1.3 Ambiguïtés de Portée

Pour traiter de l'ambiguïté de portée des quantificateurs on cherche à implémenter la méthode du *Cooper Storage*.

L'idée de la méthode est de remplacer les expressions quantifiées par autant de variables au moment de la composition. Les contributions réelles des expressions sont stockées au sein d'une liste. Lorsque la composition est terminée, on retire une à une les expressions de cette liste et on les applique successivement à la formule obtenue compositionnellement.

Selon l'ordre dans lequel les expressions sont récupérées, différentes portées vont être générées. Ce n'est donc plus une ambiguïté de type qui va décider de la portée des expressions quantifiées.

1.3.1 Exemple

L'exemple (1a) a les deux lectures indiquées en (1b) et (1c).

(1) a. Tout oiseau aime une libellule.

b. $\forall x.(bird(x) \rightarrow \exists y.(dragonfly(y) \wedge love(x,y)))$

c. $\exists y.(dragonfly(y) \wedge \forall x.(bird(x) \rightarrow love(x,y)))$

Pour les obtenir on considère que le "cœur" de la formule est fourni par le verbe, i.e. $love(x,y)$. Chacune des expressions référentielles peut être envisagé comme une formule "à trou", i.e. :

- Tout oiseau : $\forall x.(bird(x) \rightarrow \psi)$
- une libellule : $\exists y.(dragonfly(y) \wedge \phi)$

Pour construire la formule finale on sait que la contribution du verbe doit faire partie de chacune des formules associées aux expressions quantifiées. Mais la relation entre les expressions quantifiées est sous-spécifiée : on ne sait pas laquelle fait partie de l'autre. On a donc deux possibilités pour reconstruire la formule finale :

- on identifie ψ à $love(x,y)$, et donc ϕ à la contribution de *tout oiseau* : l'existentiel a portée large
- on identifie ϕ à $love(x,y)$, et donc ψ à la contribution de *une libellule* : l'universel a portée large

1.3.2 Implémentation

Pour implémenter le Cooper Storage :

- Étudiez la grammaire `gram0-cs.fcfg` pour comprendre le principe du Cooper Storage appliqué aux noms propres (pour vérifier vos résultats, utilisez le script `scriptCS-155.py`)
- Étendez la grammaire pour qu'elle traite les *NP* quantifiés :
  - Limitez vous d'abord au cas des verbes intransitifs (donc sans ambiguïté de portée)
  - Introduisez ensuite les verbes transitifs et vérifiez que plusieurs lectures sont bien générées. Des ambiguïtés sont-elles générées dans le cas de *NP* non quantifiés ?
  - Introduisez les verbes ditransitifs et observez vos résultats ; sont-ils satisfaisants ?
- Étendez la grammaires à tous les phénomènes mentionnés dans le TP précédent.
- Vous pouvez modifier le script `python` pour adapter la sortie à votre convenance

1.4 Évaluation dans un modèle

Le but de cette partie du TP est d'utiliser `nltk` pour modéliser l'évaluation de formules logiques associées à des phrases du langage naturel dans un modèle.

Pour cela on va reprendre les grammaires qui ont été écrites lors des TP précédents afin d'avoir des phrases suffisamment complexes à traiter.

1.4.1 Définir un modèle et évaluer des formules

La première partie du TP consiste à spécifier manuellement un modèle et à évaluer les représentations associées à des formules du langage naturel dans ce modèle.

Pour construire un modèle dans `nltk` il est nécessaire de fournir un domaine et une fonction d'assignation.

- À partir du script `scriptEval-155.py`, élargissez le domaine et la fonction de valuation pour tester la validité de différentes phrases traitées par votre grammaire.
- En utilisant la grammaire qui gère le Cooper Storage, modifiez votre modèle pour que dans le cas d'une ambiguïté de portée entre quantificateur existentiel et universel, seule une des lectures soit vraie (avez-vous le choix pour décider quelle sera la lecture fausse ?)

1.4.2 Mettre à jour un modèle au fur et à mesure

Plutôt que de déclarer un modèle et tester la vérité de phrases à l'intérieur de celui-ci, on cherche maintenant à créer un modèle à partir d'assertions successives.

Pour cela on utilise *Prover9*, un outil externe à `nltk`, mais interfacé avec celui-ci. *Prover9* est un prouveur de théorème et offre une suite de fonctions permettant notamment de tester la consistance logique d'un ensemble de formules.

Par défaut le chemin vers *Prover9* n'est pas déclaré et donc pas connu de `nltk`. Pour y remédier, utilisez la commande suivante dans le terminal :

```
export PROVER9HOME=/home/003/applications/prover9/bin/
```

Modifiez maintenant votre script de la façon suivante :

- Créez une instance de parseur logique avec la commande `nltk.LogicParser()` (pour évaluer les formules logiques, il faut en fournir des versions correctement parsées, on utilise done un outil déjà présent dans `nltk`).
- Créez une instance de prouveur avec la commande `nltk.Prover9()`

Vous allez maintenant utilisez les fonctions de *Prover9* pour deux types d'applications :

1. Pour tester si la vérité d'une formule logique découle des assertions précédentes. Pour cela, au fur et à mesure que le script traduit les phrases en langue naturelle :
   - parsez les formules logiques au moyen de la méthode `parse()` du parseur logique
   - testez si la formule logique actuelle découle nécessairement des formules précédentes avec la méthode `prove(lp.parse(frml),assump)` du prouveur, où `frml` est la formule en cours et `assump` l'ensemble des formules déjà traitées (parsées par le parseur logique)
   - si la formule ne découle pas des précédentes, mettez à jour l'ensemble des hypothèses en y ajoutant la formule actuelle

2. Pour tester la possibilité de **créer** un modèle au fur et à mesure des assertions. Pour cela on utilise une instance d'objet *Mace4*, créée avec `nltk.Mace(5)`.
   - Au fur et à mesure que vous parsez vos phrases, essayez de créer un modèle qui satisfait toutes les formules logiques parsées jusque-là. Pour cela utilisez la méthode `build_model(None,frmls)` où `frmls` est la liste des formules logiques parsées par le parseur logique.

Note : il vous est possible d'utiliser l'interpréteur python pour réaliser un certain nombre d'opérations intéressantes qu'il est plus complexe d'insérer dans le script :

- Le parseur logique vous permet de parser n'importe quelle formule (y compris des lambda-termes)
- La commande `free()` vous permet de connaître les variables libres d'une formule
- Si `m` est un modèle déclaré explicitement (cf. section précédente) la commande `m.satisfiers(formule, 'x', g)` permet d'afficher les valeurs de `x` telles qu'elles satisfont la formule `formule` avec la fonction d'assignation `g` dans `m`. Cette commande est intéressante pour tester les modèles que vous mettez à jour, mais comme toutes les formules fournies par votre grammaire sont des formules fermées, vous ne pouvez l'utiliser dans le script.