \mid B_1B_2T \mid E \qquad B_2E \xrightarrow{(5)} E\mathbf{b} \qquad B_1F \xrightarrow{(8)} F\mathbf{b}$$

$$\alpha\beta \xrightarrow{(3)} \beta\alpha \qquad D_2E \xrightarrow{(6)} F \qquad D_1F \xrightarrow{(9)} \varepsilon$$

pour tout $\alpha, \beta \in \{A, B, D\}$.

- Exemple : le mot **abab** est reconnu comme suit :

$$\underline{S} \xrightarrow{(1)} D_1D_2\underline{T} \xrightarrow{(2a)} D_1D_2A_1A_2\underline{T} \xrightarrow{(2b)} D_1D_2A_1A_2B_1B_2\underline{T} \xrightarrow{(2c)} D_1\underline{D_2A_1}A_2B_1B_2E$$
$$\xrightarrow{(3)} D_1A_1D_2\underline{A_2B_1}B_2E \xrightarrow{(3)} D_1A_1\underline{D_2B_1}A_2B_2E \xrightarrow{(3)} D_1A_1B_1D_2A_2\underline{B_2E}$$
$$\xrightarrow{(5)} D_1A_1B_1D_2\underline{A_2E}\mathbf{b} \xrightarrow{(4)} D_1A_1B_1\underline{D_2E}\mathbf{ab} \xrightarrow{(6)} D_1A_1\underline{B_1F}\mathbf{ab}$$
$$\xrightarrow{(8)} D_1\underline{A_1F}\mathbf{bab} \xrightarrow{(7)} \underline{D_1F}\mathbf{abab} \xrightarrow{(9)} \mathbf{abab}.$$

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

cc 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

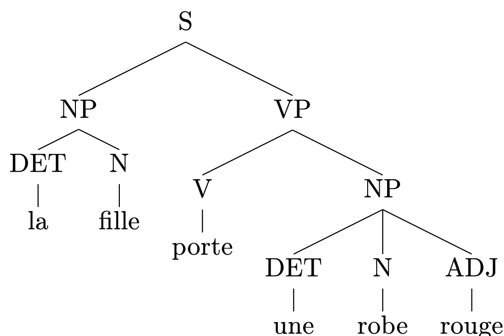
Dans cette section nous aurons besoin de deux outils non vus en cours : les *grammaires formelles à structures de traits* et le *λ-calcul*.

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'
```

Les grammaires planaires

- Sproat (*A Computational Theory of Writing Systems*, 2000), définit une *grammaire planaire* comme une grammaire disposant de cinq opérateurs de concaténation distincts

$$\stackrel{\rightarrow}{\cdot} \quad \stackrel{\leftarrow}{\cdot} \quad \stackrel{\downarrow}{\cdot} \quad \stackrel{\uparrow}{\cdot} \quad \odot$$

(et de parenthèses, puisqu'il n'y a pas d'associativité) ;

- par exemple, le caractère chinois 鳞 peut être décrit comme suit par rapport à ses composantes :

$$\text{鱼} \stackrel{\rightarrow}{\cdot} [\text{米} \stackrel{\downarrow}{\cdot} [\text{夕} \stackrel{\rightarrow}{\cdot} \text{㐄}]].$$

(Publicité : Richard Sproat sera orateur invité au colloque *Grapholinguistics in the 21st Century 2022*, cd. https://grafematik2022.sciencesconf.org/.)

Exemple de grammaire planaire I

Les règles de dérivation de l'écriture syllabique coréenne hangoul sont les suivantes [p. 43]sproat :

1. si σ_1 et σ_2 sont des syllabes, $\gamma(\sigma_1 \cdot \sigma_2) := \gamma(\sigma_1) \overset{\rightarrow}{\cdot} \gamma(\sigma_2)$;
2. pour l'attaque-noyau $\omega\nu$ et la coda κ, $\gamma(\omega\nu \cdot \kappa) := \gamma(\omega\nu) \overset{\downarrow}{\cdot} \gamma(\kappa)$;
3. quand la coda κ est complexe : $\kappa = \kappa_1 \cdot \kappa_2$, alors $\gamma(\kappa_1 \cdot \kappa_2) := \gamma(\kappa_1) \overset{\rightarrow}{\cdot} \gamma(\kappa_2)$;
4. pour une attaque ω et un noyau ν, either
   (a) $\gamma(\omega \cdot \nu) := \gamma(\omega) \overset{\rightarrow}{\cdot} \gamma(\nu)$, si ν appartient à l'ensemble de jamos verticaux, ou
   (b) $\gamma(\omega) \overset{\downarrow}{\cdot} \gamma(\nu)$, si ν appartient à l'ensemble de jamo horizontaux ;
5. (règle ajoutée par nous) quand le noyau ν est complexe : $\nu = \nu_1 \cdot \nu_2$, où ν_1 est horizontal et ν_2 est vertical, alors nous appliquons d'abord la règle 4(a) à $\omega\nu_1 \cdot \nu_2$ et ensuite la règle 4(b) à $\omega \cdot \nu_1$.

Exemple de grammaire planaire II

En guise d'illustration, appliquons ces règles à la syllabe coréenne 굃 : elle consiste en

- une attaque ㄱ,
- un noyau contenant deux jamo ㅗ et ㅣ le premier desquels est horizontal et le deuxième vertical, et
- d'une coda consistant elle aussi en deux jamo ㄹ et ㅎ.
- Selon la règle 5, nous appliquons d'abord la règle 4(a) à [ㄱㅗ]·ㅣ pour obtenir [[ㄱㅗ]$\overset{\rightarrow}{\cdot}$ㅣ] et ensuite la règle 4(b) à ㄱ·ㅗ, pour obtenir [[ㄱ$\overset{\downarrow}{\cdot}$ㅗ]$\overset{\rightarrow}{\cdot}$ㅣ].
- Ensuite nous appliquons la règle 3 à la coda ㄹ·ㅎ pour obtenir [ㄹ$\overset{\rightarrow}{\cdot}$ㅎ],
- et enfin la règle 2 pour réunir la paire attaque-noyau et la coda, afin d'obtenir

  [[ㄱ$\overset{\downarrow}{\cdot}$ㅗ]$\overset{\rightarrow}{\cdot}$ㅣ]$\overset{\downarrow}{\cdot}$[ㄹ$\overset{\rightarrow}{\cdot}$ㅎ],

  en tant que décomposition de 굃.

Les grammaires visuelles de type SR

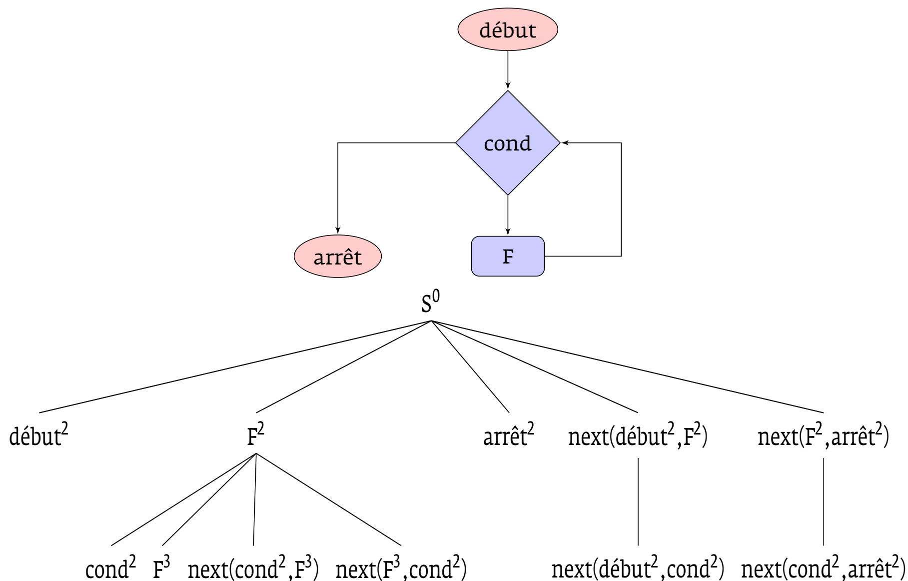
- Les *grammaires SR* (Filomena Ferrucci, 1996) généralisent les grammaires formelles de Chomsky.
- Les instances de la même lettre sont numérotées, on n'écrit donc plus *aabba* mais l'ensemble $\{a^1, a^2, a^3, b^1, b^2\}$.
- Pour décrire la « position » de chaque instance de lettre on utilise non pas un seul, mais plusieurs opérateurs de « concaténation », ce sont des relations binaires entre lettres numérotées.
- Ainsi, par exemple, pour décrire le mot classique *abba* (source d'inspiration pour les AᗺBA), on définira un opérateur de concaténation « next » et on écrira les deux ensembles :

$$M = \{a^1, a^2, b^1, b^2\}$$

$$R = \{\text{next}(a^1, b^1), \text{next}(b^1, b^2), \text{next}(b^2, a^2)\}.$$

- L'intérêt étant qu'on peut faire BEAUCOUP plus de choses.

Les grammaires visuelles de type SR



Section 2

Systèmes formels

Les systèmes formels

Un *système formel* est la donnée :

(1) d'un « *alphabet* » Σ fini ou dénombrable,

(2) d'un sous-ensemble F de l'ensemble Σ^* des suites finies d'éléments de Σ, on appelle F l'ensemble des *énoncés bien formés*,

(3) d'un *système déductif*, c'est-à-dire

  (3a) d'un sous-ensemble A de F, appelé ensemble des *axiomes*,

  (3b) d'un ensemble fini R d'applications $F^n \to F$ appelées *règles d'inférence*.

Les conditions (1) et (2) sont celles d'un langage formel. C'est la condition (3) qui permet d'aller plus loin.

Ingrédients des systèmes formels

- Les *énoncés bien formés F* sont les combinaisons de symboles de Σ qui respectent une certaine syntaxe (= les mots).
- À noter que l'ensemble des énoncés bien formés peut être défini par une grammaire formelle. Ne pas confondre les règles de production de la grammaire formelle et les règles d'inférence du système formel !
- Les *règles d'inférence* permettent de déduire de nouveaux énoncés bien formés à partir d'énoncés existant.
- Les *axiomes* sont des énoncés que l'on a choisi de ne pas déduire d'autres énoncés, ils servent d'énoncés de départ à partir desquels on déduit tous les autres.

Règles d'inférence

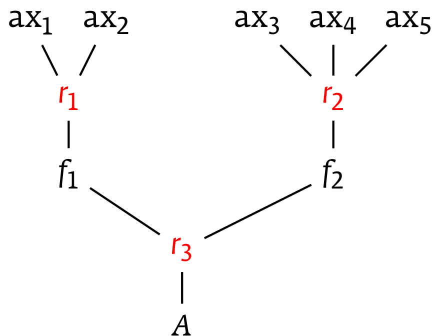
- Une *règle d'inférence* r produit un énoncé g à partir de n énoncés $f_1, \ldots, f_n$;
- on peut donc la considérer comme une application $F^n \to F$;
- ou alors une application de F^{n+1} dans l'ensemble {vrai, faux} ;
- on écrira

$$\frac{f_1, f_2, \ldots, f_n}{g} \quad r$$

- ou alors $f_1, f_2, \ldots, f_n \vdash_r g$,
- et on lira « g peut être déduit de $f_1, f_2, \ldots, f_n$ par la règle d'inférence r ».

Preuves

- Une *preuve* dans un système déductif est un graphe biparti avec comme partitions les règles R et les énoncés S,
- les feuilles appartiennent à S et sont des axiomes.
- si A est la racine de l'arbre, on dira que c'est un *théorème*, et on écrira $\vdash A$.



Section 3

Logique du 1er ordre

Ingrédients de la logique du 1er ordre

La logique du 1er ordre est un *système formel* dont l'*alphabet* est constitué :

- de *prédicats* n-aires $P, Q, R, \ldots,$
- de *fonctions* n-aires $f, g, h, \ldots,$
- de *variables* $X, Y, Z, \ldots,$
- de *connecteurs* $\neg$ (« négation ») et $\to$ (« implication »),
- du *quantificateur universel* $\forall$ (« pour tout »),
- du signe de l'*égalité* $=$,
- et des *parenthèses* ( ).

Le cas particulier du prédicat 0-aire est appelé une *proposition*, et le cas de la fonction 0-aire est appelé une *constante*.

Mais à quoi cela sert-il ?

Culture générale : les modes de raisonnement

- La *déduction* : si A est vrai et si A vrai ⇒ B vrai, alors B est vrai.
- L'*abduction* : si B est vrai et si A vrai ⇒ B vrai, alors A est vrai. (Exemple : les ânes sont des êtres vivants, je suis un être vivant, donc je suis un âne.)

  Mais aussi : *elle consiste, lorsque l'on observe un fait dont on connaît une cause possible, à conclure à titre d'hypothèse que le fait est* probablement *dû à cette cause-ci* (Wikipédia).
- L'*induction* : si A et B sont vrais, alors A vrai ⇒ B vrai. (Exemple : chez les enfants, avoir des grands pieds favorise l'aptitude en orthographe.)

  Mais aussi : *genre de raisonnement qui se propose de chercher des lois générales à partir de l'observation de faits particuliers*, sur une base probabiliste (Wikipédia).

Modus popens, modus tollens

Tous les félins sont des mammifères.
Un chat est un félin.
Donc un chat est un mammifère.

Tous les félins sont des mammifères.
Un cabillaud n'est pas un mammifère.
Donc un cabillaud n'est pas un félin.

Modus popens, modus tollens

Toutes les blacies sont des chazomares.
Une élêthiote est une blacie.
Donc une élêthiote est une chazomare.

Toutes les blacies sont des chazomares.
Une exypnade n'est pas une chazomare.
Donc une exypnade n'est pas une blacie.

Modus popens, modus tollens

Toutes les blacies sont des chazomares. $\forall X, P(X) \to Q(X)$
Une élêthiote est une blacie. $P(A)$
Donc une élêthiote est une chazomare. $\vdash Q(A)$

Toutes les blacies sont des chazomares. $\forall X, P(X) \to Q(X)$
Une exypnade n'est pas une chazomare. $\neg Q(B)$
Donc une exypnade n'est pas une blacie. $\vdash \neg P(B)$

Donnée ≠ Information ≠ Connaisssance

- Nous utilisons tous les termes « donnée », « information », « connaissance » dans la vie courante.
- Mais ils ont aussi une signification plus technique, utilisée en intelligence artificielle :
- les *données* sont des ensembles ordonnés de nombres, provenant de capteurs ou de programmes de simulation, de générateurs de données aléatoires, etc. ;
- les *informations* sont des données auxquelles on a attaché une signification : on sait ce que représente tel nombre ou tel ensemble d'octets ou telle chaîne de caractères ;
- les *connaissances* sont des motifs ou des tendances que l'on extrait des informations, dans le but de faire des prédictions.
- Exemple : le départ d'un train de la gare de Brest.

Nos outils de travail

- L'intelligence humaine opère à travers des *processus de raisonnement* basés sur des *représentations (internes) de la connaissance*.
- En intelligence artificielle on crée des *bases de connaissances*.
- Celles-ci contiennent des *énoncés* exprimés dans des *langages de représentation des connaissances*, comme par exemple :
  - les différents types de logique : propositionnelle, du 1er ordre (FOL), du 2e ordre, modale, floue, temporelle, etc., mais aussi
  - les ontologies,
  - les graphes conceptuels,
  - certains langages contrôlés, etc.
- On peut interroger une base de connaissances : pour obtenir la réponse à une requête, on se sert également du processus d'inférence.

Nos outils de travail

Toutes les logiques se partagent les notions suivantes :

1. Une *syntaxe* du langage formel sous-jacent.
2. Un *système déductif*.
3. Une *théorie des modèles*, c'est-à-dire une notion d'*interprétation* (ou de « monde possible ») des énoncés (intuitivement : une manière de leur donner du sens dans un domaine choisi) qui les dote d'une valeur de vérité.

   Une interprétation qui rend un énoncé vrai est un *modèle* de celui-ci (ainsi, $x + y = 4$ est vraie dans un monde où $x = y = 2$ et fausse dans un monde où $x = 1, y = -1$).

   On dit qu'un énoncé β est *conséquence* d'un énoncé α ($\alpha \models \beta$) quand tout modèle de α est aussi modèle de β. Exemple : $(x + y = 2) \models (x + y > 0)$.

Grammaire BNF de la logique du 1er ordre

Règles syntaxiques

$$
\begin{aligned}
\text{Énoncé} &\rightarrow \text{ÉnoncéAtomique} \mid \text{ÉnoncéComplexe}\\
\text{ÉnoncéAtomique} &\rightarrow \text{Proposition} \mid \text{Prédicat(Terme+)}\\
&\quad \mid \text{Terme} = \text{Terme}\\
\text{ÉnoncéComplexe} &\rightarrow \text{(Énoncé)} \mid \neg\text{Énoncé} \mid \text{Énoncé} \rightarrow \text{Énoncé}\\
&\quad \mid \text{Quantificateur Variable, Énoncé}\\
\text{Terme} &\rightarrow \text{Fonction(Terme+)} \mid \text{Constante} \mid \text{Variable}
\end{aligned}
$$

Règles lexicales

$$
\begin{aligned}
\text{Quantificateur} &\rightarrow \forall\\
\text{Constante} &\rightarrow A \mid B \mid \text{Michel}\ldots\\
\text{Variable} &\rightarrow X \mid Y \mid Z\ldots\\
\text{Prédicat} &\rightarrow \text{aime()} \mid \text{existe()}\ldots\\
\text{Fonction} &\rightarrow \text{père()} \mid \text{carré()} \mid \text{sinus()}\ldots
\end{aligned}
$$

Précédence des opérateurs : $\forall$, $\neg$, $=$, $\rightarrow$.

Grammaire de la logique du 1er ordre

- Si on note $P, Q, R, \ldots$ les prédicats, $f, g, h, \ldots$ les fonctions, $a, b, c, \ldots$ les constantes, $X, Y, Z, \ldots$ les variables, alors :
- on peut écrire des *formules*
  - $P(X) \wedge Q(Y)$
  - $\forall X, P(X)$
  - $P(f(X), g(Y))$,
  - $f(X, Y) = g(Z)$,
  - $a = b$, etc.
- on peut écrire des *termes*
  - X,
  - $f(X)$,
  - $f(a, g(X, b))$, etc.
- par contre les expressions « $P(X) = Q(X)$ », « $P(Q(X))$ », « $f(P)$ », « $f(Y) = P(X)$ », « $\forall X, f(X)$ », « $\forall P, P(X)$ », etc., sont *syntaxiquement incorrectes*.

Symboles supplémentaires

- Pour faciliter la lecture des énoncés, ainsi que le passage des situations du monde réel aux énoncés logiques, on introduit quatre symboles supplémentaires :
  - le $\wedge$ (« et », *conjonction*),
  - le $\vee$ (« ou », *disjonction*) et
  - le (« si et seulement si », *double implication*), ainsi que
  - le quantificateur existentiel $\exists$ (« il existe »).
- Ils sont définis à partir des symboles existants de la manière suivante :
  - $A \vee B := \neg B \rightarrow A$,
  - $A \wedge B := \neg(\neg A \vee \neg B) = \neg(B \rightarrow \neg A)$,
  - $AB := (A \rightarrow B) \wedge (B \rightarrow A) = \neg((B \rightarrow A) \rightarrow \neg(A \rightarrow B))$,
  - $\exists x\, P := \neg\forall x\, \neg P$.

Grammaire BNF enrichie

Règles syntaxiques

$$
\begin{aligned}
\text{Énoncé} &\rightarrow \text{ÉnoncéAtomique} \mid \text{ÉnoncéComplexe} \\
\text{ÉnoncéAtomique} &\rightarrow \text{Proposition} \mid \text{Prédicat(Terme+)} \mid \text{Terme} = \text{Terr} \\
\text{ÉnoncéComplexe} &\rightarrow \text{(Énoncé)} \mid \neg\text{Énoncé} \mid \text{Énoncé} \rightarrow \text{Énoncé} \\
&\mid \text{Énoncé} \wedge \text{Énoncé} \mid \text{Énoncé} \vee \text{Énoncé} \\
&\mid \text{Énoncé} \leftrightarrow \text{Énoncé} \\
&\mid \text{Quantificateur Variable, Énoncé} \\
\text{Terme} &\rightarrow \text{Fonction(Terme+)} \mid \text{Constante} \mid \text{Variable}
\end{aligned}
$$

Règles lexicales

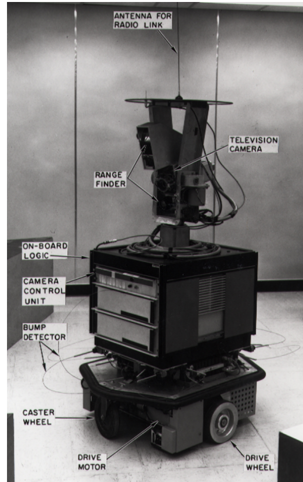
$$
\begin{aligned}
\text{Quantificateur} &\rightarrow \forall \mid \exists \\
\text{Constante} &\rightarrow A \mid X \mid \text{Michel}\ldots \\
\text{Variable} &\rightarrow a \mid x \mid s\ldots \\
\text{Prédicat} &\rightarrow \text{aime} \mid \text{existe}\ldots \\
\text{Fonction} &\rightarrow \text{Père} \mid \text{Carré} \mid \text{Sinus}\ldots
\end{aligned}
$$

Précédence des connecteurs : $\forall/\exists$, $\neg$, $=$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

Variables libres et liées

- Une variable quantifiée est appelée *liée*.
- Une variable est *libre* quand elle n'est pas quantifiée.
- Attention : dans certains énoncés mal écrits, on peut trouver le même symbole pour une variable liée et une variable libre, par exemple dans $P(x, y) \wedge \forall x\, Q(x, y)$. Dans cet énoncé, x est-elle libre ou liée ?
- Pour éviter ce genre de problème il convient de renommer les variables liées en utilisant des symboles qui ne figurent pas parmi les variables libres. Cet énoncé devient donc $P(x, y) \wedge \forall z\, Q(z, y)$, dans lequel x et y sont libres et z est liée.
- Un énoncé sans variables libres est appelé *énoncé clos* ou *sentence*.

Théorie des modèles



Aidez-moi à agir dans le monde ! (Et je ne suis pas un dalek !!)

Domaine, signature, structure

- Un *domaine* Δ est un ensemble quelconque dans lequel les variables vont prendre leurs valeurs ; on y définira les constantes, fonctions et prédicats à travers la *fonction d'interprétation*.
- Une *signature* σ est un ensemble de symboles de fonction, de constante et de prédicat, ainsi qu'une fonction qui envoie ces symboles dans $\mathbb{N}_0$ appelée *arité*.
- Une *structure* est un triplet $\mathcal{A} = (\Delta, \sigma, \mathbf{I})$ où Δ est un *domaine* (ou *univers*), σ est une *signature* et $\mathbf{I}$ est une *fonction d'interprétation*.

Fonction d'interprétation

Une *fonction d'interprétation* est une fonction $\mathbf{I}$ qui envoie :

1. les constantes de σ dans Δ, on notera $\mathbf{I}(c) = c^{\mathcal{A}} \in \Delta$;
2. toute fonction n-aire ($n \geq 1$) f de σ vers une fonction $\mathbf{I}(f) = f^{\mathcal{A}} : \Delta^n \to \Delta$;
3. toute prédicat n-aire ($n \geq 1$) P de σ vers une fonction $\mathbf{I}(P) = P^{\mathcal{A}} : \Delta^n \to \{\bot, \top\}$, où $\bot$ signifie « faux » et $\top$ signifie « vrai ».

- Ayant défini $\mathbf{I}$ sur les éléments de σ on peut l'étendre aux termes : on obtient des applications $t^{\mathbf{I}} : \Delta^j \to \Delta$ pour chaque terme ayant j variables.
- De même, on peut l'étendre aux énoncés. On obtient donc une application $\alpha^{\mathbf{I}} : \Delta^k \to \{\top, \bot\}$ pour α en supposant que α a k variables libres.

Exemple de fonction d'interprétation

- Soient les constantes Roger, Baxter, Rougette et Bleuette.
- Soient les prédicats unaires homme(.), femme(.), et les prédicats binaires aime(.,.) et déteste(.,.).
- On a le droit d'écrire des formules homme(Roger), femme(Baxter), aime(Roger,Roger), déteste(Bleuette,Roger), etc.
- Écrivons une fonction d'interprétation **I**.

Exemple de fonction d'interprétation

- D'abord il faut choisir un domaine : $\Delta = \{$🚹, 🚹, 🚺, 🚺$\}$.
- Ensuite envoyer toutes les constantes vers des éléments du domaine :

  I(Roger)=🚹
  I(Baxter)=🚹
  I(Rougette)=🚺
  I(Bleuette)=🚺

- Les prédicats unaires vers des fonctions $\Delta \to \{\top, \bot\}$:

I(homme) : 🚹 $\mapsto \top$	**I**(femme) : 🚹 $\mapsto \bot$
I(homme) : 🚹 $\mapsto \top$	**I**(femme) : 🚹 $\mapsto \bot$
I(homme) : 🚺 $\mapsto \bot$	**I**(femme) : 🚺 $\mapsto \top$
I(homme) : 🚺 $\mapsto \bot$	**I**(femme) : 🚺 $\mapsto \top$

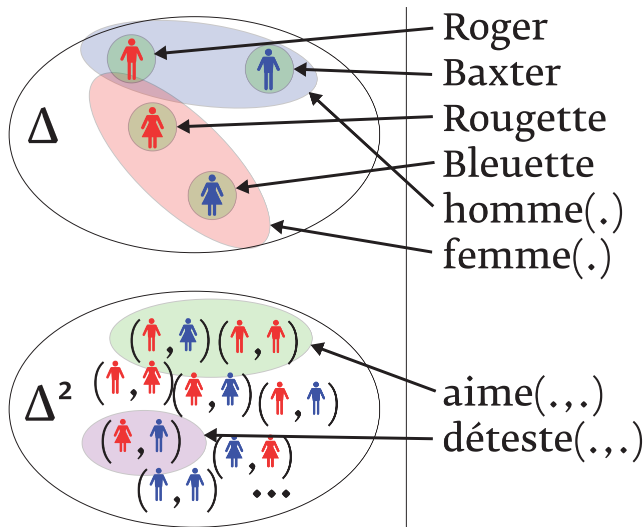
Exemple de fonction d'interprétation

- Et enfin, Les prédicats binaires vers des fonctions $\Delta^2 \to \{\top, \bot\}$:

$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚹}, \textcolor{red}{🚹}) \mapsto \top$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚹}, \textcolor{blue}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚹}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚹}, \textcolor{blue}{🚺}) \mapsto \top$
$\mathbf{I}(\text{aime}) : (\textcolor{blue}{🚹}, \textcolor{red}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{blue}{🚹}, \textcolor{blue}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{blue}{🚹}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{blue}{🚹}, \textcolor{blue}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚺}, \textcolor{red}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚺}, \textcolor{blue}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚺}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{aime}) : (\textcolor{red}{🚺}, \textcolor{blue}{🚺}) \mapsto \bot$

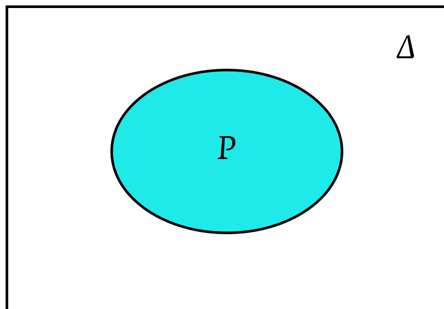
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚹}, \textcolor{red}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚹}, \textcolor{blue}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚹}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚹}, \textcolor{blue}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{blue}{🚹}, \textcolor{red}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{blue}{🚹}, \textcolor{blue}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{blue}{🚹}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{blue}{🚹}, \textcolor{blue}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚺}, \textcolor{red}{🚹}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚺}, \textcolor{blue}{🚹}) \mapsto \top$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚺}, \textcolor{red}{🚺}) \mapsto \bot$
$\mathbf{I}(\text{déteste}) : (\textcolor{red}{🚺}, \textcolor{blue}{🚺}) \mapsto \bot$, etc.

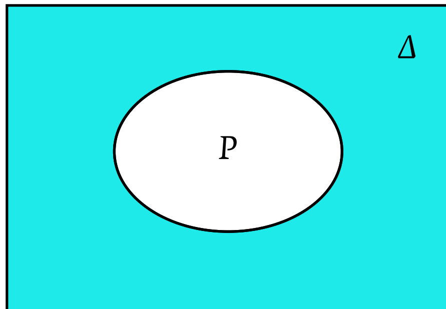
Exemple de fonction d'interprétation



Vision ensembliste de l'interprétation

- Si $\mathcal{A} = (\Delta, \sigma, \mathbf{I})$ est une structure, alors on peut associer à tout prédicat unaire P, le sous-ensemble $P^{\mathcal{A}}$ de Δ des $e \in \Delta$ tels que $P(e)$ soit vrai.
- Et $(\neg P)^{\mathcal{A}} = \Delta \setminus P^{\mathcal{A}}$.

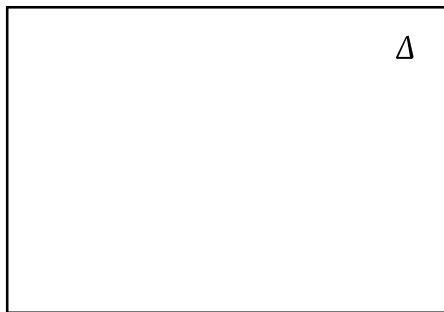




Vision ensembliste de l'interprétation

- Dans ce cas, si on dénote par $\top$ le prédicat toujours vrai, l'ensemble des $e \in \Delta$ pour lesquels $\top$ est vraie est tout Δ. De même, si $\bot$ est le prédicat toujours faux, c'est-à-dire l'ensemble des $e \in \Delta$ pour lesquels $\bot$ est vraie, cet ensemble est $\emptyset$.

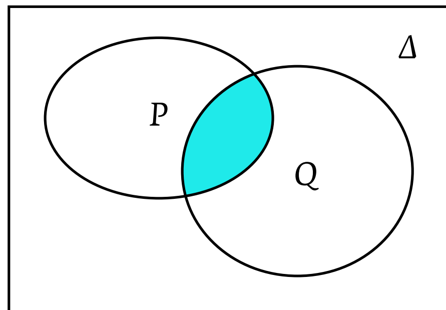
$= \top$



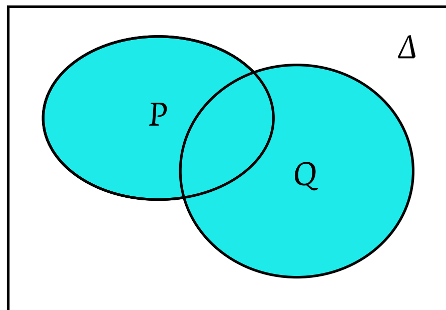
$= \bot$

Vision ensembliste de l'interprétation

- Si P et Q sont des prédicats, alors $(P \wedge Q)^{\mathcal{A}} = P^{\mathcal{A}} \cap Q^{\mathcal{A}}$
- et $(P \vee Q)^{\mathcal{A}} = P^{\mathcal{A}} \cup Q^{\mathcal{A}}$.



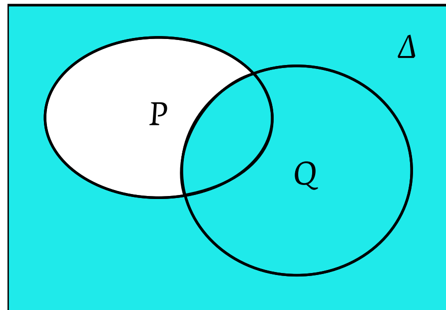
$= P \wedge Q$

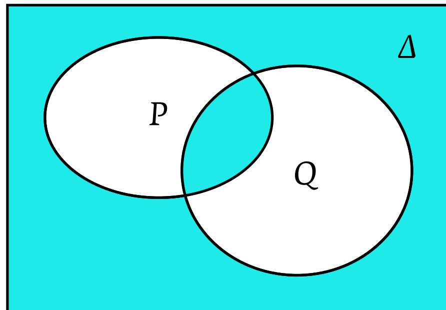


$= P \vee Q$

Vision ensembliste de l'interprétation

- De même $(P \to Q)^{\mathcal{A}}$ est $(\neg P)^{\mathcal{A}} \cup Q^{\mathcal{A}} = (\Delta \setminus P^{\mathcal{A}}) \cup Q^{\mathcal{A}}$
- et $(P \leftrightarrow Q)^{\mathcal{A}} = ((\Delta \setminus P^{\mathcal{A}}) \cup Q^{\mathcal{A}}) \cap ((\Delta \setminus Q^{\mathcal{A}}) \cup P^{\mathcal{A}}) = (\Delta \setminus (P^{\mathcal{A}} \cup Q^{\mathcal{A}})) \cup (P^{\mathcal{A}} \cap Q^{\mathcal{A}})$.



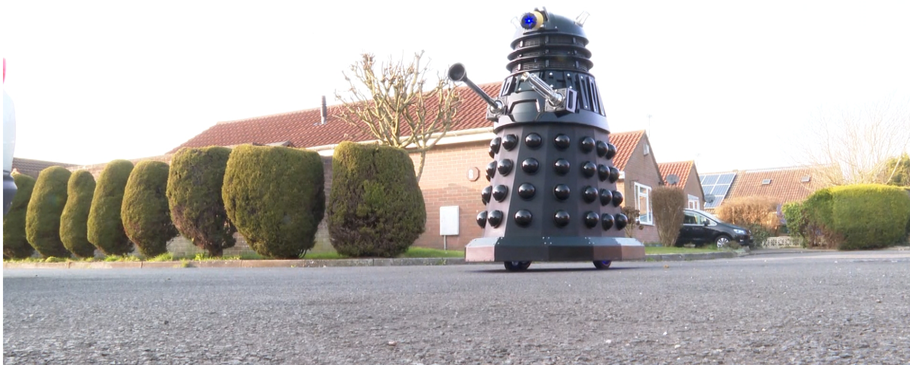


Test

- Une *tautologie* est une formule vraie quelque soit l'interprétation.
- la formule

$$\exists x, x = \text{Dalek}$$

est-elle une tautologie???



Logique propositionnelle

- On appelle *logique propositionnelle* le cas où on n'a ni prédicat (autre que les propositions), ni variable, ni fonction, ni quantificateur.
- En logique propositionnelle, on peut calculer la valeur vraie ou fausse d'un énoncé en appliquant les *tables de vérité* suivantes aux connecteurs :

A	$\neg A$
faux	vrai
vrai	faux

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	AB
faux	faux	faux	faux	vrai	vrai
faux	vrai	faux	vrai	vrai	faux
vrai	faux	faux	vrai	faux	faux
vrai	vrai	vrai	vrai	vrai	vrai

- (Ces tables nous montrent que l'on pourrait s'amuser à définir 2^4 opérateurs binaires, mais traditionnellement on se limite aux 4 ci-dessus.)

Calcul propositionnel

- Exemple de calcul : quelle est la valeur de vérité de $(P \wedge Q) \to (P \vee Q)$ selon les valeurs de vérité de P et de Q ?

P	Q	$P \wedge Q$	$P \vee Q$	$(P \wedge Q) \to (P \vee Q)$
faux	faux	faux	faux	vrai
faux	vrai	faux	vrai	vrai
vrai	faux	faux	vrai	vrai
vrai	vrai	vrai	vrai	vrai

(On dira que cette formule est une *tautologie*.)

- On voit que dans le cas bien particulier de la logique propositionnelle, une interprétation n'est rien d'autre qu'une combinaison de valeurs de vérité des propositions
- et donc une ligne de la table de vérité.

Conséquence

- Une interprétation $\mathcal{A}$ dans laquelle une formule α est vraie est appelée *modèle* de la formule. On écrit alors $\mathcal{A} \models \alpha$.
- Une formule β est une *conséquence* d'une formule α quand tout modèle de α est aussi modèle de β, autrement dit : $\forall \mathcal{A} : \mathcal{A} \models \alpha \Rightarrow \mathcal{A} \models \beta$.
- On écrit alors $\alpha \models \beta$.
- Ne pas confondre avec la déduction $\alpha \vdash \beta$ qui signifie que β peut être déduit de α par une suite d'inférences.
- (Les inférences sont appliquées au niveau formel, sans interprétation. La conséquence nécessite des interprétations.)

Interprétation

Théorème

Soient α et β deux formules. β est conséquence d'α si et seulement si $\alpha \to \beta$ est vraie dans toute interprétation.

Autrement dit : $\alpha \models \beta \iff \forall \mathcal{A} : \mathcal{A} \models (\alpha \to \beta)$

Démonstration.

On montrera la négation : $\alpha \not\models \beta \iff \exists \mathcal{A} : \mathcal{A} \not\models (\alpha \to \beta)$.

$$\begin{aligned} \alpha \not\models \beta &\overset{\text{déf}}{\Rightarrow} \exists \mathcal{A} : \mathcal{A} \models \alpha \wedge \mathcal{A} \not\models \beta && (\Rightarrow) \\ &\Rightarrow \mathcal{A} \not\models (\alpha \to \beta). \\ \exists \mathcal{A} : \mathcal{A} \not\models (\alpha \to \beta) &\Rightarrow \exists \mathcal{A} : \mathcal{A} \models \alpha \wedge \mathcal{A} \not\models \beta && (\Leftarrow) \\ &\overset{\text{déf}}{\Rightarrow} \alpha \not\models \beta. && \square \end{aligned}$$

Liens avec la langue naturelle

- Dès lors qu'elle a une valeur de vérité, une phrase déclarative peut interpréter une formule (ou inversement : une formule peut formaliser une phrase).
- Exemples : « Michel est élève de l'IMT Atlantique », « je m'ennuie », « les Bretons n'aiment pas l'écotaxe ».
- Contre-exemples : « 700 millions de Chinois, et moi, et moi, et moi », « courage, fuyons ! », « travailler plus pour gagner plus ».

Liens avec la langue naturelle

- On pourra utiliser les connecteurs $\wedge$ et $\vee$ pour formaliser les conjonctions « et » et « ou ».
- De même on pourra utiliser $\rightarrow$ pour les conclusions.
- Néanmoins, attention :
  « (Les coqs pondent des œufs) $\rightarrow$ (IMT Atlantique est une grande école) » est vraie,
  « (Les ânes volent) $\rightarrow$ (IMT Atlantique est une discothèque) » est vraie aussi.

Usage des quantificateurs

- $\forall$ est le *quantificateur universel*, on l'utilise souvent suivi d'une implication :

$$\text{tout homme est mortel}$$
$$\forall x\,(\mathrm{homme}(x) \rightarrow \mathrm{mortel}(x)).$$

- Attention à ne pas confondre avec

$$\forall x\,(\mathrm{homme}(x) \wedge \mathrm{mortel}(x)),$$

qui a une toute autre signification (laquelle ?).

Usage des quantificateurs

- $\exists$ est le *quantificateur existentiel*, on l'utilise souvent suivi d'une conjonction :

  Il existe un élève de Télécom dont le prénom est Xavier

$$\exists x\,(\text{élèveTélécom}(x) \wedge \text{prénom}(x, \text{Xavier})).$$

- À ne pas confondre avec

$$\exists x\,(\text{élèveTélécom}(x) \rightarrow \text{prénom}(x, \text{Xavier}))$$

  (pourquoi ?).

- Exercice : comment interpréter $\forall x \exists y\, \text{aime}(x, y)$ et $\exists x \forall y\, \text{aime}(x, y)$?

Qu'est-ce que l'égalité ?

- L'opérateur = dénote une relation spéciale, celle de l'*égalité* entre deux termes :

  $x = y$ signifie que x et y doivent être interprétés par la même valeur du domaine, quelque soit l'interprétation : $x^{\mathbf{I}} = y^{\mathbf{I}} = e \in \Delta$.
- De même, sa négation $\neq$ signifie que deux termes ne pourront jamais être interprétés par les mêmes valeurs du domaine.
- Pour dire, par exemple, que a a au moins deux frères, on écrira

  $$\exists x\, \exists y\, (\text{frère}(x, a) \wedge \text{frère}(y, a) \wedge x \neq y)$$

  dans le domaine des êtres humains (ou des nœuds d'un graphe, ou…).

Exemples : formaliser des proverbes

- *Chose promise, chose due.*

Exemples : formaliser des proverbes

- ***Chose promise, chose due.***
- Domaine du monde réel.

Exemples : formaliser des proverbes

- ***Chose promise, chose due.***
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».

Exemples : formaliser des proverbes

- ***Chose promise, chose due.***
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».
- $\forall X\, \text{chose}(X) \wedge \text{promise}(X) \rightarrow \text{due}(X)$.

Exemples : formaliser des proverbes

- *Chose promise, chose due.*
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».
- $\forall X\, \text{chose}(X) \wedge \text{promise}(X) \rightarrow \text{due}(X)$.
- *Il n'y a que la vérité qui blesse.*

Exemples : formaliser des proverbes

- *Chose promise, chose due.*
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».
- $\forall X \, \mathrm{chose}(X) \wedge \mathrm{promise}(X) \to \mathrm{due}(X)$.
- *Il n'y a que la vérité qui blesse.*
- Domaine du monde réel.

Exemples : formaliser des proverbes

- ***Chose promise, chose due.***
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».
- $\forall X\, \text{chose}(X) \wedge \text{promise}(X) \rightarrow \text{due}(X)$.
- ***Il n'y a que la vérité qui blesse.***
- Domaine du monde réel.
- Predicats : « vérité », « blesse ».

Exemples : formaliser des proverbes

- *Chose promise, chose due.*
- Domaine du monde réel.
- Predicats : « chose », « promise » et « due ».
- $\forall X\, \text{chose}(X) \wedge \text{promise}(X) \rightarrow \text{due}(X)$.
- *Il n'y a que la vérité qui blesse.*
- Domaine du monde réel.
- Predicats : « vérité », « blesse ».
- $\forall X\, \text{blesse}(X) \rightarrow \text{vérité}(X)$.

Exemples : formaliser des proverbes

- *Il n'est point de sot métier.*

Exemples : formaliser des proverbes

- *Il n'est point de sot métier.*
- Domaine du monde réel.

Exemples : formaliser des proverbes

- ***Il n'est point de sot métier.***
- Domaine du monde réel.
- Predicats : « métier » et « sot ».

Exemples : formaliser des proverbes

- *Il n'est point de sot métier.*
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\, \mathrm{metier}(X) \wedge \mathrm{sot}(X)))$.

Exemples : formaliser des proverbes

- ***Il n'est point de sot métier.***
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\,\mathrm{metier}(X) \land \mathrm{sot}(X)))$.
- ***Fais ce que je dis, pas ce que je fais.***

Exemples : formaliser des proverbes

- *Il n'est point de sot métier.*
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\,\mathrm{metier}(X) \wedge \mathrm{sot}(X)))$.
- *Fais ce que je dis, pas ce que je fais.*
- Domaine du monde réel.

Exemples : formaliser des proverbes

- ***Il n'est point de sot métier.***
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\, \mathrm{metier}(X) \wedge \mathrm{sot}(X)))$.
- ***Fais ce que je dis, pas ce que je fais.***
- Domaine du monde réel.
- Predicats : « faire ». Constantes : « moi », « toi ».

Exemples : formaliser des proverbes

- *Il n'est point de sot métier.*
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\, \mathrm{metier}(X) \wedge \mathrm{sot}(X)))$.
- *Fais ce que je dis, pas ce que je fais.*
- Domaine du monde réel.
- Predicats : « faire ». Constantes : « moi », « toi ».
- $\forall X\, \mathrm{dire}(moi, X) \to \mathrm{faire}(toi, X) \wedge (\mathrm{faire}(moi, X) \to \neg\mathrm{faire}(toi, X))$.

Exemples : formaliser des proverbes

- ***Il n'est point de sot métier.***
- Domaine du monde réel.
- Predicats : « métier » et « sot ».
- $\neg\exists X\,\mathrm{metier}(X) \wedge \mathrm{sot}(X)))$.
- ***Fais ce que je dis, pas ce que je fais.***
- Domaine du monde réel.
- Predicats : « faire ». Constantes : « moi », « toi ».

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.
- En logique du 1er ordre on ne peut formaliser que : ***Je ne sais qu'une chose et je ne sais rien.***

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.
- En logique du 1er ordre on ne peut formaliser que : ***Je ne sais qu'une chose et je ne sais rien.***
- Predicats : « savoir », « chose ». Constantes : « moi ».

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (῝Εν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.
- En logique du 1er ordre on ne peut formaliser que : ***Je ne sais qu'une chose et je ne sais rien.***
- Predicats : « savoir », « chose ». Constantes : « moi ».
- $\exists X\, \mathrm{savoir}(moi, X) \wedge (\forall Y\, \mathrm{savoir}(moi, Y) \rightarrow (X = Y)) \wedge \neg\exists X\, \mathrm{savoir}(moi, X)$.

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.
- En logique du 1er ordre on ne peut formaliser que : ***Je ne sais qu'une chose et je ne sais rien.***
- Predicats : « savoir », « chose ». Constantes : « moi ».
- $\exists X\, \text{savoir}(moi, X) \wedge (\forall Y\, \text{savoir}(moi, Y) \rightarrow (X = Y)) \wedge \neg\exists X\, \text{savoir}(moi, X)$.
- Qui en réalité est une contradiction.

Exemples : formaliser des proverbes

- ***Je ne sais qu'une chose, c'est que je ne sais rien.*** (Ἓν οἶδα, ὅτι οὐδὲν οἶδα.)
- Domaine du monde réel.
- En logique du 1er ordre on ne peut formaliser que : ***Je ne sais qu'une chose et je ne sais rien.***
- Predicats : « savoir », « chose ». Constantes : « moi ».
- $\exists X\, \mathrm{savoir}(moi, X) \wedge (\forall Y\, \mathrm{savoir}(moi, Y) \rightarrow (X = Y)) \wedge \neg\exists X\, \mathrm{savoir}(moi, X)$.
- Qui en réalité est une contradiction.

Équivalences logiques

- On dit qu'on a *équivalence logique* entre deux énoncés s'ils possèdent exactement les mêmes modèles.
- Voici une liste d'équivalences qui nous seront bien utiles pour faire des calculs :

$\alpha \wedge \beta$	$\equiv$	$\beta \wedge \alpha$	*commutativité*
$\alpha \vee \beta$	$\equiv$	$\beta \vee \alpha$	commutativité
$\alpha \wedge (\beta \wedge \gamma)$	$\equiv$	$(\alpha \wedge \beta) \wedge \gamma$	*associativité*
$\alpha \vee (\beta \vee \gamma)$	$\equiv$	$(\alpha \vee \beta) \vee \gamma$	associativité
$\neg\neg\alpha$	$\equiv$	α	*double négation*
$\neg(\alpha \wedge \beta)$	$\equiv$	$\neg\alpha \vee \neg\beta$	*De Morgan*
$\neg(\alpha \vee \beta)$	$\equiv$	$\neg\alpha \wedge \neg\beta$	De Morgan
$\alpha \wedge (\beta \vee \gamma)$	$\equiv$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$	*distributivité*
$\alpha \vee (\beta \wedge \gamma)$	$\equiv$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$	distributivité
$\forall x \neg P$	$\equiv$	$\neg\exists x P$	
$\neg\forall x P$	$\equiv$	$\exists x \neg P$	
$\forall x P$	$\equiv$	$\neg\exists x \neg P$	
$\exists x P$	$\equiv$	$\neg\forall x \neg P$	

Satisfaisabilité, Mick Jagger, théories

- Un énoncé est une *tautologie* si toutes ses interprétations sont des modèles. Exemple : $A \vee \neg A$. On note $\models A$.
- Un énoncé est *satisfaisable* s'il possède au moins un modèle.
- Un énoncé est *insatisfaisable* (comme) s'il ne possède aucun modèle. Exemple : $\alpha \wedge \neg\alpha$. On dit aussi que c'est une *contradiction* et on note $\not\models A$.
- Un ensemble d'énoncés fermé pour la relation de conséquence est appelé une *théorie*,
- les énoncés d'une théorie sont des *théorèmes*.

QC : Question cruciale

- Une *base de connaissances* est une conjonction de formules logiques possédant un ou plusieurs modèles (si possible : intéressants).
- On se pose la question cruciale :

Question cruciale

Étant donnés une *base de connaissances* KB et un énoncé α, est-ce qu'on a KB $\models \alpha$?

Autrement dit : est-ce que tout modèle de KB est un modèle de α ? ou encore : est-ce que α est vrai dans tous les mondes possibles dans lesquels KB est vraie ? (et en particulier, le nôtre, si KB est une base de connaissances du monde réel).

Deux théorèmes fondamentaux

Théorème (Complétude)

Pour toute formule ϕ et tout ensemble de formules Γ de logique du 1$^{\text{er}}$ ordre, $\Gamma \models \phi$ ssi $\Gamma \vdash \phi$.

Théorème (Compacité)

Soit Γ un ensemble de formules de logique du 1$^{\text{er}}$ ordre, alors Γ est satisfaisable ssi tout sous-ensemble fini de Γ est satisfaisable.

- Autrement dit, pour montrer que $\mathrm{KB} \models \alpha$, qui est la QC, il suffit de déduire α de KB par une suite d'inférences.
- Reste à trouver un système déductif implémentable/implémenté, pour que la machine fasse le travail pour nous.

L'importance de l'insatisfaisabilité

Théorème

KB$\models \alpha$ ssi KB$\wedge\neg\alpha$ est insatisfaisable.

Démonstration.

Simplissime, laissée au·à la lecteur·rice. □

- Autrement dit, pour montrer que KB $\models \alpha$, qui est la QC, il suffit de montrer par des inférences que KB$\wedge\neg\alpha$ est une contradiction.
- Reste à trouver un système déductif implémentable/implémenté, pour que la machine fasse le travail pour nous.

Forme normale prénexe

- On dit que l'énoncé φ est en *forme normale prénexe* (PNF) s'il est de la forme
$$\mathrm{Q}_1 x_1 \cdots \mathrm{Q}_n x_n \psi$$
où Q_i sont des quantificateurs et ψ est un énoncé sans quantificateurs.
- Exemples : $\forall x \exists y\, (f(x) = y)$ est en PNF. $\neg\forall x \exists y\, P(x, y, z)$ ne l'est pas (pourquoi ?), ni $\exists x \forall y\, \neg P(x, y, z) \wedge \forall x \exists y\, Q(x, y, z)$.

PNF

Théorème

Pour tout énoncé de logique du 1er ordre, il existe un énoncé en PNF qui lui est équivalent.

L'algorithme pour l'obtenir consiste à renommer les variables liées et à déplacer les quantificateurs de variables de manière à ce qu'ils englobent des parties de l'énoncé dans lesquelles ces variables n'apparaissent pas.

- Exemple d'application de l'algorithme PNF :

$$\begin{aligned}\varphi &\equiv \exists x \forall y\, \neg P(x,y,z) \wedge \forall x \exists y\, Q(x,y,z)\\ &\equiv \exists x \forall y\, \neg P(x,y,z) \wedge \forall u \exists v\, Q(u,v,z)\\ &\equiv \exists x \forall y \forall u \exists v\, (\neg P(x,y,z) \wedge Q(u,v,z)).\end{aligned}$$

CNF

- On dit qu'un énoncé est en CNF (*forme normale conjonctive*) s'il s'écrit sous forme d'une conjonction de disjonctions ($(F_1 \vee F_2) \wedge (G_1 \vee \neg G_2)$, etc.).

Proposition

Tout énoncé (sans quantificateurs) est équivalent à un énoncé en CNF.

Exemple

Voici l'algorithme pour l'obtenir :

1. remplacer les $\leftrightarrow$ par des conjonctions de $\rightarrow$;
2. remplacer les $\alpha \rightarrow \beta$ par des $\neg\alpha \vee \beta$;
3. faire entrer les négations dans les parenthèses pour obtenir des littéraux (autrement dit : appliquer les lois de *De Morgan*) ;
4. utiliser la propriété distributive.

Exemple :

Énoncé de départ : $B \leftrightarrow (F \vee G)$.
Élimination du $\leftrightarrow$: $B \rightarrow (F \vee G) \wedge (F \vee G) \rightarrow B$.
Élimination du $\rightarrow$: $(\neg B \vee F \vee G) \wedge (\neg(F \vee G) \vee B)$.
Faire entrer les $\neg$: $(\neg B \vee F \vee G) \wedge ((\neg F \wedge \neg G) \vee B)$.
Appliquer la *distributivité* : $(\neg B \vee F \vee G) \wedge (\neg F \vee B) \wedge (\neg G \vee B)$.

CPNF

- Quand un énoncé est sous forme prénexe et sa partie sans quantificateurs sous CNF, on dira qu'il est sous *CPNF* (*forme normale prénexe conjonctive*).

Théorème

Pour tout énoncé de logique du 1$^{\text{er}}$ ordre, il existe un énoncé en CPNF qui lui est équivalent.

Skolémisation

Definition

Un énoncé est sous *forme normale de Skolem* (*SNF*) s'il est en CPNF avec uniquement des quantificateurs universels.

Théorème

Soit φ un énoncé et φ^S son skolémisé. Alors φ est satisfaisable ssi φ^S est satisfaisable.

Skolémisation

- Il existe un algorithme qui fournit pour tout énoncé φ, un énoncé φ_S sous SNF que l'on appelle sa *skolémisation* :
  1. à partir de φ, obtenir $\varphi' = Q_1 \cdots Q_m\, \varphi(x_1, \ldots, x_m)$ en CPNF ;
  2. si tous les Q_i sont $\forall$, φ' est en SNF ;
  3. sinon, à partir de φ' obtenir $s(\varphi')$ qui a un $\exists$ en moins que φ' :
     – si le $\exists$ est en première position ($\exists x_1$), on supprime $\exists x_1$ et on remplace x_1 dans l'énoncé par une constante c qui n'apparaît pas dans φ' ;
     – si le $\exists$ est en i-ème position, on supprime $\exists x_i$ et on remplace x_i par une fonction $(i-1)$-aire $f(x_1, \ldots, x_{i-1})$ qui n'apparaît pas dans φ' ;
  4. en répétant cette étape k fois (pour k quant. exist.) on obtient $\overbrace{s \circ \cdots \circ s}^{k \text{ fois}}(\varphi')$, qui est en SNF.

Exemple de skolémisation

- En français : « Tous ceux qui aiment tous les animaux sont aimés par qqun.
- En logique : $\forall x\,((\forall y\,\mathrm{Animal}(y) \to \mathrm{Aime}(x, y)) \to (\exists y\,\mathrm{Aime}(y, x)))$;
- on élimine la 2e implication :
  $\forall x\,(\neg((\forall y\,\mathrm{Animal}(y) \to \mathrm{Aime}(x, y))) \vee (\exists y\,\mathrm{Aime}(y, x)))$;
- on élimine la 1re implication :
  $\forall x\,(\neg(\forall y\,(\neg(\mathrm{Animal}(y)) \vee \mathrm{Aime}(x, y))) \vee (\exists y\,\mathrm{Aime}(y, x)))$;
- on gère les négations :
  $\forall x\,(\exists y\,\neg(\neg\mathrm{Animal}(y) \vee \mathrm{Aime}(x, y))) \vee (\exists y\,\mathrm{Aime}(y, x)))$;
- et encore : $\forall x\,(\exists y\,(\mathrm{Animal}(y) \wedge \neg\mathrm{Aime}(x, y))) \vee (\exists y\,\mathrm{Aime}(y, x)))$;

Exemple de skolémisation

- On est resté là : $\forall x\,(\exists y\,(\mathrm{Animal}(y) \wedge \neg\mathrm{Aime}(x,y))) \vee (\exists y\,\mathrm{Aime}(y,x)))$;
- on skolemise : $\forall x\,(\mathrm{Animal}(f(x)) \wedge \neg\mathrm{Aime}(x,f(x))) \vee \mathrm{Aime}(g(x),x)$ où f et g sont des fonctions de Skolem ;
- on distribue la disjonction :
  $\forall x\,(\mathrm{Animal}(f(x)) \vee \mathrm{Aime}(g(x),x)) \wedge (\neg\mathrm{Aime}(x,f(x)) \vee \mathrm{Aime}(g(x),x))$.

Cet énoncé est plus opaque que celui du début, mais aussi plus maniable pour la machine. (Ici $f(x)$ est un « animal potentiellement non-aimé par x » et $g(x)$ est « quelqu'un qui peut aimer x ».)

Conséquences

Lesquelles parmi les affirmations suivantes sont correctes ?

1. faux $\models$ vrai
2. vrai $\models$ faux
3. $(A \wedge B) \models (A \leftrightarrow B)$
4. $A \leftrightarrow B \models A \vee B$
5. $A \leftrightarrow B \models \neg A \vee B$
6. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B \vee C) \wedge (B \wedge C \wedge D \rightarrow E)$
7. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$
8. $(A \vee B) \wedge \neg(A \rightarrow B)$ est satisfaisable
9. $(A \wedge B) \rightarrow C \models (A \rightarrow C) \vee (B \rightarrow C)$
10. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \rightarrow C) \wedge (B \rightarrow C))$

⑪ $(A \leftrightarrow B) \land (\neg A \lor B)$ est satisfaisable

---

SOLUTION 1. vrai puisque faux n'a aucun modèle
2. faux puisque vrai a tous les modèles, faux n'en a aucun
3. vrai, la PG (= partie de gauche) a un seul modèle qui satisfait la PD
4. faux car si A et B sont faux la PG est vraie mais pas la PD
5. vrai $\neg A \lor B$ n'est autre que $A \to B$ qui est trivialement impliqué
6. vrai on réécrit la PD en CNF et on voit que les clauses de la PG sont des sous-clauses de celles de PD
7. faux, car situation inverse que dans 6 : clause plus petite, moins de modèles
8. vrai, on écrit du CNF et on obtient qqch $\land A \land \neg B$ donc voilà notre modèle : A vrai, B faux
9. vrai, pour que PD soit faux il faut que A et B soient vrais et C faux, et dans ce cas PG est également faux, dans tous les autres cas PG est vrai et PD est vrai. C'est contre-intuitif si on interprète $\to$ comme « cause »
10. vrai, faire la table de vérité

TP de logique III

11. la deuxième partie ($A \to B$) est un cas particulier de la première donc les modèles de la première sont aussi des modèles de la deuxième (cf. question 5).

Tautologies ? Contradictions ?

Décider lesquels parmi les énoncés suivants sont des tautologies, des contradictions ou ni l'un ni l'autre.

1. fumée $\to$ fumée
2. fumée $\to$ feu
3. (fumée $\to$ feu) $\to$ ($\neg$fumée $\to$ $\neg$feu)
4. fumée $\vee$ feu $\vee$ $\neg$feu
5. ((fumée $\wedge$ chaleur) $\to$ feu) $\leftrightarrow$ ((fumée $\to$ feu) $\vee$ (chaleur $\to$ feu))
6. grand $\vee$ stupide $\vee$ (grand $\to$ stupide)
7. (grand $\wedge$ stupide) $\vee$ $\neg$stupide

TP de logique IV

SOLUTION 1. tautologie
2. ni l'un ni l'autre
3. ni l'un ni l'autre
4. tautologie
5. tautologie
6. tautologie
7. ni l'un ni l'autre

Les métiers

Soit le vocabulaire suivant :

- $O(p, o)$ prédicat, la personne p occupe le poste o
- $C(p_1, p_2)$ prédicat, la personne p_1 est client de la personne p_2
- $B(p_1, p_2)$ prédicat, la personne p_1 est un supérieur hiérarchique de la personne p_2

TP de logique V

- D, Ch, Av, Ac constantes, des métiers : docteur, chirurgien, avocat, acteur
- E, J constantes, des personnes : Émilie, Joël.

Utiliser ces symboles pour écrire les énoncés suivants en logique du 1er ordre :

1. Émilie est soit chirurgienne, soit avocate
2. Joël est acteur, mais il aussi un autre job
3. Tous les chirurgiens sont docteurs
4. Émilie a un boss qui est avocat
5. Il existe un avocat dont tous les clients sont des docteurs
6. Tout chirurgien a un avocat

TP de logique VI

SOLUTION 1. $O(E, Ch) \vee O(E, Av)$
2. $O(J, Ac) \wedge \exists p, (p \neq Ac \wedge O(J, p))$
3. $\forall p, O(p, Ch) \rightarrow O(p, D)$
4. $\neg \exists p, C(J, p) \wedge O(p, Av)$
5. $\exists p, B(p, E) \wedge O(p, Av)$
6. $\exists p, O(p, Av) \wedge \forall q (C(q, p) \rightarrow O(q, D))$
7. $\forall p (O(p, Ch) \rightarrow \exists q (O(q, Av) \wedge C(p, q)))$.

Conjecture de Goldbach

Soient les nombres naturels $\mathbb{N}$ avec le prédicat $<$, les fonctions $+$ et $\times$, et les constantes 0 et 1. Écrire en logique du 1$^{\text{er}}$ ordre :

1. la propriété de parité ($\text{Pair}(x)$)
2. la propriété d'être nombre premier ($\text{Premier}(x)$)
3. la conjecture de Goldbach[1].

TP de logique VII

SOLUTION 1. $\forall x, \mathrm{Pair}(x) \leftrightarrow \exists y, x = y + y$
2. $\forall x, \mathrm{Premier}(x) \leftrightarrow \forall y, z\ x = y \times z \rightarrow (y = 1 \vee z = 1)$
3. $\forall x, \mathrm{Pair}(x) \leftrightarrow \exists y, z\ \mathrm{Premier}(x) \wedge \mathrm{Premier}(y) \wedge x = y + z$ (il ne reste plus qu'à la prouver…)

L'ADN

En utilisant les prédicats 1-aires ADN, Personne, Père, Mère et le prédicat 3-aire DérivéDe, écrire en logique du 1er ordre la phrase : l'ADN d'une personne est unique et dérivé de ceux de ses parents.

SOLUTION A=ADN, P=Personne, D=DérivéDe, M=mère, Pe=père

$$(\forall x, y(P(x) \wedge P(y) \wedge x \neq y) \rightarrow A(x) \neq A(y)) \wedge (P(x) \rightarrow D(A(x), A(M(x)), A(Pe(x)))).$$

TP de logique VIII

Exercice casse-tête, pour approfondir les notions

Vrai ou faux ? Expliquer

1. Tout énoncé existentiellement quantifié est vrai dans tout modèle contenant exactement un objet.
2. « $\forall x, y\ x = y$ » est satisfaisable.

SOLUTION 1. Faux. Il suffit de prendre $\exists x, y\ x \neq y$, cet énoncé n'est satisfaisable que dans un modèle basé sur un domaine possédant au moins deux objets distincts.
2. Vrai. Il suffit de prendre un modèle à un seul objet.

---

[1] Tout nombre pair est somme de deux nombres premiers : $2 = 1 + 1$, $4 = 2 + 2$, $6 = 1 + 5$, $8 = 7 + 1$, $10 = 7 + 3$, $12 = 7 + 5$, $14 = 7 + 7$, ...

Voici la suite du TP « Logique et langage naturel ».

(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 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

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

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

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

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

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

Donc on peut en conclure que

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

Preuve :

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

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 $\langle\langle e, e\rangle, t\rangle$!

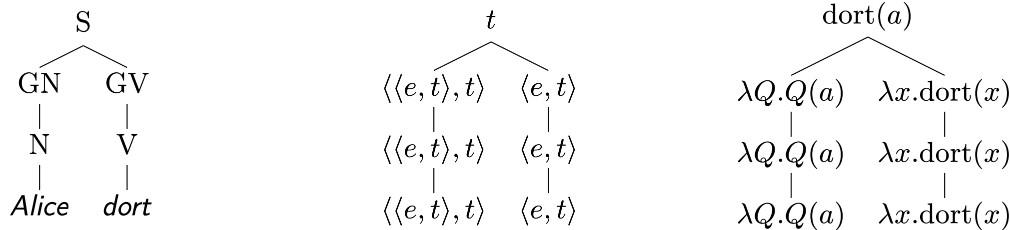


Figure 1: Arbre syntaxique de la phrase *Alice dort*, arbre de ses types sémantiques, arbre de la traduction en formalisme logique.

- 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 $\lambda Q.Q(\text{alice})(\lambda x.\text{dort}(x))$, 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.

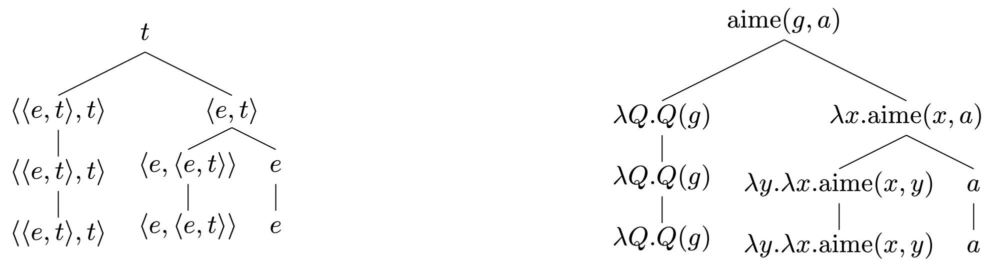
1.5.1 Exercice sur l'amour

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

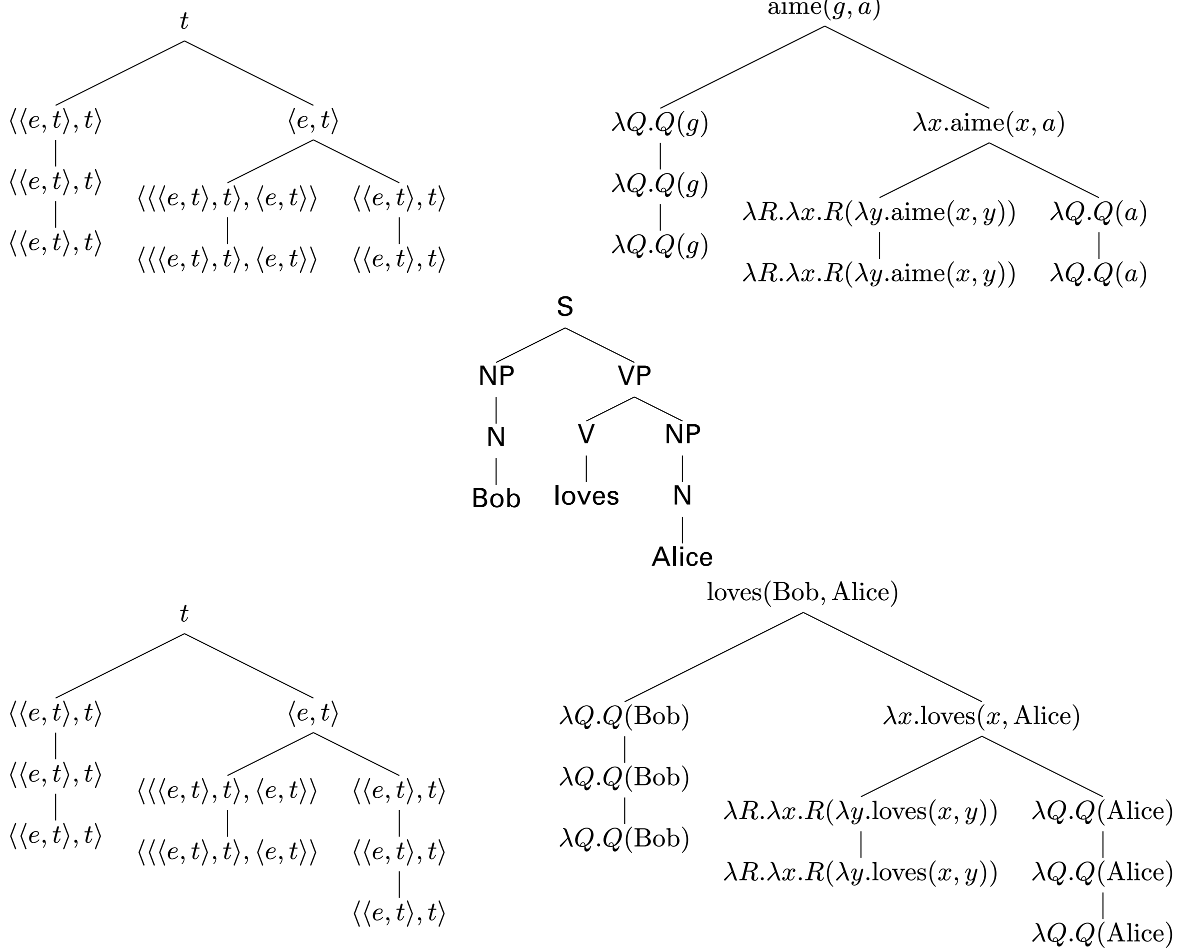



```
from nltk.sem.logic import *
lexpr = Expression.fromstring
print(lexpr(r'\y (\x.aime(x,y)) (alice)').simplify())
print(lexpr(r'\Q.Q(gerald) (\y (\x.aime(x,y)) (alice))').simplify())
```

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)
```

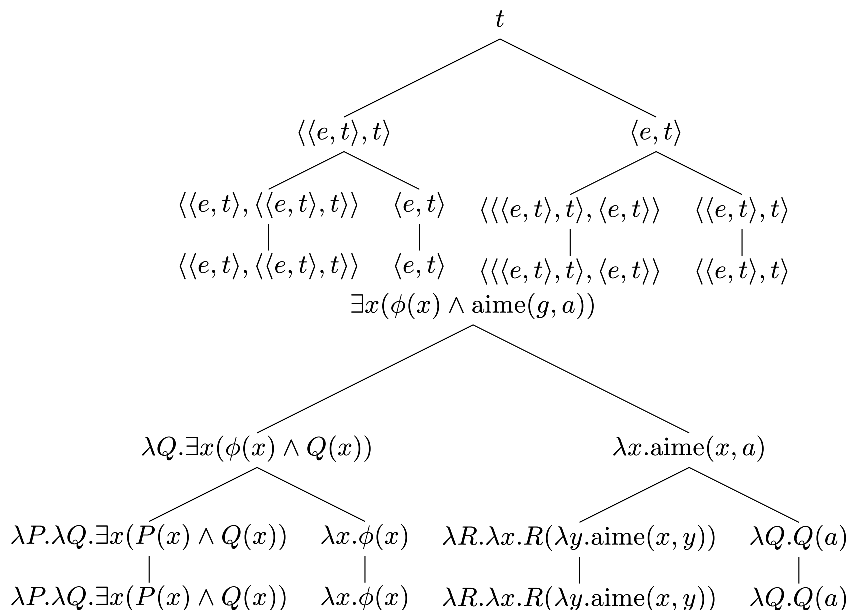
Figure 2: Arbres syntaxiques des phrases *Gérard aime Alice* et *un philosophe aime Alice*.

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 $\text{philosophe}(g)$ 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* ?

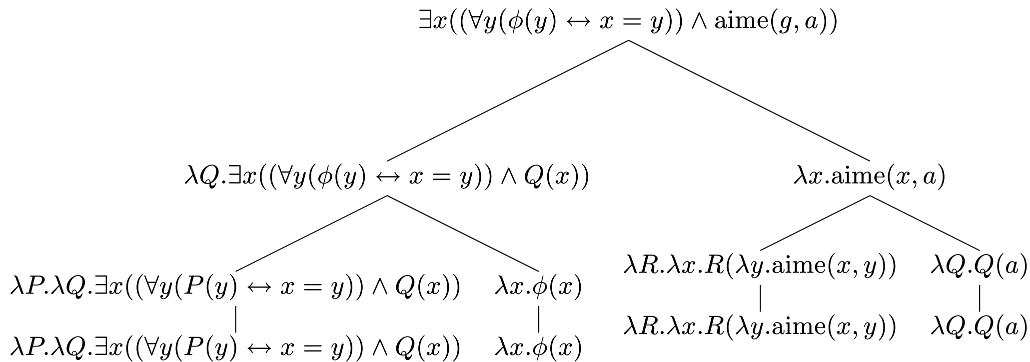
Formulons la question autrement : comment indiquer qu'il n'y a qu'*un seul* philosophe, et que quand on dit *le philosophe* on parle justement de lui ?

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...

Section 4

Logique typée

Logique typée

- On munit la syntaxe de la logique du 1er ordre d'un *ensemble de types*.
- Chaque constante et variable appartient à un type.
- Chaque fonction et chaque prédicat a une signature (types d'entrée, type de sortie).
- D'ailleurs il n'y a plus de distinction entre fonction et prédicat : le dernier est une fonction dont le type de sortie est booléen.
- Les connecteurs $\neg, \rightarrow, \leftrightarrow, \wedge, \vee$ ont la signature (booléen, booléen) ou (booléen,booléen,booléen).

Z3 et Z3py

- Z3 est un démonstrateur automatique de théorèmes en logique typée, développé par Micro$oft (mais diffusé gratuitement).
- Nous allons utiliser son interface Python.
- Exemple : Si Humain(Socrate) et $\forall X\,\mathrm{Humain}(X) \rightarrow \mathrm{Mortel}(X)$ forment notre KB, peut-on conclure que Mortel(Socrate) ?
- On va appliquer le résultat qui dit qu'il suffit de montrer que $\mathrm{KB} \wedge \neg \mathrm{Mortel}(\mathrm{Socrate})$ est insatisfaisable.

Z3 et Z3py

1. On va déclarer un type appelé `Humain` et on dira que `Socrate` et la variable `x` sont de ce type.
2. On va déclarer des fonctions `is_Humain` et `is_Mortel` de signature (`Humain`, booléen).
3. On va écrire les trois formules :
   - Humain(Socrate) ;
   - $\forall X\,\text{Humain}(X) \to \text{Mortel}(X)$;
   - Mortel(Socrate),

   dans la syntaxe de Z3py :
   - `is_Humain(Socrate)` ;
   - `ForAll(x,Implies(is_Humain(x),is_Mortel(x)))` ;
   - `Not(is_Mortel(Socrate))`.
4. Et on demandera à Z3py si la formule globale ainsi obtenue est satisfaisable : il nous répondra « `unsat` ».

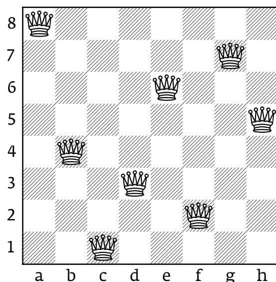
Le *modus ponens* Z3pysé

```
from z3 import *

Humain = DeclareSort('Humain')
Socrate = Const('Socrate', Humain)
is_Mortel = Function('is_Mortel',Humain,BoolSort())
is_Humain = Function('is_Humain',Humain,BoolSort())

s = Solver()

x = Const('x', Humain)
s.add(ForAll(x,Implies(is_Humain(x),is_Mortel(x))))
s.add(is_Humain(Socrate))
s.add(Not(is_Mortel(Socrate)))

print(s.check())
```

Plus de Z3py en TP

En TP on utilisera Z3py plus abondamment, pour :

- montrer que c'est Curiosité qui a tué le chat ;
- résoudre le problème des huit reines :



- montrer que les dragons rusés ne doivent pas craindre les touristes 狡猾的龙会让游客们感到害怕吗? ；
- trouver des carrés magiques :

9	3	14	8
4	16	1	13
6	1	7	11
15	5	12	2

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

BY NC ND

UE TLFT

TP Z3
(mercredi 26 février 2020)

Yannis Haralambous (IMT Atlantique)

Le but de ce TP est la découverte du vérificateur de théorème Z3 et de son API Python.

Installer

```
python -m pip install z3 --user
```

Sur MacOS X, préférer :

```
brew install z3
```

1 Introduction

Voici le programme vu en cours :

```
from z3 import *

Humain = DeclareSort('Humain')
Socrate = Const('Socrate', Humain)
is_Mortel = Function('is_Mortel',Humain,BoolSort())
is_Humain = Function('is_Humain',Humain,BoolSort())

s = Solver()

x = Const('x', Humain)
s.add(ForAll(x,Implies(is_Humain(x),is_Mortel(x))))
s.add(is_Humain(Socrate))
s.add(Not(is_Mortel(Socrate)))

print(s.check())
```

On déclare un type «Humain» et des variable de ce type appelées «Socrate» et «x». On déclare des fonctions «Is_Mortel» et «Is_Humain» avec les bons types.

Ensuite on crée un solveur.

À chaque fois qu'on exécutera un `s.add`, on ajoutera une ligne à la KB (donc une conjonction à la formule globale). Les ingrédients de base de la lagique du 1er ordre s'écrivent `Or`, `And`, `Implies`, `Not`, `ForAll`, `Exists`, `==` (pour $\leftrightarrow$).

La méthode `check` va nous dire si la formule est satisfaisable ou non.

Si on change le code :

```
from z3 import *

set_param(proof=True)
Humain = DeclareSort('Humain')
Socrate = Const('Socrate', Humain)
is_Mortel = Function('is_Mortel',Humain,BoolSort())
is_Humain = Function('is_Humain',Humain,BoolSort())

s = Solver()

x, y, z = Consts('x y z', Humain)
s.add(ForAll(x,Implies(is_Humain(x),is_Mortel(x))))
s.add(is_Humain(Socrate))
s.add(is_Mortel(Socrate))

print(s.check())

print(s.model())
```

on va obtenir une formule satisfaisable et si on demande un modèle on aura :

```
[Socrate = Humain!val!0,
 is_Humain = [else -> True],
 is_Mortel = [else -> True]]
```

c'est-à-dire : «Socrate» est un élément (indexé par `0`) de notre domaine, et les deux prédicats sont toujours vrais.

2 Qui a tué le chat ?

Formaliser les phrases suivantes :

Tous ceux qui aiment tous les animaux sont aimés par qqun
Quiconque tue un animal n'est aimé par personne
Jack aime tous les animaux
C'est soit Jack soit Curiosité qui a tué le chat appelé Luna

Montrer en utilisant Z3 que c'est Curiosité qui a tué le chat. Pour définir le type on utilisera `EnumSort` qui permet de définir un type et de donner tous les éléments du domaine de ce type, en même temps :

```
S, (jack,luna,curiosity) = EnumSort('S', ('jack','luna','curiosity'))
```

SOLUTION

```
from z3 import *

S, (jack,luna,curiosity) = EnumSort('S', ('jack','luna','curiosity'))
animal = Function('animal',S,BoolSort())
chat = Function('chat',S,BoolSort())
aime = Function('aime',S,S,BoolSort())
tue = Function('tue',S,S,BoolSort())

s=Solver()

x,y,z=Consts('x y z',S)
s.add(ForAll(x,Implies(ForAll(y,Implies(animal(y),aime(x,y))),Exists(y,aime(y,x)))))
s.add(ForAll(x,Implies(Exists(z,And(animal(z),tue(x,z))),ForAll(y,Not(aime(y,x))))))
```

```
s.add(ForAll(x,Implies(animal(x),aime(jack,x))))
s.add(Or(tue(jack,luna),tue(curiosity,luna)))
s.add(chat(luna))
s.add(ForAll(x,Implies(chat(x),animal(x))))

print(s.check())
print(s.model())
```

Demander `s.model()` et interpréter le modèle.

SOLUTION

```
[animal = [else -> Or(Var(0) == jack, Var(0) == luna)],
 chat = [else -> Var(0) == luna],
 tue = [else -> And(Var(0) == curiosity, Var(1) == luna)],
 aime = [else ->
         Or(And(Var(0) == jack, Var(1) == jack),
            And(Var(0) == jack, Var(1) == luna))]]
```

3 Les dragons chinois

Formaliser les phrases suivantes :

所有强龙都能喷火。
Aucun dragon fort ne peut ne pas souffler le feu.
一只狡猾的龙总是有角的。
Un dragon rusé a toujours des cornes.
弱龙是没有角的。
Aucun dragon faible n'a des cornes.
游客们只狩猎那些不喷火的龙。
Les touristes ne chassent que les dragons ne soufflant pas le feu.

On veut répondre à la question suivante :

狡猾的龙会让游客们感到害怕吗？ 即：它会被捉吗？
Un dragon rusé doit-il craindre les touristes ? (autrement dit : est-il chassé ?)

SOLUTION

```
from z3 import *

Dragon, (drago,) = EnumSort('Dragon', ('drago',))
fort = Function('fort',Dragon,BoolSort())
feu = Function('feu',Dragon,BoolSort())
ruse = Function('ruse',Dragon,BoolSort())
cornes = Function('cornes',Dragon,BoolSort())
chasse = Function('chasse',Dragon,BoolSort())
x=Const('x',Dragon)

s=Solver()

s.add(Not(Exists(x,And(fort(x),Not(feu(x))))))
s.add(ForAll(x,Implies(ruse(x),cornes(x))))
s.add(Not(Exists(x,And(Not(fort(x)),cornes(x)))))
s.add(ForAll(x,Implies(chasse(x),Not(feu(x)))))

s.add(Exists(x,And(ruse(x),chasse(x))))

print(s.check())
```

4 Généalogie

Soit la base de connaissances KB suivante :

$$\begin{cases} \text{pa}(\text{Rob}, \text{Kev}) \\ \text{pa}(\text{Rob}, \text{Sama}) \\ \text{pa}(\text{Sama}, \text{Tho}) \\ \text{pa}(\text{Dor}, \text{Jim}) \\ \text{pa}(\text{Bor}, \text{Jim}) \\ \text{pa}(\text{Bor}, \text{Eli}) \\ \text{pa}(\text{Jim}, \text{Tho}) \\ \text{pa}(\text{Sama}, \text{Samu}) \\ \text{pa}(\text{Jim}, \text{Samu}) \\ \text{pa}(\text{Zel}, \text{Max}) \\ \text{pa}(\text{Samu}, \text{Max}) \\ \forall X, Y, Z \; \text{pa}(X, Z) \wedge \text{pa}(Z, Y) \rightarrow \text{gp}(X, Y) \end{cases}$$

où $\text{pa}(X, Y)$ signifie que X est parent de Y, et $\text{gp}(X, Y)$ que X est grand-parent de Y.

Montrer que

$$\exists X \; \text{gp}(\text{Rob}, X) \wedge \text{pa}(X, \text{Max}).$$

et trouver le X en question.

Attention, il y a un piège !

Si on déclare que certains prédicats sont vrais, cela ne veut pas dire que les autres sont faux. On demande la satisfaisabilité, donc le solveur va tout essayer, y compris les valeurs «vrai» pour les prédicats pour lesquels on n'a pas donné de valeur. Si vous y prenez mal il vous donnera comme réponse «Rob», en prétendant que Rob est parent de lui-même et aussi parent de Max (on n'a jamais dit le contraire).

SOLUTION

```
from z3 import *
Human, (Rob, Kev, Sama, Tho, Dor, Jim, Bor, Eli, Samu, Zel, Max) \
  = EnumSort('Human', ('Rob', 'Kev', 'Sama', 'Tho', 'Dor', 'Jim', 'Bor', 'Eli', 'Samu', 'Zel', 'Max'))
parent      = Function('parent',      Human, Human, BoolSort())
grandParent = Function('grandParent', Human, Human, BoolSort())

s = Solver()
x, y, z = Consts('x y z', Human)
s.add(ForAll([x, y, z], Implies(And(parent(x, z), parent(z, y)), grandParent(x, y))))
parents = [ (Rob, Kev), (Rob, Sama), (Sama, Tho),  (Dor, Jim)  \
          , (Bor, Jim), (Bor, Eli), (Jim, Tho),  (Sama, Samu)  \
          , (Jim, Samu), (Zel, Max), (Samu, Max)               \
          ]
s.add(ForAll([x, y], Implies(parent(x, y), Or([And(x==i, y == j) for (i, j) in parents]))))
witness = Const('witness', Human)
s.add(grandParent(Rob, witness))
s.add(parent(witness, Max))

print(s.check())
print(s.model()[witness])
```

---

5 Les huit reines

Il s'agit de mettre huit reines sur un échiquier sans qu'elles se menacent mutuellement, c'est-à-dire de manière qu'elles soient ni sur une même lige verticale ou horizontale, ni sur une même diagonale.

On va utiliser un autre type, le type entier `Int`.

On va d'abord créer une liste Python de huit variables qui correspondent aux huit lignes.

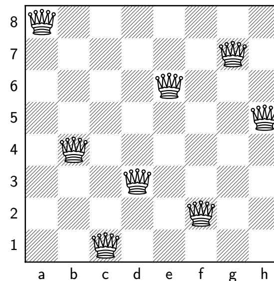
Les valeurs de ces variables correspondent aux colonnes.

Première contrainte : les colonnes sont de valeurs entre 1 et 8.

Deuxième contrainte : elles sont toutes distinctes (pour cela Z3py nous fournit la méthode `Distinct` que l'on peut appliquer à une liste de variables et qui retourne une formule).

Troisième contrainte : elles ne doivent pas être sur la même diagonale.

On met les trois contraintes ensemble et on résout.



SOLUTION

```
from z3 import *

Q = [ Int('Q_%i' % (i + 1)) for i in range(8) ]

# Each queen is in a column {1, ... 8 }
val_c = [ And(1 <= Q[i], Q[i] <= 8) for i in range(8) ]

# At most one queen per column
col_c = [ Distinct(Q) ]

# Diagonal constraint
diag_c = [ If(i == j,
              True,
              And(Q[i] - Q[j] != i - j, Q[i] - Q[j] != j - i))
           for i in range(8) for j in range(i) ]

s=Solver()
s.add(val_c + col_c + diag_c)
print(s.check())
print(s.model())
```

6 Les carrés magiques

Un carré magique est un carré de nombres distincts entre 1 et la taille du carré, tel que les sommes des nombres de toute ligne, de toute colonne, et des deux diagonales soient toutes égales au même nombre, appelé *nombre magique*. Par exemple :

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Lemme 1. *Le nombre magique est égal à $N(N^2+1)/2$, où N est la taille de l'arête du carré.*

Démonstration. Si N est la longueur de l'arête du carré, alors les nombres à l'intérieur du carré vont de 1 à N^2. Donc leur somme est égale à $S = \sum_{i=1}^{N^2} = N^2(N^2+1)/2$. Il y a N lignes, on peut donc dire que la somme moyenne des nombres d'une ligne est égale à $S/N = N(N^2+1)/2$. Mais comme les nombres de toutes les lignes ont la même somme, il en résulte que, si le carré magique existe, *chaque* ligne a une somme de $S/N = N(N^2+1)/2$. □

Pour accélérer les calculs, on va utiliser des nombres entiers en implémentation machine : on définira chacune des variables comme :

```
v = BitVec('x%dy%d' % (x, y), 32)
```

où x et y sont les coordonnées, ce sont donc des entiers sur 32 bits.

Voici le plan : pour une taille de carré donnée,

1) on crée des variables de ce type pour toutes les cases du carré
2) on déclare que leurs valeurs sont distinctes
3) on calcule le nombre magique pour la taille donnée
4) on déclare que la somme de toute ligne est égale au nombre magique (utiliser la fonction Z3 `Sum`)
5) on déclare que la somme de toute colonne est égale au nombre magique
6) on déclare que la somme de toute diagonale est égale au nombre magique
7) on résout.

SOLUTION

```
import sys
import time
from z3 import *

def solve_magic_square(size):
    """Try to a solution for a size-magic square"""
    def column(matrix, i):
        """Get the column i of matrix"""
        return [matrix[j][i] for j in range(size)]

    def get_diagonals(matrix):
        """Get the diagonals of matrix"""
        return ([matrix[i][i] for i in range(size)], [matrix[i][size - i - 1] for i in range(size)])

    def get_constrained_int(x, y, s):
        """Get an Int and add the constraints associated directly in the solver"""
        # Int() is really really slower!
        x = BitVec('x%dy%d' % (x, y), 32)
        s.add(x > 0, x <= size**2)
        return x

    s = Solver()
    magic = (size * (size**2 + 1)) / 2
    matrix = [[get_constrained_int(y, x, s) for y in range(size)] for x in range(size)]

    # Each value must be different
    s.add(Distinct([matrix[i][j] for j in range(size) for i in range(size)]))

    for i in range(size):
        # Sum of each line, column must be equal to magic
        s.add(Sum(matrix[i]) == magic, Sum(column(matrix, i)) == magic)

    # Sum of each diagonal must be equal to magic
    d1, d2 = get_diagonals(matrix)
    s.add(Sum(d1) == magic, Sum(d2) == magic)
```

```
    if s.check() == unsat:
        raise Exception('The problem is not sat')

    m = s.model()
    return [[m[val].as_long() for val in line] for line in matrix], magic

def display_magic_square(s, magic):
    """Display the magic square with the solution"""
    print('Magic value: %d' % magic)
    for i in range(len(s)):
        print(('%.3d|' * len(s)) % tuple(s[i]))

def main(argc, argv):
    if argc < 2:
        print('Usage: ./magic_square_z3 <n>')
        return -1

    n = int(argv[1], 10)
    print('Trying to find a solution for a %d-magic square..' % n)
    t1 = time.time()
    s, magic = solve_magic_square(n)
    t2 = time.time()

    print('Found a solution in %ds:' % (t2 - t1))
    display_magic_square(s, magic)
    return 1

if __name__ == '__main__':
    sys.exit(main(len(sys.argv), sys.argv))
```

La *Melancholia* de Dürer (1514)

Section 5

Retour aux bases : les règles d'inférence

Règles d'inférence

- Rappelons la notation : si, à travers la règle d'inférence r, à partir de plusieurs énoncés $\alpha_1, \ldots, \alpha_n$ on peut inférer (= déduire, prouver) un énoncé β, alors on écrit

$$\frac{\alpha_1, \ldots, \alpha_n}{\beta}\, r.$$

- Deux règles d'inférence sont indispensables, la première est celle du *modus ponens* :

$$\frac{\alpha \to \beta \qquad \alpha}{\beta}\, \textit{modus ponens}$$

- et la deuxième est celle de la *généralisation* :

$$\frac{A}{\forall x\, A}\, \text{généralisation}.$$

Exemple

Montrons que $\forall x(P(x) \rightarrow P(x))$:

(1) $P(a) \rightarrow ((P(a) \rightarrow P(a)) \rightarrow P(a))$ Ax. 1
(2) $P(a) \rightarrow ((P(a) \rightarrow P(a)) \rightarrow P(a)) \rightarrow ((P(a) \rightarrow (P(a) \rightarrow P(a)) \rightarrow (P(a) \rightarrow P(a))))$ Ax. 2
(3) $(P(a) \rightarrow (P(a) \rightarrow P(a))) \rightarrow (P(a) \rightarrow P(a))$ (1)+(2) *mod. pon.*
(4) $(P(a) \rightarrow (P(a) \rightarrow P(a)))$ Ax. 1
(5) $P(a) \rightarrow P(a)$ (3)+(4) *mod. pon.*
(6) $\forall x(P(x) \rightarrow P(x))$ (5) général.

Ce n'est certes pas un résultat très spectaculaire, mais cela montre qu'il aurait été inutile de l'inclure parmi les axiomes.

Monty Python : brûlez la sorcière !

- Dans le film *Monty Python : Sacré Graal*, les Monty Python utilisent une logique un peu particulière pour prouver qu'une — au demeurant, très charmante — jeune femme (jouée par Connie Booth) est une sorcière.
- Nous allons montrer qu'en utilisant la méthode déductive, leur raisonnement est fallacieux.
- Mais que, par contre, en utilisant l'abduction, leur raisonnement se tient.

http://www.youtube.com/watch?v=cMvZFRTih6g&t=1m11s

Monty Python : brûlez la sorcière !

- (1) Que fait-on avec les sorcières ? On les brûle !
- (2) Que brûle-t-on d'autre ? Le bois !
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \text{Witch}(X) \rightarrow \text{Burns}(X)$
- (2) Que brûle-t-on d'autre ? Le bois !
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \text{Witch}(X) \rightarrow \text{Burns}(X)$
- (2) $\forall X, \text{Wood}(X) \rightarrow \text{Burns}(X)$
- (3) Que fait le bois ? Il flotte.
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X$, Witch(X) $\rightarrow$ Burns(X)
- (2) $\forall X$, Wood(X) $\rightarrow$ Burns(X)
- (3) $\forall X$, Wood(X) $\rightarrow$ Floats(X)
- (4) Qui d'autre flotte ? Un canard.
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X$, Witch$(X) \rightarrow$ Burns(X)
- (2) $\forall X$, Wood$(X) \rightarrow$ Burns(X)
- (3) $\forall X$, Wood$(X) \rightarrow$ Floats(X)
- (4) Floats(Duck)
- (5) Non énoncé, mais sous-entendu : tout ce qui a le même poids qu'un objet qui flotte, flotte aussi.
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X, \mathrm{Witch}(X) \rightarrow \mathrm{Burns}(X)$
- (2) $\forall X, \mathrm{Wood}(X) \rightarrow \mathrm{Burns}(X)$
- (3) $\forall X, \mathrm{Wood}(X) \rightarrow \mathrm{Floats}(X)$
- (4) Floats(Duck)
- (5) $\forall X, Y, (\mathrm{Floats}(X) \wedge (\mathrm{weight}(X) = \mathrm{weight}(Y))) \rightarrow \mathrm{Floats}(Y)$
- (E) Expérimentalement, on montre que la jeune femme a le même poids que le canard.
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X$, Witch(X) $\rightarrow$ Burns(X)
- (2) $\forall X$, Wood(X) $\rightarrow$ Burns(X)
- (3) $\forall X$, Wood(X) $\rightarrow$ Floats(X)
- (4) Floats(Duck)
- (5) $\forall X, Y$, (Floats(X) $\wedge$ (weight(X) = weight(Y))) $\rightarrow$ Floats(Y)
- (E) weight(Connie) = weight(Duck)
- (α) Conclusion : elle est une sorcière !

Monty Python : brûlez la sorcière !

- (1) $\forall X$, Witch(X) $\rightarrow$ Burns(X)
- (2) $\forall X$, Wood(X) $\rightarrow$ Burns(X)
- (3) $\forall X$, Wood(X) $\rightarrow$ Floats(X)
- (4) Floats(Duck)
- (5) $\forall X, Y$, (Floats(X) $\wedge$ (weight(X) = weight(Y))) $\rightarrow$ Floats(Y)
- (E) weight(Connie) = weight(Duck)
- (α) Witch(Connie)

Monty Python : brûlez la sorcière !

- (1) `ForAll(X,Implies(Witch(X),Burns(X)))`
- (2) `ForAll(X,Implies(Wood(X),Burns(X)))`
- (3) `ForAll(X,Implies(Wood(X),Floats(X)))`
- (4) `Floats(Duck)`
- (5) `ForAll([X,Y],Implies(And(Floats(X),w(X)==w(Y)), Floats(X)))`
- (E) `w(Connie)==w(Duck)`
- ($\neg\alpha$) `Not(Witch(Connie))`

Monty Python : brûlez la sorcière !

```
from z3 import *
Living, (Connie, Duck) = EnumSort('Living', ('Connie', 'Duck'))
X, Y = Consts('X Y',Living)
Witch = Function('Witch',Living,BoolSort())
Burns = Function('Burns',Living,BoolSort())
Floats = Function('Floats',Living,BoolSort())
Wood = Function('Wood',Living,BoolSort())
w = Function('Witch',Living,IntSort())
s = Solver()
s.add(ForAll(X,Implies(Witch(X),Burns(X))))
s.add(ForAll(X,Implies(Wood(X),Burns(X))))
s.add(ForAll(X,Implies(Wood(X),Floats(X))))
s.add(Floats(Duck))
s.add(ForAll([X,Y],Implies(And(Floats(X),w(X)==w(Y)), Floats(X))))
s.add(w(Connie)==w(Duck))
s.add(Not(Witch(Connie)))
print(s.check())
```

retourne, malgré tous les efforts des Monty, `sat`.

Jesse Hoey : le *modus bogus* ☺

- Sur un *registre humoristique*, Jesse Hoey introduit la règle d'inférence de l'abduction :

$$\frac{\alpha \rightarrow \beta \qquad \beta}{\alpha}\ \textit{modus bogus}.$$

- Il montre qu'avec cette règle d'inférence supplémentaire, le raisonnement des Monty Python est correct, et que la jeune fille est bel et bien une sorcière ! Voici comment il procède :

Jesse Hoey : le *modus bogus* ☺

Rappel :
(1) : $\forall X$, Witch(X) $\rightarrow$ Burns(X)
(2) : $\forall X$, Wood(X) $\rightarrow$ Burns(X)
(3) : $\forall X$, Wood(X) $\rightarrow$ Floats(X)
(4) : Floats(Duck)
(5) : $\forall X, Y$, (Floats(X) $\wedge$ (w(X) = w(Y))) $\rightarrow$ Floats(Y)

$$\dfrac{(1) \qquad \dfrac{(2) \qquad \dfrac{(3) \qquad \dfrac{(5) \qquad \dfrac{\dfrac{(4)}{\text{Floats(Duck)}} \qquad \dfrac{\text{(expérience)}}{\text{w(Connie)} = \text{w(Duck)}}}{\text{Floats(Duck)} \wedge (\text{w(Connie)} = \text{w(Duck)})}\,\wedge}{\text{Floats(Connie)}}\,m.p.}{\text{Wood(Connie)}}\,\textit{modus bogus}}{\text{Burns(Connie)}}\,\textit{modus ponens}}{\text{Witch(Connie)}}\,\textit{modus bogus}$$

Section 6

Logiques modales

Logique aléthique

- Dans l'exemple de la sorcière on a fait quelques simplifications :
  - On n'a pas tenu compte du fait que le bois « peut » brûler, alors que les sorcières « doivent » brûler.
  - Le bois et les sorcières « peuvent » flotter.
- Pour démêler cela, la *logique aléthique* introduit des opérateurs
  - $\square$ = « est nécessaire » avec $\neg\square$ = « est contingent »
  - $\lozenge$ = « est possible » avec $\neg\lozenge$ = « est impossible ».

  avec des axiomes du type
  - $\square P \rightarrow \lozenge P$ (quand c'est nécessaire, c'est possible),
  - $P \rightarrow \lozenge P$ (quand ça arrive c'est que c'est possible),
  - $\square P \rightarrow \square\square P$ (le nécessaire est nécessairement nécessaire),
  - $P \rightarrow \square\lozenge P$ (quand ça arrive c'est nécessairement possible),
  - $\lozenge P \rightarrow \square\lozenge P$ (quand c'est possible, c'est nécessairement possible), etc.

Logique épistémique

- Autre logiquel modale, la *logique épistémique* s'intéresse à la connaissance de la connaissance : on a un opérateur $\mathbf{K}_A$ qui signifie : « l'agent A sait que… ».
- Exemple : la phrase « James Bond sait que quelqu'un est espion » peut être interprétée de deux manières :
  1. $\exists x\, \mathbf{K}_{\mathrm{JB}} \mathrm{Espion}(x)$;
  2. $\mathbf{K}_{\mathrm{JB}} \exists x\, \mathrm{Espion}(x)$.

Logique épistémique

- Les règles d'inférence ne sont pas les mêmes. On a bien le *modus ponens* :
$$(\mathbf{K}_A P \wedge \mathbf{K}_A(P \rightarrow Q)) \rightarrow \mathbf{K}_A(Q).$$
- On a toujours la tautologie $\mathbf{K}_A(P \vee \neg P)$
- mais, par contre, l'énoncé $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$ *n'est pas une tautologie*, il peut y avoir des cas où on ne sait pas si P est vrai ou non.
- On admet généralement (à tort ?) que $\mathbf{K}_A P \rightarrow P$.
- Également, au grand dam de Socrate (ἓν οἶδα, ὅτι οὐδὲν οἶδα), $\mathbf{K}_A P \rightarrow \mathbf{K}_A(\mathbf{K}_A P)$.
- La logique épistémique permet la représentation de connaissances plus proche de la réalité du raisonnement humain.

Section 7

Les ontologies

Les ontologies et les domaines de connaissance

- En informatique, pour associer des mots à des concepts et définir des relations entre eux, on utilise les *ontologies*.
- Une ontologie comporte :
    - des *concepts*,
    - des *relations* entre les concepts,
    - des *attributs*,
    - une *hiérarchie de concepts*,
    - une *hiérarchie de relations* définie à partir de la hiérarchie de concepts,
    - et un ensemble de types (chaîne, entier, etc.).

Les ontologies et les domaines de connaissance

- Mathématiquement parlant :
  1. Une ontologie $\mathcal{O} := (\mathrm{C}, \leq_{\mathrm{C}}, \mathrm{R}, \sigma_{\mathrm{R}}, \leq_{\mathrm{R}}, A, \sigma_A, T)$
  2. où $\mathrm{C}, \mathrm{R}, A, T$ sont les concepts, relations, attributs, types ;
  3. $\sigma_{\mathrm{R}} : R \to \mathrm{C}^+$, $\sigma_A : A \to \mathrm{C} \times T$ sont les signatures des relations et des attributs ;
  4. $\leq_{\mathrm{C}}$ munit C d'une structure de treillis ;
  5. $\leq_{\mathrm{R}}$ est un ordre partiel de R, défini de la manière suivante : $r_1 \leq_{\mathrm{R}} r_2 \Rightarrow |\sigma_{\mathrm{R}}(r_1)| = |\sigma(r_2)|$ et $\pi_i(\sigma_{\mathrm{R}}(r_1)) \leq_{\mathrm{C}} \pi_i(\sigma_{\mathrm{R}}(r_2))$ pour tout i (ici π_i est la i-ème projection).
- Exemple : concepts `rivière`, `ville`, relation `traverse()`, signature σ_{R}(`traverse`) = (`rivière`, `ville`).

Les ontologies et les domaines de connaissance

- Si on veut peupler une ontologie à partir du texte, il faut associer des mots aux concepts. On définit un *lexique* $\mathcal{L}$ pour une ontologie $\mathcal{O}$ comme étant $\mathcal{L} := (S_C, S_R, S_A, \mathrm{Ref}_C, \mathrm{Ref}_R, \mathrm{Ref}_A)$ où S_C, S_R, S_A sont des lexèmes pour les concepts, relations et attributs, et $\mathrm{Ref}_* \subseteq S_* \times *$ des *références lexicales* pour les concepts, relations et attributs.
- Ainsi, par exemple, quand on a des synonymes (voiture, auto, bagnole, caisse) on pourra dire que $\mathrm{Ref}_C^{-1}(\texttt{voiture}) = \{\text{voiture, auto, bagnole, caisse}\}$, où `voiture` dénote le concept de voiture.

Les ontologies et les domaines de connaissance

- Une *base de connaissances* $\mathcal{K}$ pour une ontologie $\mathcal{O}$ est un quadruplet $(I, \iota_C, \iota_R, \iota_A)$ où I est un ensemble d'*instances* (ou *objets*) et ι_* des fonctions des instantiation :
$\iota_C : C \to 2^I$, $\iota_R : R \to 2^{I^+}$, $\iota_A : A \to 2^I \times \text{valeurs}(T)$;
- Même question que tout à l'heure : trouver des entités lexicales pour nos instances. Même réponse :
- Un *lexique d'instances* pour une base de connaissances $\mathcal{K}$ est une paire (S_I, R_I) où S_I sont des signes pour les instances et $R_I \subseteq S_I \times I$ est une relation qui associe des références lexicales aux instances.

Alimentation d'ontologie

- Extraire de la connaissance à partir de textes dans le but d'alimenter une ontologie revient à extraire les couches successives suivantes d'information :

1. des termes (fleuve, rivière, pays, ville, capitale, ...) et les entités nommées (Brest, IMT Atlantique, Laury Thilleman, etc.) ;
2. des synonymes { fleuve, rivière, cours d'eau, ... } ;
3. des concepts `rivière`, `capitale`, `ville`, ... (un concept est une paire de définitions intentionnelle et extensionnelle, ainsi qu'un lexème référent) ;
4. des hiérarchies de concepts `capitale` $\leq_C$ `ville` ;
5. des relations entre concepts `traverse(rivière,pays)` ;
6. des hiérarchies de relations `est_capitale` $\leq_R$ `est_situé_dans` ;
7. des schémas d'axiomes `disjoint(rivière,ville)` ;
8. des axiomes logiques généraux : $\forall x(\mathtt{pays}(x) \rightarrow \exists y\, \mathtt{capitale_de}(y, x)) \wedge \forall z(\mathtt{capitale_de}(z, x) \rightarrow y = z)$.

Section 8

Logiques de description

Les bases

- Les *logiques de description* sont des variantes de la logique du 1er ordre qui forment des compromis sur deux tableaux :
  1. elles ajoutent des nouvelles notations pour améliorer l'expressivité du langage (par ex. la possibilité de dire qu'il existe exactement n éléments ayant une propriété donnée) ;
  2. elles sont moins puissantes que la logique du 1er ordre, ce qui les rend décidables.
- On a trois types d'objets :
  1. des *individus* (ce qui correspond aux constantes de la logique du 1er ordre),
  2. des *concepts* ou *classes* (des prédicats unaires, dont l'interprétation ensembliste correspond à des ensembles d'individus),
  3. des *rôles* ou *propriétés* (des prédicats binaires, dont l'interprétation ensembliste correspond à des ensembles de paires d'individus).

Protégé : l'ontologie wildlife.owl

Un logiciel libre pour gérer des ontologies OWL.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

< > wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) Search...

Active ontology × | Entities × | Individuals by class × | DL Query ×

Ontology header:

Ontology IRI file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl

Ontology Version IRI e.g. file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.ow

Annotations

owl:VersionInfo

Version du 17 octobre 2002

Ontology metrics:

Metrics

Axiom	58
Logical axiom count	21
Declaration axioms count	23
Class count	13
Object property count	3
Data property count	0
Individual count	3
Annotation Property count	5

Class axioms

SubClassOf	8
EquivalentClasses	2
DisjointClasses	1
GCI count	0
Hidden GCI Count	0

Object property axioms

General class axioms:

General class axioms

Ontology imports | Ontology Prefixes | General class axioms

Imported ontologies:

Direct Imports

Indirect Imports

Protégé : l'ontologie wildlife.owl

```
<rdf:RDF xmlns="http://www.mydomain.org/african"
     xml:base="http://www.mydomain.org/african"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="file:wildlife.owl">
        <owl:VersionInfo>
            Version du 17 octobre 2002
        </owl:VersionInfo>
    </owl:Ontology>
    ...
</rdf:RDF>
```

Protégé : visualisation

Avec l’onglet OntoGraf.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl)

Search...

Active ontology | Entities | Individuals by class | DL Query | OntoGraf

Class hierarchy: owl:Thing

Asserted

- owl:Thing
  - animal
    - carnivore
      - lion
    - herbivore
      - giraffe
  - branche
  - feuille
  - owl:allValuesFrom
  - owl:onProperty
  - owl:unionOf
  - plante
    - arbre
    - plante–comestible

OntoGraf:

Search: contains Search Clear

owl:onProperty
giraffe
feuille
owl:unionOf
herbivore
lion
animal
branche
owl:Thing
carnivore
arbre
plante
plante–comestible
owl:allValuesFrom

Protégé : classes

Dans `wildlife.owl`, les animaux forment une classe.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

< > wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) Search...

〉animal

Active ontology × Entities × Individuals by class × DL Query × OntoGraf ×

Annotation properties | Datatypes | Individuals
Classes | Object properties | Data properties

Class hierarchy: animal

Asserted

- owl:Thing
  - animal
    - herbivore
      - giraffe
    - carnivore
      - lion
  - branche
  - feuille
  - owl:allValuesFrom
  - owl:onProperty
  - owl:unionOf
  - plante
    - arbre
    - plante-comestible

animal — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl#animal

Annotations | Usage

Annotations: animal

Annotations
rdfs:comment
Les animaux forment une classe

Description: animal

Equivalent To

SubClass Of

General class axioms

SubClass Of (Anonymous Ancestor)

Instances

Target for Key

Disjoint With
plante

Disjoint Union Of

Protégé : classes

Les animaux forment une classe.

```
<owl:Class rdf:about="file:wildlife.owl#animal">
    <owl:disjointWith rdf:resource="file:wildlife.owl#plante"/>
    <rdfs:comment>
        Les animaux forment une classe
    </rdfs:comment>
</owl:Class>
```

Dans le formalisme de la logique de description, on écrira simplement « Animal » pour dénoter cette classe.

Les giraffes sont des herbivores qui sont des animaux :

$\text{Giraffe} \sqsubset \text{Herbivore} \sqsubset \text{Animal}$

Protégé : individus

Gigi et Giginou sont des giraffes différentes.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

< > wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) Search...

animal

Active ontology × Entities × Individuals by class × DL Query × OntoGraf ×

Annotation properties | Datatypes | Individuals
Classes | Object properties | Data properties

Gigi — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl#Gigi

Annotations | Usage

Individuals: Gigi

- Gigi
- Giginou
- Léo

Annotations: Gigi

Annotations

Description: Gigi

Types
giraffe

Same Individual As

Different Individuals
Giginou

Property assertions: Gigi

Object property assertions

Data property assertions

Negative object property assertions

Negative data property assertions

Protégé : individus

Gigi et Giginou sont des giraffes différentes.

```
<owl:NamedIndividual rdf:about="file:wildlife.owl#Gigi">
    <rdf:type rdf:resource="file:wildlife.owl#giraffe"/>
</owl:NamedIndividual>
<owl:NamedIndividual rdf:about="file:wildlife.owl#Giginou">
    <rdf:type rdf:resource="file:wildlife.owl#giraffe"/>
</owl:NamedIndividual>
<rdf:Description>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#AllDifferent"
    <owl:distinctMembers rdf:parseType="Collection">
        <rdf:Description rdf:about="file:wildlife.owl#Gigi"/>
        <rdf:Description rdf:about="file:wildlife.owl#Giginou"/>
    </owl:distinctMembers>
</rdf:Description>
```

Giraffe(Gigi)
Giraffe(Giginou)
Gigi $\not\approx$ Giginou

Protégé : rôles

Un animal peut manger qqch. Manger est l'inverse d'être mangé.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl)

Search...

mange

Active ontology | Entities | Individuals by class | DL Query | OntoGraf

Annotation properties | Datatypes | Individuals | Classes | Object properties | Data properties

mange — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl#mange

Annotations | Usage

Object property hierarchy: mange

Asserted

- owl:topObjectProperty
  - is-part-of
  - mange
  - mangé-par

Annotations: mange

Annotations

Chara

- Functional
- Inverse functic
- Transitive
- Symmetric
- Asymmetric
- Reflexive
- Irreflexive

Description: mange

Equivalent To

SubProperty Of

Inverse Of
- mangé-par

Domains (intersection)
- animal

Ranges (intersection)

Disjoint With

Protégé : rôles

Un animal peut manger qqch. Manger est l’inverse d’être mangé.

```
<owl:ObjectProperty rdf:about=”file:wildlife.owl#mange”>
    <owl:inverseOf rdf:resource=”file:wildlife.owl#mangé-par”/>
    <rdfs:domain rdf:resource=”file:wildlife.owl#animal”/>
</owl:ObjectProperty>
```

$\text{Mange} \equiv \text{Est-mangé-par}^{-}$
$\text{Est-mangé-par} \equiv \text{Mange}^{-}$

Protégé : rôles

Léo mange Gigi.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

< > wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) Search...

Active ontology × Entities × Individuals by class × DL Query × OntoGraf ×

Annotation properties | Datatypes | Individuals
Classes | Object properties | Data properties

Individuals: Léo

- Gigi
- Giginou
- Léo

Léo — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl#Léo

Annotations | Usage

Usage: Léo

Show: this different

Found 6 uses of Léo

- Léo
  - Léo Type lion
  - Individual: Léo
  - Léo mange Gigi

Description: Léo

Types
lion

Same Individual As

Different Individuals

Property assertions: Léo

Object property assertions
mange Gigi

Data property assertions

Negative object property assertions

Negative data property assertions

Protégé : rôles

Léo mange Gigi.

```
<owl:NamedIndividual rdf:about="file:wildlife.owl#Léo">
    <rdf:type rdf:resource="file:wildlife.owl#lion"/>
    <wildlife:mange rdf:resource="file:wildlife.owl#Gigi"/>
</owl:NamedIndividual>
```

Mange(Léo,Gigi)

Les bases

- On définit trois types de formules :
  1. celles de l'*ABox*, « A » comme « assertion », qui décrivent des concepts et des rôles (prédicats unaires et binaires sur des constantes) ;
  2. celles de la *TBox*, « T » comme « terminologie », qui décrivent des relations entre concepts ;
  3. celles de la *RBox*, « R » comme « relation », qui décrivent une hiérarchie des rôles, des compositions de rôles et des relations entre rôles, comme l'exclusion mutuelle, la réflexivité, la symétrie, la transitivité, etc.

ABox

- Une *assertion de concept* est un prédicat unaire du type Giraffe(Gigi).
- Une *assertion de rôle* est un prédicat binaire Mange(Léo,Gigi).
- Dans les logiques de description on ne fait pas l'*hypothèse de l'unicité des noms* : deux individus de même nom peuvent avoir le même référent, on écrira cloclo $\approx$ claudeFrançois, et samsonFrançois $\not\approx$ claudeFrançois.

TBox

- Puisqu'on peut interpréter les *concepts* (prédicats unaires) comme des ensembles, on peut aussi utiliser des relations de théorie d'ensembles :
- Mère $\sqsubset$ Parent, qui équivaut à $\forall X\, \text{Mère}(X) \rightarrow \text{Parent}(X)$;
- Personne $\equiv$ Humain, qui équivaut à $\forall X\, \text{Personne}(X) \leftrightarrow \text{Humain}(X)$.

RBox

- La *hiérarchie des concepts* induit une *hiérarchie des rôles* :
- ParentDe $\sqsubset$ AncêtreDe, qui équivaut à $\forall X, Y\, \mathrm{ParentDe}(X, Y) \rightarrow \mathrm{AncêtreDe}(X, Y)$.
- Une relation binaire peut se combiner avec une autre : sachant que le frère d'un père est un oncle, on peut écrire
- FrèreDe $\circ$ ParentDe $\sqsubset$ OncleDe, qui équivaut à $\forall X, Y, Z\, \mathrm{FrèreDe}(X, Y) \wedge \mathrm{ParentDe}(Y, Z) \rightarrow \mathrm{OncleDe}(X, Z)$.
- Enfin, on peut donner des caractéristiques générales des concepts : Disjoint(ParentDe, EnfantDe), ce qui équivaut à $\forall X, Y\, \mathrm{ParentDe}(X, Y) \rightarrow \neg \mathrm{Parent}(Y, X)$.

Restriction existentielle

- Toujours en mimant la théorie des ensembles, on peut écrire Mère $\equiv$ Femme $\sqcap$ Parent, qui équivaut à $\forall X\, \text{Mère}(X) \leftrightarrow \text{Femme}(X) \wedge \text{Parent}(X)$,
- ou Parent $\equiv$ Père $\sqcup$ Mère, qui équivaut à $\forall X\, \text{Parent}(X) \leftrightarrow \text{Mère}(X) \vee \text{Père}(X)$,
- la négation correspond au complémentaire d'un ensemble : $\neg$Célibataire est équivalent à Marié.

Restriction existentielle

- L'ensemble complet correspond à un prédicat qui est toujours vrai, on l'écrit $\top$, pour exprimer une partition on écrira $\top \sqsubset \text{Homme} \sqcup \text{Femme}$.
- L'ensemble vide correspond à un prédicat qui est toujours faux, on l'écrit $\perp$, pour exprimer une exclusion mutuelle on écrira $\text{Homme} \sqcap \text{Femme} \sqsubset \perp$.
- Sous Protégé, $\top$ s'écrit owl:Thing et $\perp$ s'écrit owl:Nothing, ce deux classes sont hardcodées.

Restriction existentielle

- Imaginons qu'on a un rôle Parent(X, Y) et que l'on cherche à caractériser les X qui sont parents. Il s'agit donc de dire « je cherche les X pour lesquels $\exists Y$ tel que Parent(X, Y) », on écrira $\exists\text{Parent}.\top$;
- le $\top$ signifie qu'on prend les X pour lesquels il existe un Y, sans les filtrer davantage.
- Si je cherchais ceux qui sont des parents d'au moins une fille, j'écrirais $\exists\text{Parent}.\text{Fille}$.

Protégé : restriction existentielle

Les plantes comestibles sont des plantes qui sont mangées par des herbivores et par des carnivores.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

< > wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) Search...

plante 〉 plante-comestible

Active ontology × | Entities × | Individuals by class × | DL Query × | OntoGraf ×

Annotation properties | Datatypes | Individuals
Classes | Object properties | Data properties

Class hierarchy: plante-comestible

Asserted

- owl:Thing
  - **animal**
    - **herbivore**
      - **giraffe**
    - **carnivore**
      - **lion**
  - **branche**
  - **feuille**
  - owl:allValuesFrom
  - owl:onProperty
  - owl:unionOf
  - **plante**
    - **arbre**
    - **plante-comestible**

plante-comestible — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.ov

Annotations | Usage

Annotations: plante-comestible

Annotations

rdfs:comment

Les plantes comestibles sont des plantes qui sont mangées par des herbivores et par des carnivores

Description: plante-comestible

Equivalent To

SubClass Of

- **mangé-par** some **carnivore**
- **mangé-par** some **herbivore**
- **plante**

General class axioms

SubClass Of (Anonymous Ancestor)

Instances

Target for Key

Protégé : restriction existentielle

Les plantes comestibles sont des plantes qui sont mangées par des herbivores.

```
<owl:Class rdf:about="file:wildlife.owl#plante-comestible">
    <rdfs:subClassOf rdf:resource="file:wildlife.owl#plante"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="file:wildlife.owl#mangé-par"/
            <owl:someValuesFrom rdf:resource="file:wildlife.owl#carnivo
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:comment>
        Les plantes comestibles sont des plantes qui sont mangées par d
    </rdfs:comment>
</owl:Class>
```

$$\text{Plante-comestible} \sqsubseteq \exists \text{Est-mangé-par.Herbivore}$$

Restriction universelle

- De même, ceux qui
  1. n'ont que des filles, ou
  2. ne sont pas parents

  sont dénotés par : $\forall \mathtt{Parent}.\mathtt{Fille}$.
- Mais que signifie alors $\forall \mathtt{Parent}.\top$?

Restriction universelle

- Pour le savoir, prenons les définitions formelles des sémantiques de ces notations.
- Soit R un rôle, et $R^{\mathbf{I}}$ son interprétation. Rappelons que dans une interprétation ensembliste, $R^{\mathbf{I}}$ devient un ensemble de paires d'éléments $R^{\mathbf{I}} = \{(x^{\mathbf{I}}, y^{\mathbf{I}}), \ldots\}$ du domaine Δ. On dira que $y^{\mathbf{I}}$ est un R-*successeur* de $x^{\mathbf{I}}$ si la paire $(x^{\mathbf{I}}, y^{\mathbf{I}})$ appartient à $R^{\mathbf{I}}$.
- Soit C un concept (et donc $C^{\mathbf{I}}$ est un sous-ensemble de Δ). Alors l'interprétation de $\exists R.C$ est $\{x^{\mathbf{I}} \mid \text{quelques successeurs de } x^{\mathbf{I}} \text{ sont dans } C^{\mathbf{I}}\}$.
- Et celle de $\forall R.C$ est $\{x^{\mathbf{I}} \mid \text{tous les successeurs de } x^{\mathbf{I}} \text{ sont dans } C^{\mathbf{I}}\}$.
- Donc, quelle est l'interprétation de $\forall \text{Parent}.\top$?

Protégé : restriction universelle

Les giraffes sont des herbivores qui ne mangent que des feuilles.

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl) : [/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl]

wildlife (file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl)

Search...

animal 〉 herbivore 〉 giraffe

Active ontology × | Entities × | Individuals by class × | DL Query × | OntoGraf ×

Annotation properties | Datatypes | Individuals

Classes | Object properties | Data properties

Class hierarchy: giraffe

Asserted

- owl:Thing
  - **animal**
    - **herbivore**
      - **giraffe**
    - **carnivore**
      - **lion**
  - **branche**
  - **feuille**
  - owl:allValuesFrom
  - owl:onProperty
  - owl:unionOf
  - **plante**
    - **arbre**
    - **plante-comestible**

giraffe — file:/Users/yannis/Google_Drive/yannis/texmf/cours/ue-tlft/wildlife.owl#giraffe

Annotations | Usage

Annotations: giraffe

Annotations

rdfs:comment

Les giraffes sont herbivores et ne mangent que des feuilles

rdfs:subClassOf

Description: giraffe

Equivalent To

SubClass Of

- **herbivore**
- **mange only feuille**

General class axioms

SubClass Of (Anonymous Ancestor)

Instances

- **Gigi**
- **Giginou**

Protégé : restriction universelle

Les giraffes sont des herbivores qui ne mangent que des feuilles.

```
<owl:Class rdf:about=”file:wildlife.owl#giraffe”>
    <rdfs:subClassOf rdf:resource=”file:wildlife.owl#herbivore”/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource=”file:wildlife.owl#mange”/>
            <owl:allValuesFrom rdf:resource=”file:wildlife.owl#feuille”
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:comment>
        Les giraffes sont herbivores et ne mangent que des feuilles
    </rdfs:comment>
    <rdfs:subClassOf></rdfs:subClassOf>
</owl:Class>
```

$$\text{Giraffe} \sqsubset \text{Herbivore} \sqcap \forall \text{Mange}.\text{Feuille}$$

Restrictions au-plus et au-moins, réflexivité

- De même que l'on peut demander au moins un successeur ou tous les successeurs, on peut aussi spécifier « au moins n successeurs » : $\geq nR.C$, ainsi qu'« au plus n successeurs » : $\leq nR.C$.
- Une autre notation permet de décrire l'ensemble des éléments pour lesquels un rôle R est réflexif : $\exists R.\mathit{Self}$.
- Une manière de décrire un concept est en donnant explicitement ses membres : au lieu de Parent(jacques,julie), on peut aussi écrire $\{\text{jacques}\} \sqsubset \exists\text{Parent}.\{\text{julie}\}$.

Restrictions au-plus et au-moins, réflexivité

- On note R^- le *rôle inverse* de R, par exemple Parent^- est équivalent à Enfant puisque $\forall X, Y\, \text{Parent}(X, Y) \leftrightarrow \text{Enfant}(Y, X)$.
- Enfin, on note U le *rôle universel*, c'est-à-dire celui qui associe chaque élément à chaque autre élément (y compris lui-même).
- Enfin, on note R^+ la *clôture transitive* de R, c'est-à-dire le plus petit rôle transitif contenant R.

Logique $\mathcal{SROIQ}$

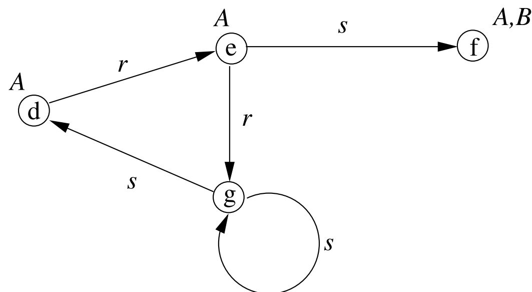
- $\mathcal{SROIQ}$ est une logique de description, telle que : Si N_C est un concept quelconque, N_R un rôle quelconque et N_I un individu quelconque, alors on définit de manière itérative :
  1. une *expression de concept* $\mathbf{C}$ par $\mathbf{C} ::= N_C \mid (\mathbf{C} \sqcap \mathbf{C}) \mid \mathbf{C} \sqcup \mathbf{C}) \mid \neg\mathbf{C} \mid \top \mid \bot \mid \exists\,\mathbf{R}.\mathbf{C} \mid \forall\,\mathbf{R}.\mathbf{C} \mid \geq n\,\mathbf{R}.\mathbf{C} \mid \leq n\,\mathbf{R}.\mathbf{C} \mid \exists\,\mathbf{R}.\mathit{Self} \mid \{N_I\}$,
  2. une *expression de rôle* $\mathbf{R}$ par $\mathbf{R} ::= U \mid N_R \mid N_R^-$.
- Les axiomes d'une ontologie $\mathcal{SROIQ}$ sont des types suivants :
  1. ABox : $\mathbf{C}(N_I)$, $\mathbf{R}(N_I, N_I)$, $N_I \approx N_I$, $N_I \not\approx N_I$,
  2. TBox : $\mathbf{C} \sqsubset \mathbf{C}$, $\mathbf{C} \equiv \mathbf{C}$,
  3. RBox : $\mathbf{R} \sqsubset \mathbf{R}$, $\mathbf{R} \equiv \mathbf{R}$, $\mathbf{R} \circ \mathbf{R} \sqsubset \mathbf{R}$, $\mathit{Disjoint}(\mathbf{R}, \mathbf{R})$.

Propriétés de $\mathcal{SROIQ}$

- Voici les propriétés de la logique de description $\mathcal{SROIQ}$:
  1. $\mathcal{S}$ (également appelé $\mathcal{ALCR}^+$) : ($\mathcal{AL}$) présence de noms de concepts et de rôles, de $\top$, du constructeur $\sqcap$ (conjonction), du quantificateur universel, ($\mathcal{C}$) de la négation de concept, ($\mathcal{R}^+$) de la clôture transitive ;
  2. $\mathcal{R}$: inclusion de rôles, réflexivité, irréflexivité, exclusion de rôles ;
  3. $\mathcal{O}$: possibilité de décrire un concept extensionnellement ($\{\mathrm{N}_{I,1}, \ldots, \mathrm{N}_{I,n}\}$) ;
  4. $\mathcal{I}$: possibilité d'avoir des rôles inverses (R^-) ;
  5. $\mathcal{Q}$: possibilité d'avoir des restrictions pleinement quantifiées ($\leq n$, $\geq n$).
- La norme OWL 2 du consortium WWW correspond à la logique $\mathcal{SROIQ}$.

(Nos interprétations sont de type « monde fermé », c'est-à-dire que si on n'affirme pas explicitement qu'un individu appartient à une classe, alors on considère qu'il n'appartient pas à cette classe.)

Dans l'interprétation I ci-dessous (sur le domaine $\Delta = \{d, e, f, g\}$) :



énumérer les éléments des concepts suivants :

$A \sqcup B$
$\exists s.A$
$\exists s.\neg A$
$\forall s.A$
$\exists s.\exists s.\exists s.\exists s.A$
$\forall r.A \sqcap \forall r.\neg A$
$\neg\exists r.(\neg A \sqcap \neg B)$
$\exists r.(A \sqcap \forall s.\neg B) \sqcap \neg\forall r.\exists r.(A \sqcup \neg A)$

Logique de description : exercice 2

- Modéliser en logique de description l'assertion « Tous les camions ayant une remorque sont munis d'exactement 18 roues ».
- On utilisera le rôle « est-muni-de » et les concepts « Roue », « Remorque », « Camion ».

Logique de description : exercice 2

- Modéliser en logique de description l'assertion « Tous les camions ayant une remorque sont munis d'exactement 18 roues ».
- On utilisera le rôle « est-muni-de » et les concepts « Roue », « Remorque », « Camion ».
- En logique de description :
  $(1\text{est-muni-de}.\text{Remorque} \sqcap \text{Camion}) \sqsubseteq 18\text{est-muni-de}.\text{Roue}.$

Logique de description : exercice 2

- Modéliser en logique de description l'assertion « Tous les camions ayant une remorque sont munis d'exactement 18 roues ».
- On utilisera le rôle « est-muni-de » et les concepts « Roue », « Remorque », « Camion ».
- En logique de description :
  $(1\text{est-muni-de}.\text{Remorque} \sqcap \text{Camion}) \sqsubseteq 18\text{est-muni-de}.\text{Roue}.$
- N'y aurait-il pas un piège ?

Logique de description : exercice 2

- Modéliser en logique de description l'assertion « Tous les camions ayant une remorque sont munis d'exactement 18 roues ».
- On utilisera le rôle « est-muni-de » et les concepts « Roue », « Remorque », « Camion ».
- En logique de description :
  $(1\text{est-muni-de}.\text{Remorque} \sqcap \text{Camion}) \sqsubseteq 18\text{est-muni-de}.\text{Roue}$.
- N'y aurait-il pas un piège ?
- 18est-muni-de.Roue signifie qu'il existe exactement 18 individus de la classe Roue dont le camion est muni. Mais qui nous garantit qu'ils ne représentent pas les mêmes éléments du domaine dans une interprétation donnée ? Il faudrait les nommer et dire qu'ils sont différents deux-à-deux.

Section 9

Graphes conceptuels

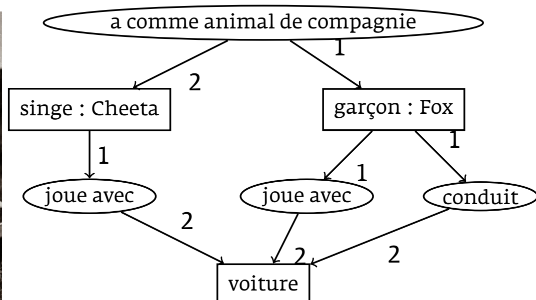
Graphes conceptuels

- Une invention française (Chein & Mugnier).
- Pour faire le lien entre le domaine que l'on veut décrire et la structure de graphe, on définit la notion de *vocabulaire* :
- Un *vocabulaire de graphe conceptuel* est un triplet $(T_C, T_R, \mathcal{I})$, où T_C est l'ensemble de *types de concept*, il est partiellement ordonné et a un plus grand élément $\top$; T_R est l'ensemble de *symboles de rélation*, il est également partiellement ordonné et chacun de ses éléments a une *arité* ≥ 1 ; $\mathcal{I}$ est l'ensemble des *marqueurs d'individu*. On ajoute à $\mathcal{I}$ l'élément $*$ (*marqueur générique*) avec la propriété $\forall i \in \mathcal{I}, i \geq *$. Ces ensembles sont mutuellement disjoints.

Graphes conceptuels

- Un *graphe conceptuel de base* sur un vocabulaire $\mathcal{V}$ donné, est la donnée d'un graphe G biparti, de partitions C et R, non-orienté avec éventuellement des arêtes multiples et d'une fonction ℓ, définis de la manière suivante :
- les sommets sont des *concepts* C et des *rélations* R
- l'image de $c \in C$ par ℓ est une paire (t, i) avec $t \in T_C$ et $i \in \mathcal{I}$;
- pour $r \in R$, $\ell(r) \in T_R$;
- le degré de $r \in R$ est égal à l'arité de $\ell(r)$;
- les arêtes contiguës à $r \in R$ sont numérotées de 1 jusqu'à $\deg(r)$.

Graphes conceptuels, exemple

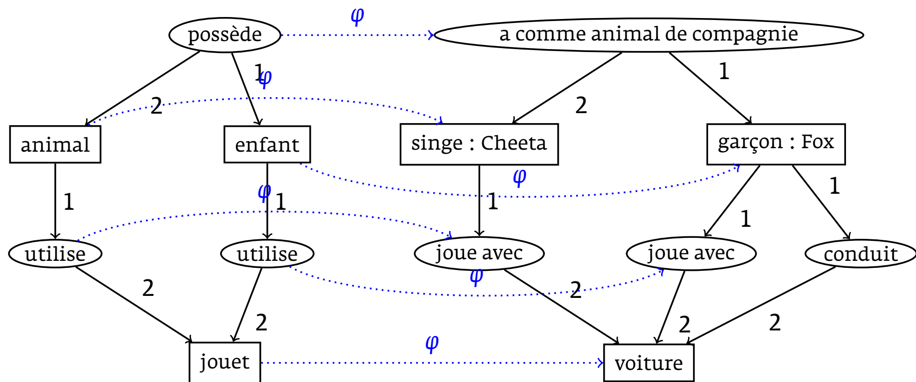


La photo est interprétée par « Fox est un garçon, Cheeta est un singe. Cheeta est l'animal de compagnie de Fox. Ils jouent ensemble avec une voiture-jouet, conduite par Fox. »
Ici, « singe », « garçon » et « voiture » sont des types de concepts, « joue avec » et « conduit » sont des rélations d'arité 2 (sujet, COD), « Cheeta » et « Fox » sont des marqueurs d'individu, alors que le sommet du bas est en fait « voiture : $*$ » (on ne note pas le marqueur générique).

Graphes conceptuels

- On définit un *ordre des concepts* à partir de l'ordre des types de concept : si $(t,i),(t',i')\in C$, alors $(t,i)\leq(t',i')$ si et seulement si $t\leq t'$ et $i\leq i'$.
- Si $t\leq t'$ on dira que t' est un *hypéronyme* de t.
- (À noter que $i\leq i'$ ne peut arriver que si $i=i'$ ou si $i'=*$, puisque les i,i' différents de $*$ ne peuvent être comparés.)
- Rappel d'algèbre : un *homomorphisme* entre deux structures est une application qui respecte leurs loins internes.
- Si G et G' sont des graphes conceptuels sur le même vocabulaire, un *homomorphisme de graphes conceptuels* est un homomorphisme de graphes φ
  - qui envoie C dans C',
  - R dans R',
  - $\mathcal{I}\cup\{*\}$ dans $\mathcal{I}'\cup\{*\}$,
  - qui est tel que $\varphi(c)\leq c$, $\varphi(r)\leq r$
  - et qui respecte les numéros des arêtes.

Graphes conceptuels



Graphes conceptuels

- Soient G et G' deux graphes conceptuels sur le même vocabulaire. On définit la *rélation de subsumption* $\succeq$ de la manière suivante : $G \succeq G'$ ssi il existe un homomorphisme de graphe conceptuel $\varphi \colon G \to G'$.
- Soit Q et G des graphes conceptuels sur le même vocabulaire. On dira que Q est une requête acceptée par G si $Q \succeq G$. Les résultats de la requête sont les images $\varphi(Q)$ pour tout homomorphisme $\varphi \colon Q \to G$.

Graphes conceptuels

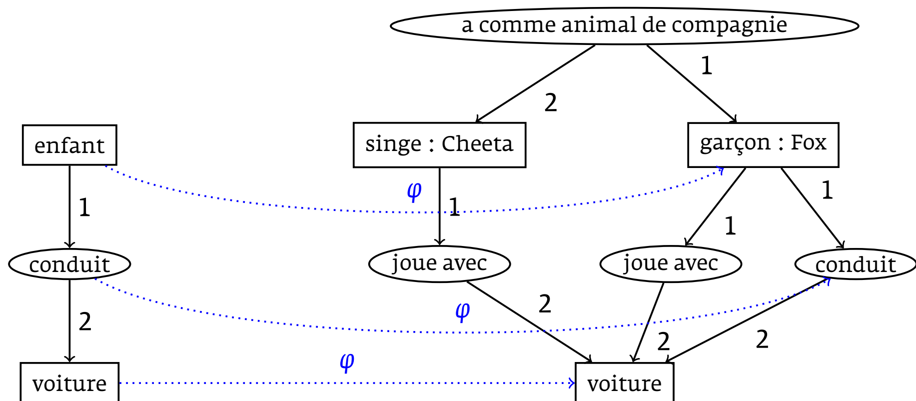
- On peut imaginer diverses situations où la notion de requête peut être utile :

(1) on considère que G est un graphe conceptuel qui décrit la réalité et on voudrait savoir si Q est « conforme » à cette réalité (est-ce que Q est une « conséquence » de G) ;

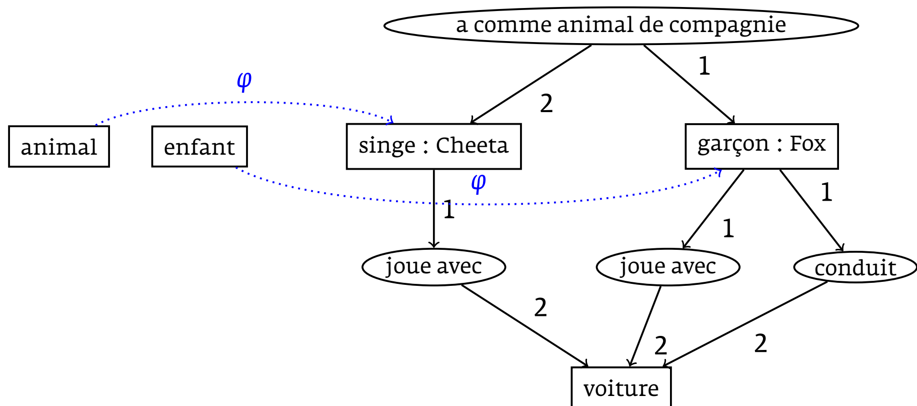
(2) G décrit l'image ci-dessus, Q correspond à une requête d'image basée sur une description (utilisant le même vocabulaire). Est-ce que Q nous permettra de trouver G, et pour quelle raison ?

- Etc.

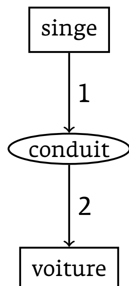
« Un enfant conduit une voiture »

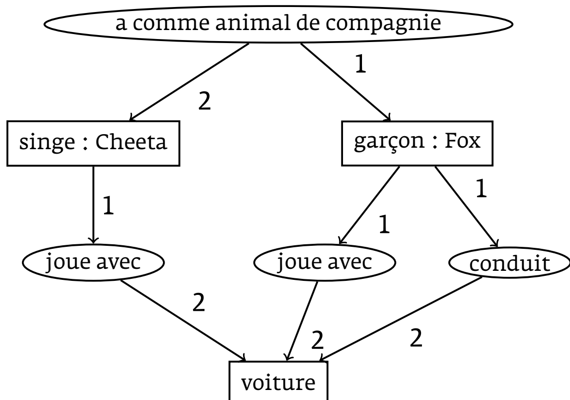


« Un enfant possède un animal »



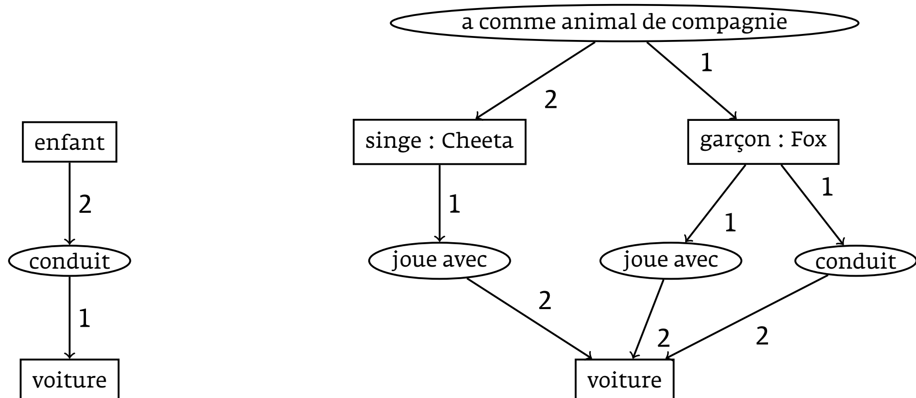
« Un singe conduit une voiture »





Impossible puisque l'arête entre « singe » et « conduit » ne peut être envoyée à « singe : Cheeta »—« conduit » qui n'existe pas. De même, on ne peut pas envoyer « conduit » vers « joue avec » puisqu'elles sont incomparables.

«Une voiture conduit un enfant»



Impossible puisque l'application qui envoie «enfant» à «garçon : Fox», «conduit» à «conduit» et «voiture» à «voiture», n'est pas un homomorphisme de graphes contextuels puisqu'elle ne respecte pas les numéros des arêtes.