Université Paris 7 Denis Diderot

Linguistique Informatique

Réalisation de la grammaire de Montague du français en interface syntaxico-sémantique sur ordinateur

par Alexandre Grebenkov

sous la direction de Pascal Amsili

Paris

2007

Résumé

Ce mémoire est consacré à la construction d'une grammaire du français au sein de la théorie de Richard Montague et à sa réalisation sur ordinateur. Le travail effectué est divisé en deux parties : la partie théorique, qui comprend l'étude de la problématique et le statut actuel des recherches dans ce domaine, et ensuite la construction d'une théorie du français inspirée des théories existantes (dont la majorité décrit la langue anglaise) et la partie pratique, qui consiste en la réalisation de la grammaire construite sur ordinateur.

Ces deux parties interfèrent considérablement. Nous nous sommes concentrés sur les points, qui se manifestent visiblement à l'intersection de deux approches : édification théorique et implémentation pratique. On espère, que cette motivation clarifiera, la raison pour laquelle l'accent a été mis sur certains points.

Pour indiquer les intérêts personnels de l'auteur, il faut dire que, à notre avis, il existe une prépondérance de l'édification théorique par rapport à l'utilisation pratique des grammaires proposées. En conséquence, au début du travail notre objectif était moins de construire une théorie usuelle, que de proposer une conception nouvelle d'implémentation de théorie linguistique.

Le mémoire se compose de trois sections principales : 1) introduction dans le problématique, description des idées de Montague ; 2) construction de la théorie pour la langue française à la base des théories existantes ; 3) réalisation de la grammaire dans l'interface syntaxico-sémantique sur ordinateur. Les annexes contiennent des informations exhaustives sur les formats, les syntaxes et les techniques utilisés dans l'implémentation de la grammaire sur ordinateur.

Plusieurs ouvrages ont été utilisés dans ce travail, mais les plus importants, qui ont apporté la contribution la plus importante sont « *Semantics in Generative Grammar* » de Kratzer & Heim, qui a déterminé la direction scientifique, et « *Representation and Inference for Natural Language* » de Blackburn & Bos, qui a donné beaucoup à penser sur l'implémentation pratique de la théorie.

Contenu

I.

La sémantique formelle

I.1. Introduction

Parmi les objets de recherches en linguistique, le *sens* occupe une place cruciale. Toute la problématique relative au sens est recouverte par le terme général de *sémantique*, qui pose de nombreuses questions. Qu'est-ce que le sens ? Comment décrire le sens ? A quelles formes linguistiques accorder du sens ? Comment lier le sens entre objets linguistiques et références externes ? Toutes ces questions conduisent au problème de la construction d'une théorie sémantique.

La tradition des différentes théories sémantiques compte des siècles d'Aristote à nos jours. Il existe plusieurs théories sémantiques en tout genre. Utilisons la classification générale de Barbara Partee :

1. La sémantique lexicale

   La sémantique qui s'occupe du sens des mots et la liaison entre le sens et les autres aspects. Parmi les chercheurs importants il y a Ch. Fillmore, I. Melchuk, Ju. Apresjan, D. Dowty.

2. La sémantique cognitive

   L'idée principale est dans l'étude des faits linguistiques du point de vue de l'interaction de la langue et de la cognition. C'est un domaine frontière entre la sémantique, l'épistémologie et la psychologie. On peut évoquer les noms principaux R. Jackendoff, G. Lakoff, L. Talmy.

3. La sémantique formelle

   Cette sémantique est née à la charnière de la logique, des mathématiques et de la linguistique. Son fondateur reconnu est Richard Montague. L'idée principale est de traiter de la langue naturelle comme un langage formel. D'autres chercheurs réputés sont D. Lewis, M. Cresswell, B. Partee.

4. La sémantique computationnelle

   Parmi les objectifs de cette direction de recherche figurent les aspects computationnels, les mises en œuvre informatiques, les problèmes pratiques, qui peuvent à leur tour attirer l'attention sur le développement de la théorie.

Le travail présenté ici se rattache au quatrième type de sémantique. Nous nous occuperons de l'implémentation informatique de la grammaire du français, ainsi que des aspects computationnels en pratique. Nous commencerons par la théorie de Montague pour nous introduire dans la problématique, qui est à l'origine de la grammaire du français proposée.

I.2. Idées principales

La sémantique formelle moderne est le résultat d'un double rapprochement : des linguistes d'une part et des logiciens d'autre part, qui, chacun suivant son parcours, ont finalement convergé. Montague était le premier à appliquer systématiquement les méthodes des logiciens de la syntaxe et de la sémantique formelles à la langue naturelle. On peut considérer la première phrase de son ouvrage « *English as a Formal Language* » comme un manifeste : «I reject the contention that an important theoretical difference exists between formal and natural languages».

Essayons d'énumérer les idées principales, qui toutes ensemble distinguent l'approche montagovienne des autres théories sémantiques (l'ordre d'apparition exprime notre vision de la logique interne de la théorie de Montague) :

1. Application de la notion de la vérité dans l'interprétation des énoncés.
2. Déclaration de la variété des mondes possibles, dans lesquels on interprète un énoncé.
3. Acceptation du Principe de Compositionnalité.
4. *Utilisation de types sémantiques (catégories d'après Montague)*
5. Uniformité structurale des règles syntaxiques et sémantiques.

6. Utilisation d'un langage formel pour décrire des phénomènes linguistiques :

6.1. Calcul des prédicats

6.2. Lambda-calcul

Ces points cruciaux ont le caractère d'être soit plus théoriques, soit plus techniques (les rubriques en italique), mais nous les énumérons tous ensemble, parce que leur unité détermine le repérage de la théorie de Montague. Il faut faire une remarque sur le dernier point *Lambda-calcul*. Formellement, Montague lui-même n'a pas utilisé lambda-calcul, mais on l'inscrit sur la liste des idées principales, car depuis longtemps le lambda-calcul est un formalisme standard en recherches de la sémantique computationnelle, conventionnellement accepté par majorité de sémanticiens.

Les grammaires qui traitent de la langue naturelle comme un langage formel et qui répondent dans une certaine mesure à ces conditions sont conventionnellement appelées *les grammaires de Montague*.

Nous allons examiner toutes ces idées en bref dans ce chapitre, sauf lambda-calcul qu'on introduit plus tard, quand on en aura besoin directement.

La notion de vérité

Ce postulat est probablement le plus important dans la théorie de Montague. La notion de la *vérité* a décalé les priorités dans la théorie sémantique. Elle est entrée dans la sémantique comme un critère principal. Selon ce critère, la partie essentielle de l'interprétation sémantique d'une certaine phrase consiste en la spécification *des conditions, dans lesquelles la phase est vraie ou fausse*.

On peut évoquer D. Dowty, R. Wall et S. Peter « *Introduction to Montague Semantics* » : comprendre une phrase, c'est « savoir ce à quoi le monde doit ressembler pour qu'elle soit vraie » ; en d'autres termes, c'est connaître ses *conditions de vérité* (de là vient l'appellation des théories – *sémantique véri-conditionnelle*). Ainsi donc, la partie essentielle de la sémantique consiste en une

construction de la théorie des conditions de vérité pour une langue, ce que Montague fait dans sa grammaire de l'anglais.

Les valeurs de la vérité sont traditionnellement définies comme l'ensemble $D_t := \{0, 1\}$. Alors chaque phrase devient une formule, chaque formule renvoie la valeur de la vérité selon les conditions de vérité, c'est-à-dire selon l'état des affaires dans le monde (dans le modèle), qui est reflété dans la formule par des variables, des prédicats etc.

Evidemment, l'introduction de la notion de la vérité a un rôle crucial dans la théorie linguistique, ce qui pose plusieurs questions sur l'adéquation de une telle théorie : est-ce qu'elle peut apporter quelque chose à la connaissance du système langagier, puisque le linguiste n'a pas à se préoccuper de ce qui est vrai ou de ce qui est faux, ces problèmes relevant davantage de considérations philosophiques que de considérations linguistiques.

On peut trouver la réponse partielle dans l'ouvrage de Chomsky « *Syntactic structures* » (1957). Il s'agit notamment de la dichotomie très connue: *performance / competence*. Dans le cas de la théorie de Montague, on essaie de modéliser la compétence comme un système abstrait destiné à rendre compte de certaines des facultés de l'apprentissage et de la maîtrise d'une langue. Dans ce cadre, il semble que la notion de vérité trouve naturellement sa place.

Et dans cette perspective, la définition de la compétence sémantique proposée par M. J. Cresswell est la suivante : « Ce que j'avancerai comme définition de la compétence sémantique d'un individu n'est ni plus ni moins que sa capacité à dire devant une phrase relative à une situation, si cette phrase, dans cette situation, est vraie ou fausse ».

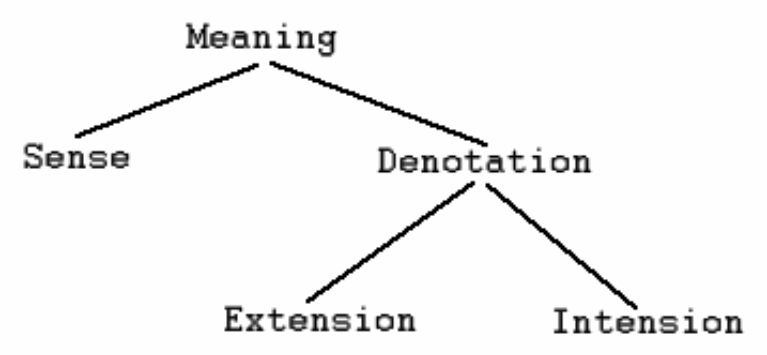
« Sémantique des mondes possibles »

Dans ses recherches Montague a soulevé la question de l'existence d'autres modèles (ou d'autres mondes), dans lesquels le sens d'un énoncé pourrait se distinguer.

Il y a des exemples classiques comme : *The king of the France* (dont la référence n'existe pas) ou des relations entre *Morgenstern*, *Abendstern* et *Venus* (qui tous les trois dénotent la même chose).

Selon Montague la sémantique doit savoir décrire les situations potentielles, imaginées par des hommes, même quand la langue ne peut pas se référer à des états de choses existants. Ce qui est nouveau est que la théorie linguistique (chaque théorie sémantique est une espèce de théorie linguistique) doit prendre en compte les relations que la langue entretient avec *ce qui n'est pas la langue,* c'est-à-dire les individus, les choses, les états de choses, les situations tout ce qu'on peut appeler le *monde*.

La structure du sens devient encore plus compliquée. Ronnie Cann propose la structure suivante :



Nous nous intéressons à la dénotation, alors que le *sense* est interne relativement à linguistique, il définit les relations entre les mots et les expressions linguistiques, sans toucher le monde externe. L'extension est un dénotât d'une expression, une référence transparente dans le contexte, qui n'implique pas d'ambiguïté, à laquelle on peut appliquer la loi de Leibniz (qui permet de substituer les expressions extensionnellement équivalentes dans les formules sans changer la valeur de la vérité). L'intension d'une phrase est chargée de déterminer la valeur de la vérité pour chaque monde possible. Elle parcourt tous les mondes et renvoie à ceux, pour lesquels elle est vraie.

Montague a proposé la logique intensionnelle pour traiter des intensions. Cette approche ressemble à l'introduction de multidimensionnalité dans la sémantique. L'intension d'une phrase est une fonction des mondes possibles vers

des valeurs de vérité. En simplifiant beaucoup, nous pouvons dire que chaque objet (soit une expression, soit un dénotât) logique reçoit un indice, qui attribue cet objet au monde correspondant, par exemple : $[[\alpha]]^M$ est une dénotation de α dans le monde M. Après avoir construit les systèmes « parallèles », on définit des règles de traduction. Son travail « *Universal Grammar* » (1970) est partiellement consacré à cette problématique. La sémantique de cette direction a été ensuite appelée « la sémantique des mondes possibles ». Dans une sémantique des mondes possibles, une expression linguistique est toujours interprétée en relation à un monde possible.

Les cas intensionnels sont hors des intérêts de ce travail, on travaillera sur les références transparentes, c'est-à-dire les extensions.

Le Principe de Compositionnalité

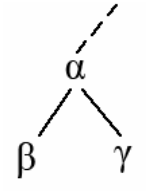
Ce principe, que l'on attribue à Frege (et même parfois il est appelé le Principe de Frege), s'énonce, en général, de la manière suivante : « *le sens du tout est une fonction du sens des parties et de leur mode de combinaison* » (formulation d'après Michel Galmiche « *Sémantique linguistique et logique* »).

La formalisation du Principe est citée par « *Stanford Encyclopedia of Philosophy* » (en traduction) :

« Soit F une fonction syntaxique de k arguments, définie sur E (l'ensemble des expressions syntaxiques), soit m une fonction, qui fait l'attribution de dénotations à des expressions syntaxiques.

Donc m est compositionnelle relativement à F, s'il existe une telle fonction partielle G, définie sur M (l'ensemble des dénotations possibles), telle que pour chaque $F(e_1,\ldots,e_k)$ défini on a $m(F(e_1,\ldots,e_k)) = G(m(e_1),\ldots,m(e_k))$. »

La conséquence la plus importante de ce Principe est la *localité* de toute analyse. Pour illustrer ce principe, prenons la structure la plus simple $\alpha \rightarrow \beta\ \gamma$:



Le Principe de Compositionnalité nous dit que le construction et l'interprétation du sens de α ne dépend que de constituants β et γ : $[[\alpha]] = [[\beta]] \times [[\gamma]]$ et toute la question est dans ce qui est derrière le

symbole « × » (le mode de combinaison). Toute la structure dehors de cette partie ne participe pas à l'analyse de cette partie-là.

Ainsi donc, la théorie sémantique doit non seulement attribuer le sens à une expression, mais aussi expliquer comment les parties contribuent au tout.

La question de combinaison de constituants nous amène à l'opération cruciale, qui assure le Principe de Compositionnalité, l'*application fonctionnelle*, dont on parlera après l'introduction de types sémantiques.

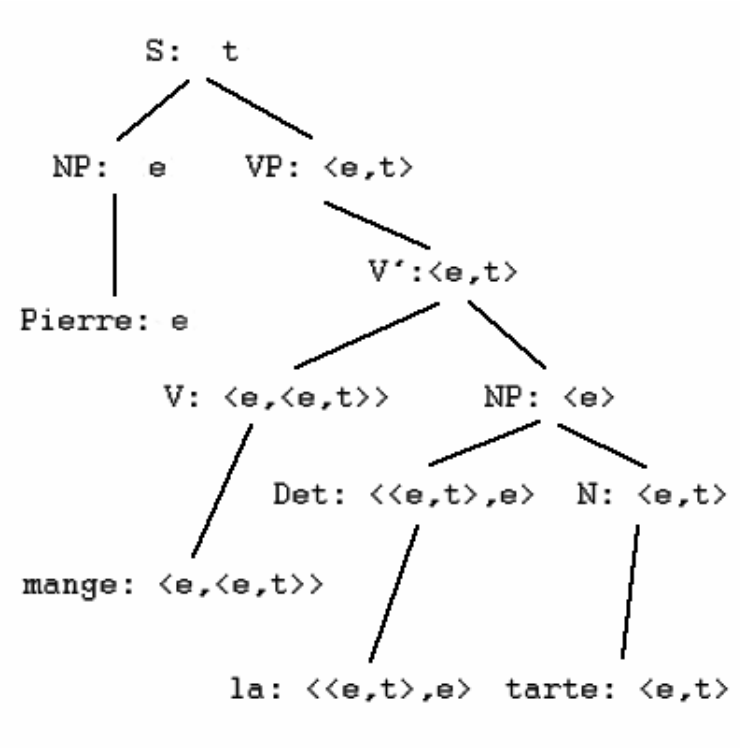
Types sémantiques

Les *catégories* (ensuite les *types*) ont été introduites par Montague pour réaliser le Principe de Compositionnalité. La structure syntaxique était insuffisante pour garantir l'interprétation correcte de la combinaison des constituants immédiats.

L'idée principale est d'attribuer un type à chaque nœud de la structure d'une expression. A chaque catégorie d'éléments lexicaux est assigné un type. Dans ce cas-là, au début de l'analyse, seuls les nœuds terminaux possèdent un type, mais dans l'analyse d'une phrase les types s'attribuent aux nœuds intermédiaires par l'application fonctionnelle, c'est-à-dire leurs types sont composés à condition toutefois que les types soient compatibles (règles de typage).

Prenons comme exemple la phrase : *Pierre mange la tarte*.

L'arbre syntaxique décoré par le système de typage de Montague est le suivant :



Cette organisation permet d'effectuer l'application fonctionnelle et d'éliminer les structures incompatibles.

Formellement, les types sont définis de la manière suivante :

Types sémantiques

1) `e` et `t` sont des types sémantiques
2) Si `a` et `b` sont des types sémantiques, alors `<a,b>` est aussi un type sémantique
3) Rien d'autre n'est un type sémantique

Domaines des dénotations sémantiques

1) D_e := D (l'ensemble des individus)
2) D_t := {0, 1} (l'ensemble des valeurs de la vérité)
3) Pour tous les types sémantiques `a` et `b`, $D_{<a,b>}$ est l'ensemble de toutes les fonctions de D_a vers D_b.

Cette définition est récursive, elle « génère » un ensemble infini de types, dont seulement une petite partie est utilisée en description d'une langue.

En fait, Montague a utilisé une notation légèrement différente, mais l'idée reste la même.

Uniformité structurale des règles syntaxiques et sémantiques

Cette thèse est liée au Principe de Compositionnalité, qui caractérise un certain type de liens entre la syntaxe et la sémantique des langues naturelles. Le commentaire de D. Dowty, R. Wall et S. Peter « *Introduction to Montague Semantics* » donne une vision assez claire des implications essentielles de ce principe :

« Formulé de cette manière, il semble qu'il s'agisse d'un truisme, mais en fait, dès qu'il est envisagé sérieusement à l'intérieur du cadre de la sémantique véri-conditionnelle, il implique quelques contraintes relativement sévères sur les systèmes de règles syntaxiques et sémantiques que l'on peut construire. A noter, par exemple, que les "parties" dont il est question dans cette formulation du Principe de Frege doivent être des constituants syntaxiques de l'expression concernée. De plus, les sens de ces constituants doivent entrer dans le sens de l'expression entière, *d'une manière fixe,* déterminée une fois pour toutes par la règle sémantique correspondant à la règle syntaxique au moyen de laquelle les constituants ont été assemblés. »

Evidemment, ce principe impose des restrictions très rigoureuses sur l'organisation du système, particulièrement, sur la liaison entre la syntaxe et la sémantique :

« La réalisation de cette compositionnalité parfaite implique quelque sacrifice, entre autres en ce qui concerne le caractère plausible des analyses syntaxiques. Mais Montague était souverainement indifférent aux critiques des chomskyens, pour qui une analyse syntaxique doit être motivée par des considérations purement syntaxiques et ne pas être seulement fonctionnelle à l'analyse sémantique: il déclarait « ne pas trouver grand intérêt à la syntaxe, sinon comme prélude à la sémantique » (Diego Marconi, « *La philosophie du langage au XXème siècle* »).

Idéalement, nous souhaitons avoir une correspondance univoque entre la syntaxe et la sémantique. En premier lieu cela concerne l'analyse des constituants, c'est-à-dire qu'une opération au niveau syntaxique devrait être répétée au niveau

sémantique. Ce postulat est connu sous le nom de *hypothèse de correspondance règle par règle* (Rule-to-Rule hypothesis, traduit d'après Ronnie Cann « *Formal Semantics* ») :

Pour chaque règle syntaxique il existe une règle sémantique correspondante. Bien qu'il soit souhaitable de s'en tenir à ce postulat pour réserver l'organisation transparente de la grammaire, cette condition n'est pas nécessaire. Il existe des solutions de quantification, qui violent cette uniformité, par exemple*, Cooper storage* ou *Keller storage*.

Il s'agit des phrases comme *Every man loves a woman*, qui a deux interprétations sémantiques : 1) pour chaque homme il existe quelque femme qu'il aime ; 2) il existe une seule femme qui est aimée par tous les hommes. L'ambiguïté est apparue à cause deux quantificateurs *every* et *a*.

Cooper storage et *Keller storage* sont les dispositifs structurels, qui permet de décaler la résolution finale de l'ambiguïté jusqu'à l'obtention de tous éléments nécessaires. C'est-à-dire, que certaine information locale est sauvegardée pendant toute l'analyse, et la décision se prend au nœud dernier. Généralement, ce transfert de l'information viole l'idée de l'uniformité syntaxico-sémantique.

Calcul des prédicats

Le langage basique des formalismes construits au sein de l'approche de Montague est le calcul des prédicats, qui est utilisé universellement des mathématiques vers la linguistique.

Les éléments de ce langage :

1. Les opérateurs logiques ("!" – négation, "&" – conjonction, "|" – disjonction, ">" – implication),
2. Les quantificateurs (∀ , ∃).
3. Les symboles techniques (",", "(", ")" etc.).
4. L'ensemble des variables : x, y, z etc.
5. L'ensemble des constantes : a, b, c etc.
6. L'ensemble des prédicats : A, B, C etc.

Nous allons définir les *formules atomiques*.

Si P est un prédicat de n places et $m_1, \ldots, m_n$ sont des variables ou constantes, alors $P(m_1, \ldots, m_n)$ est *une formule atomique*.

Maintenant, il faut définir les formules bien formées (*well formed formules* ou *wffs*), c'est-à-dire acceptées dans le langage.

1. Toutes les formules atomiques sont des formules bien formées.
2. Si φ et ψ sont des formules bien formées, alors !φ, (φ>ψ), (φ&ψ), (φ|ψ) sont des formules bien formées.
3. Si φ est une formule bien formée et x est un variable, alors ∃xφ et ∀xφ sont des formules bien formées.
4. Rien d'autre n'est une formule bien formée.

Les formules atomiques correspondent aux phrases « basiques » d'une langue naturelle (*clause*).

Pour illustrer la mécanique, soit l'ensemble de constantes = {Pierre, Marie}, les prédicats AIMER(x,y), x étant sujet et y étant objet, HEUREUX(x), GARCON(x), nous pouvons alors décrire les phrases suivantes :

Pierre aime Marie.

```
AIMER(Pierre, Marie)
```

Chaque garçon aime Marie.

```
∀x(GARCON(x)>AIMER(x, Marie))
```

Marie n'aime pas Pierre.

```
!AIMER(Marie, Pierre)
```

Chaque garçon, que Marie aime, est heureux.

```
∀x(GARCON(x)&AIMER(Marie, x)>HEUREUX(x))
```

Une petite remarque sur la notation : pour lier une phrase de la langue naturelle à sa formule, nous emploierons les doubles crochets « [[ ]] ». Pour ne pas utiliser le mot *sens*, qui est très complexe, nous prenons le mot *dénotation*. Donc l'expression [[α]] est une dénotation de α. Nous allons établir les égalités comme ceci :

[[Pierre aime Marie]] = `AIMER(Pierre, Marie)`

Normalement, les langages plus complexes sont élaborés à partir du calcul des prédicats. Nous allons examiner et utiliser le λ-calcul dans les sections suivantes, car il est très commode pour décrire des applications fonctionnelles.

I.3. L'anglais d'après Montague

Montague a démontré ces thèses dans la grammaire consécutive d'une partie de l'anglais, présentée dans son article « *The Proper Treatment of Quantification in Ordinary English* » (1970).

Nous allons exposer l'organisation de sa grammaire et décrire les moments principaux de son traitement de l'anglais.

Catégories

Le lexique est divisé en catégories sémantiques, exprimées directement par CAT, et en catégories syntaxiques attribuées selon les catégories sémantiques. Elles sont toutes réunies dans le tableau emprunté à Barbara Partee « *Some Transformational Extensions of Montague Grammar* ».

Catégorie (CAT)	Abréviation	Nom d'après Montague	Equivalent linguistique
t	*base*	truth-value expression, or declarative sentences	Sentence
e	*base*	entity or individual expression	(noun phrase)
t/e	*IV*	intransitive verb phrase	verb phrase
t/IV	*T*	Term	noun phrase
IV/T	*TV*	transitive verb phrase	transitive verb
IV/IV	*IAV*	IV	VP-adverb
t//e	*CN*	common noun phrase	noun
t/t	*none*	sentence-modifying adverb	sentence-modifying adverb
IAT/T	*none*	IAV-making preposition	Locative, preposition
IV/t	*none*	sentence-taking verb phrase	V which takes that-COMP
IV/IV	*none*	IV-taking verb phrase	V which takes infinitive COMP

L'utilisation d'une simple ou double barre oblique distingue les catégories syntaxiques différentes si les CAT sont identiques (« intransitive verb phrase » et « common noun phrase », par exemple).

Lexique

Il y a 9 sous-classes lexicales :

B_{IV} = {run, walk, talk, rise, change}

B_T = {John, Mary, Bill, ninety, he_0, he_1, he_2, …}

B_{TV} = {find, lose, eat, love, date, be, seek, conceive}

B_{IAV} = {rapidly, slowly, voluntarily, allegedly}

B_{CN} = {man, woman, pen, unicorn, price, postman, temperature}

$B_{t/t}$ = {necessarily}

$B_{IAV/T}$ = {in, about}

$B_{IV/t}$ = {believe that, assert that}

$B_{IV/IV}$ = {try to, wish to}

B_A = Ø = "l'ensemble vide" pour chaque catégorie A autre que les catégories mentionnées.

P_A – l'ensemble de phrases de la catégorie A (comme P_{CN} ou P_{TV}).

Règles de la grammaire

Il y a trois étapes de la grammaire, à chacune desquelles les règles sont définies : règles syntaxiques, règles de traduction de la syntaxe vers la logique intensionnelle, règles d'interprétation de la sémantique des mondes possibles pour la logique intensionnelle.

Le composant syntaxique de la grammaire est défini par l'entrée récursive simultanée des ensembles de P_A pour chaque A à l'intérieur de l'ensemble de P_t. Chaque règle syntaxique est généralement formulée de la manière suivante :

Si $\alpha_1 \in P_{A1}$, $\alpha_1 \in P_{A2}$, …, $\alpha_n \in P_{An}$, alors $F_i(\alpha_1,\ldots, \alpha_n) \in P_B$

ou F_i est une spécification du mode de combinaison syntaxique des constituants et B est la catégorie de la phrase de résultat.

Donc, pour chaque catégorie A/B ou A//B il y a une règle :

Si $\alpha_1 \in P_{A/B}$ et $\beta \in P_B$, alors $F(\alpha, \beta) \in P_A$

Dans cette grammaire Montague a proposé 17 règles syntaxiques.

Une règle de traduction vers la logique intensionnelle a été mise en correspondance à chaque règle syntaxique. L'uniformité de règles est particulière à Montague. Même dans la désignation de règles il met en relief l'uniformité structurelle.

Le dernier jeu de règles sont les interprétations de la logique vers l'anglais, au nombre de neuf règles pour chaque sous-classe lexicale.

En faisant le total, il faut dire que l'exposition de la grammaire est donnée par Montague d'une façon très précise et formelle. Cette approche a été soumise à la critique comme trop abstraite et éloignée, parce que il n'était pas évident de voir les phénomènes de la linguistique traditionnelle derrière les théorèmes de Montague.

Exemples et problèmes classiques

A la fin de l'article Montague examine certains exemples linguistiques, qui peuvent clarifier sa grammaire, au premier abord, très éloignée des faits réels.

Les exemples les plus intéressants touchent le problème de la quantification et de l'opposition de l'intensionnalité et de l'extensionnalité. La notation que nous utilisons est légèrement modifiée de celle de Montague pour se conformer à la notation standard.

Les cas simples extensionnels :

[[*Bill walks*]] = walks(Bill)

[[*Every man walks*]] = $\forall$x(man(x)$\rightarrow$ walks(x))

[[*John finds a unicorn*]] = $\forall$x(unicorn(x)&find(John,x))

Mais si l'on prend un exemple avec le verbe intensionnel *to seek* :

John seeks a unicorn

on reçoit deux interprétations en logique, le premier étant :

[[*John seeks a unicorn*]] = $\exists$x(unicorn(x)&seek(John,x))

mais cette interprétation nous dit qu'il *existe* une licorne, ce qui n'est pas prouvé et le sens de l'expression peut être mieux reformulé par la phrase *John tries to find a unicorn.* Pour refléter cette intension de *John seeks a unicorn*, il faut indiquer que John cherche les propriétés intensionnelles de l'existence de licorne, ce qui est reflété par la formule :

[[*John seeks a unicorn*]] = seek(^John, ^P(∃x(unicorn(x)&P(^x))))

où des chapeaux descendant signifient l'intensionnalité

et des chapeaux ascendant signifient l'extensionnalité.

Si paraphraser *John seeks a unicorn* comme *John tries to find a unicorn*, Montague remarque, que le verbe *try to* est un verbe de deuxième ordre (c'est-à-dire qu'il peut prendre comme argument un autre verbe), et on obtient deux interprétations :

[[*John tries to find a unicorn*]] = try(^John, ^y∃x(unicorn(x)&find(ˇy, x)))

[[*John tries to find a unicorn*]] = ∃x(unicorn(x)&try(^John, ^y find(ˇy,x)))

Dans ce cas-là, l'ambiguïté dépend de la portée de quantificateur devant *unicorn*.

Encore un exemple classique. Prenons deux phrases :

The postman is Bill

The postman is running

la conclusion (l'implication) est *Bill is running.*

Mais la construction pareille ne marche pas :

The temperature is ninety

The temperature is rising

alors on obtiendrait *Ninety is rising* par la logique standard, ce qui n'est pas nécessairement vrai. La réponse à ce paradoxe est cachée dans l'essence de l'intensionnalité : « The sentence *The temperature is ninety* asserts the identity not of two individual concepts but only of their extensions. »

Ambiguïté à cause de deux quantifications dans l'expression *A woman loves every man* :

[[*A woman loves every man*]] = ∃x(woman(x)&∀y(man(y)>love(x,y)))

Il existe une seule femme, qui aime chaque homme.

[[*A woman loves every man*]] = $\forall y(man(y)>\exists x(woman(x)\&love(x,y)))$

Pour chaque homme il existe quelque femme, qui l'aime.

Certains de ces exemples seront traités dans les grammaires, qu'on va étudier dans le chapitre suivant.

I.4. Evaluation des grammaires

L'approche de Montague a influencé le développement de plusieurs théories sémantiques, qui se distinguent l'une de l'autre par certains aspects. Une question légitime se pose : comment estimer ces grammaires ?

Cette question est commune pour toute la linguistique, car en ce domaine, malheureusement, on manque de critères perceptibles. Les théories linguistiques sont rarement utilisées dans *engineering* commercial. Ce qu'on appelle TAL généralement n'a que peu à voir avec les théories linguistiques, ce sont des méthodes et techniques auxiliaires et secondaires par rapport à la théorie linguistique.

Evidemment, ce n'est pas possible de mesurer les notions souvent cognitives par leur application pratique, ce serait incorrect. Dans chaque domaine il existe une distinction entre la science et la pratique : physique théorique et physique appliquée, mathématique abstraite et mathématique appliquée. Mais en linguistique, la distinction est très forte. Chaque théorie physique trouve son domaine d'application, probablement, pas tout d'un coup, mais l'évolution permet d'éliminer les théories injustifiées (l'exemple classique est tout le jeu de théories expliquant la structure d'un atome au début du XXème siècle, après lequel il nous est resté le modèle planétaire de Rutherford).

A présent en linguistique il existe, à notre avis, une distance très sérieuse entre la théorie et la pratique. La linguistique n'est pas « palpable » et presque toutes les théories fonctionnent au niveau abstrait.

Essayons de formuler certains critères pour nous orienter dans l'estimation des théories :

1. Combien de phénomènes est-ce que la théorie couvre ? L'objectif idéal de la linguistique comme de toute autre science est de proposer l'image complète et universelle de l'objet en considération – la langue. Mais l'objectif réel est, évidemment, de trouver certaine balance entre le « profondeur » et le « largeur » d'une théorie.

2. Elégance de la théorie : la longueur d'une règle pour décrire un phénomène F (proposé par Chomsky).

3. Simplicité de la théorie: le nombre de règles pour décrire un phénomène F (proposé par Chomsky).

4. Possibilité de construire l'algorithme de calcul.

5. Simplicité de réalisation de l'algorithme de calcul.

6. Relevance cognitive : est-ce que la théorie peut expliquer certaines régularités générales (les universaux), qui concernent les questions de l'apprentissage des langues et de la conscience.

7. Adaptabilité aux nouveautés: est-ce que la théorie peut incorporer les phénomènes nouveaux et s'adapter aux tâche nouvelles.

Nous en tiendrons note de ces propriétés d'une théorie linguistique pendant notre travail.

Pour terminer la première partie, nous faisons une petite remarque. Les mots « théorie » et « grammaire » abondent dans ce mémoire. Ces mots sont les notions du plus haut niveau d'abstraction, en conséquence, cela ne vaut pas la peine de les définir. Traditionnellement, les théories sont perçues comme plus générale et les grammaires – comme plus locales. Nous nous en tenons au même avis, en rangeant les idées plus générales dans le domaine de la théorie et les idées plus concrètes dans le domaine de la grammaire. Mais si on laisse cette subtilité, ces deux mots sont mutuellement échangeables.

II.

Grammaires de Montague

A la fin de cette partie on présentera la grammaire du français dans le cadre de l'approche de Montague, qui a été implémentée sur ordinateur dans l'interface syntaxico-sémantique GANSS. On montera toutes les étapes de construction de cette grammaire, qui a évidemment absorbé plusieurs traits des grammaires déjà existantes. Avant sa présentation, deux ouvrages seront attentivement étudiés, parce qu'ils ont considérablement influencé notre travail :

1. Le travail de Blackburn & Bos pour l'anglais (1999)
2. La théorie de Kratzer & Heim pour l'anglais (1997)

Car ces deux grammaires traitent de la même problématique et utilisent à bien des égards les mêmes méthodes ; les particularités les plus intéressantes seront présentées, sans répéter les généralités.

II.1. Le travail de Blackburn & Bos

L'ouvrage de Blackburn & Bos « *Representation and Inference for Natural Language* » (1999) est particulièrement intéressant, car il contient la théorie et son implémentation en même temps. Ce n'est pas par un effet du hasard, si cette section est titre « le travail » contrairement à la théorie de Kratzer & Heim. Ce n'est pas une théorie dans le sens pur. Par la composition du livre on voit que les auteurs se sont plutôt concentrés sur la présentation pratique des techniques fondamentales, qui peuvent être utilisées dans la réalisation d'une théorie sémantique sur ordinateur. Prolog a été choisi comme langage de réalisation.

Blackburn & Bos préviennent dans l'introduction de la manière dont ils comprennent le domaine de leur recherches :

« Computational semantics is a relatively new subject, and trying to define such a lively area (if indeed it is a single area) seems premature, even counterproductive. However, in this book we take 'semantics' to mean 'formal

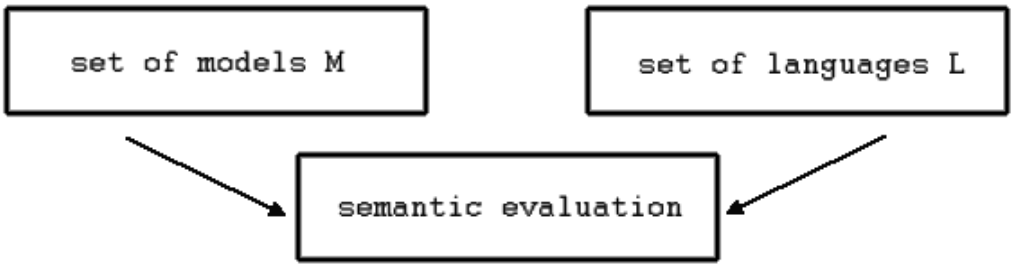
semantics' and 'computational semantics' to be the business of using a computer to actually build such representations *(semantic construction)* and reason with the result *(inference)*. »

Notre intérêt se limitera à la première partie « *semantic construction* ». Les inférences logiques sont un autre domaine, auquel nous ne toucherons pas dans ce travail. Il faut remarquer que le problème de déductions s'intègre bien dans le paradigme du Prolog (on peut même dire que Prolog est lui-même un système de déductions), mais la résolution de déductions par d'autres moyens augmente sérieusement la complexité du problème. En outre, certaines techniques connues sont implémentées, par exemple, *storage methodes* (Cooper storage, Keller storage). Le deuxième volume est consacré à l'implémentation en Prolog de la conception DRS (*Discourse Representation Structures*), qui est aussi hors de notre considération.

L'objectif de cette section est de donner une vision claire de la grammaire Blackburn & Bos pour ensuite emprunter les solutions réussies dans notre travail pratique.

Modèles, langages, évaluation sémantique

La pierre angulaire de toute la construction est la corrélation de trois parties : modèles, langages et bloc d'évaluation sémantique, lequel décide si le langage choisi satisfait à un certain modèle :



Le « modèle » correspond au monde dans le sens de Montague, c'est-à-dire une certaine situation fixée prise en considération. Le « langage » est l'ensemble de formules, qui expriment certains faits. Sans répéter l'essence du calcul de prédicats, disons, que ce formalisme est basique pour Blackburn & Bos :

« Many semanticists claim that intermediate levels of logical representation are essentially redundant. … Logical representations – that is, formulas of a logical language – encapsulate meaning in a clean compact way. »

L'évaluation sémantique consiste en la confrontation des assertions exprimées par les formules avec l'état des affaires. On comprend que c'est une reformulation du test sur la vérité d'une phrase. Ce bloc est un mécanisme de vérification.

Prenons un exemple de Blackburn & Bos. Le modèle simple :

Soit M_1 un modèle : $M_1 = <D, F>$, ou D étant le domaine et F une fonction interprétative, qui définit les valeurs sémantiques.

Le domaine $D = \{d_1, d_3, d_3, d_4\}$ est le vocabulaire en M_1.

La fonction F est définie de la manière suivante :

$F(Mia) = d_1$	*Mia est d_1*
$F(Honey\text{-}Bunny) = d_2$	*Honey-Bunny est d_2*
$F(Vincent) = d_3$	*Vincent est d_3*
$F(Pumpkin) = d_4$	*Pumpkin est d_4*
$F(customer) = \{d_1, d_2, d_3, d_4\}$	*Mia et Vincent sont des consommateurs*
$F(robber) = \{d_2, d_4\}$	*Honey-Bunny et Pumpkin sont des voleurs*
$F(love) = \{(d_4, d_2), (d_3, d_1)\}$	*Vincent aime Mia,*
	Pumpkin aime Honey-Bunny

Comme le modèle M_1 est très simple, il n'y a pas grande chose à exprimer et interroger. Mais pour illustrer le fonctionnement, prenons un langage L_1, qui comprend trois formules :

```
∀x(robber(x)>customer(x))
∃x(customer(x)>robber(x))
!∃x(love(x,Vincent))
```

et un langage L_2 de deux formules :

```
(love(Pumpkin,Honey-Bunny))
∃x(love(Mia,x))
```

On voit que L_1 est satisfait en M_1 (chaque voleur est un consommateur – Pumpkin et Honey-Bunny; il existe un consommateur, qui est un voleur en même temps – Pumpkin et Honey-Bunny et personne n'aime Vincent), car chaque formule est *vraie* dans ces *conditions de la vérité*. L_2 échoue selon notre modèle M_1, le problème surgit à cause de Mia, qui n'aime personne, il n'existe pas tel x, que Mia aime, donc on a la contradiction, L_2 est faux. Mais si l'on change notre modèle à $M_2 = M_1$ `U {F(love) = (Mia,Marsellus)}`, dans ce cas-là L_2 devient valable par rapport à M_2.

Pour définir formellement l'évaluation sémantique, les auteurs introduisent la notion de « *satisfaction* », étant une relation, qui s'établit entre la formule, le modèle et l'*assignement de valeurs aux variables.* L'assignement de valeurs aux variables en M = <D, F> est une fonction *g* de l'ensemble de variables vers D. En laissant de côté certains détails formels, on donne l'idée générale de la définition :

M, g |= ψ (ψ est satisfait en M avec l'assignement g) ssi

M, g |= $R(t_1,\ldots, t_n)$	ssi $(I(t_1),\ldots, I(t_n))\in F(R)$, où I est une interprétation
M, g |= !ψ	ssi not M, g |= ψ
M, g |= ψ & φ	ssi M, g |= ψ and M, g |= φ
M, g |= ψ | φ	ssi M, g |= ψ or M, g |= φ
M, g |= ψ > φ	ssi not M, g |= ψ or M, g |= φ
M, g |= ∃xψ	ssi M, g' |= ψ, pour certaine variante x g' de g
M, g |= ∀xψ	ssi M, g' |= ψ, pour toute variante x g' de g

Comme nous l'avons dit, la particularité du travail de Blackburn & Bos est que ces constructions théoriques trouvent leur réalisation en Prolog.

Nous ne donnons pas ici les programmes en Prolog (sauf quelques exemples des règles), parce qu'ils sont difficiles à lire sans préparation, et en outre, ils sont assez longs, mais si on les réduit, la logique se perd. Le livre contient toutes les explications nécessaires, il faut s'y référer pour les détails. En simplifiant beaucoup, on peut dire que chaque objet de la théorie se transforme en une variable ou une constante, chaque énumération se transforme en une liste, chaque règle se

transforme en un théorème. Tous ensemble composent un programme en Prolog, qui peut répondre aux questions sur le modèle ou peut déduire les inférences dans le sens du Prolog, ayant les théorèmes pour un point de départ. Evidemment, cette explication ne compte pas plusieurs questions techniques (input, output, définition des opérateurs, définition des nombreuses procédures), qui généralement constituent une partie importante du travail pratique.

« Grammar engineering »

Les auteurs utilisent la conception qui nous plaît beaucoup de « grammar engineering », qui présuppose, qu'on construit la théorie au fur et à mesure, en passant à l'étape suivante seulement après avoir donné la réalisation du phénomène considéré en Prolog.

Cette conception implique la développement conséquent de chaque solution choisie, parce que chaque changement dans les principes coûte cher dans le changement de l'implémentation : « As we have learned (often the hard way) incorporating these changes and keeping track of what is going on, requires a disciplined approach towards grammar design. »

Les auteurs soulignent, que « grammar engineering principles have strongly influenced the design of our grammars—and *not,* we would like to stress, purely for pedagogic reasons. »

Cet ouvrage est fondé sur le cours d'études de *Computational Linguistics*, et l'exposé même du matériau diffère avantageusement des théories, probablement, plus vastes dans les faits couverts, mais moins clarifiées et argumentées.

Blackburn & Bos décrivent leur grammaire dans la manière suivante :

« We should strive for a grammar that is *modular* (each component should have a clear role to play and a clean interface with the other components), *extendible* (it should be straightforward to enrich the grammar should the need arise) and *reusable* (we should be able to reuse a significant portion of the grammar even when we change the underlying representation language). »

Ces généralités donnent la bonne direction de l'élaboration d'une grammaire, il faut les respecter, si on travaille sur la partie pratique, sur l'implémentation d'une grammaire. Dans notre travail, nous nous efforcerons d'utiliser la même méthodologie et les mêmes principes d'organisation de la grammaire.

Architecture de la grammaire

Pour relater l'architecture de la grammaire de Blackburn & Bos, utilisons encore une fois la description d'auteur :

« We have adopted a fairly simple three-level grammar architecture consisting of a collection of semantically annotated *rules,* a *lexicon,* and a set of *semantic macros.* The rules are DCG rules annotated with an extra slot which applies semantic functions to arguments using the `@` operator. The lexicon lists information about words belonging to most syntactic categories in an easily extractable form; again, this component will stay fixed throughout the book. Finally, we have the crucial semantic macros. This is a level at which we state what we have previously called 'lexical entries'. It is here that we will do most of our semantic work, and our modifications will largely be confined to this level. »

Nous allons étudier succinctement chaque composant de la grammaire.

« Rules »

Il y a deux types de règles : syntaxiques – pour définir la structure et lexiques – pour obtenir l'information sur les nœuds terminaux.

Les règles syntaxiques sont de type DCG (*Definite Clause Grammars*, c'est un mécanisme intégré dans Prolog pour traiter des *grammaires hors-contexte*) :

```
s --> np, vp
np --> np, coord, np
noun --> noun, coord, noun
Vp --> mod, vbar(inf)
```

etc. La liste intégrale est dans l'annexe du livre de Blackburn & Bos.

Il y a des défauts standard pour la grammaire syntaxique de ce genre :

1. Un cercle limité de faits linguistiques est couvert.
2. La morphologie est abandonnée. Tous les verbes sont en présent, singulier, 3e personne.

Ces défauts sont communs pour toutes les grammaires de ce type (ce sont les règle du jeu), quand même cette grammaire est utile pour les buts principaux :

« But for all its shortcomings, this small set of rules assigns tree structures to an interesting range of English sentences or small discourses. »

Le deuxième type des règles est des règles lexicales, qui s'appliquent aux nœuds terminaux (les chaînes symboliques de l'input de parseur). Elles sont de la forme suivante :

```
noun(Noun) -->
   {
      lexicon(noun,Sym,Phrase,_) – appel au dictionnaire
      nounSem(Sym,Noun) – la représentation sémantique
   },
   Phrase
```

De telles règles existent pour chaque catégorie. Leur implémentation en Prolog est moins importante, que leur but, qui est d'attribuer la catégorie syntaxique à chaque mot dans l'input, si ce mot est dans le dictionnaire (*lexicon*).

« Lexicon »

La structure du dictionnaire est simple, une entrée pour chaque mot :

```
lexicon(Cat,Sem,Phrase,Misc)
```

où `Cat` est la catégorie syntaxique, `Sem` est une façon de représentation sémantique, `Phrase` est la chaîne de caractères de ce mot, `Misc` contient l'information diverse selon la catégorie (une genre, par exemple, `male` ou `female`).

Les catégories syntaxiques utilisées : `iv, tv, det, pn, pro, noun, relpro, prep, mod, neg, coord, dcoord.`

Quelques exemples : Les catégories syntaxiques utilisées :

```
lexicon(iv,collapse,[collapse],inf)
lexicon(iv,collapse,[collapses],fin)
lexicon(noun,boxer,[boxer],human)
lexicon(det,_,[every],uni)
lexicon(det,_,[a],indef)
```

En effet, ces entrées ne contiennent pas de l'information sémantique. Il y a que des étiquettes de types `[quelquemot]`. Le vrai sémantique se passe dans les *semantic macros*.

« Semantic macros »

Semantic macros sont le niveau intermédiaire entre la syntaxe et la sémantique. En effet, chaque catégorie syntaxique est associée à une règle sémantique, qui prend comme argument un mot de cette catégorie. Cette solution permet de se passer de types sémantiques (la section I.2.), puisque la typisation est chiffrée dans ces règles intermédiaires. Chaque macro est une formule abstraite de certaine catégorie. Quelques exemples des macros :

```
nounSem(Sym,lambda(X,Formula)):-
    compose(Formula,Sym,[X])
tvSem(Sym,lambda(K,lambda(Y,K@lambda(X,Formula)))):-
    compose(Formula,Sym,[Y,X])
detSem(uni,lambda(P1,lambda(P2,forall(X,(P1@X)>
    (P2@X))))).
```

Ce niveau est plus important, sur lequel tout le travail de Blackburn & Bos est concentré. Des macros nouveaux sont ajoutés, si on veut décrire des phénomènes nouveaux.

Pour résumer : l'utilisateur entre la phrase, ensuite le dictionnaire attribue une catégorie syntaxique à chaque mot, les règles syntaxiques déterminent la structure et l'ordre d'application fonctionnelle, les *semantic macros* construisent la représentation sémantique.

En faisant le total, on peut dire que l'expérience, que nous avons appris du travail de Blackburn & Bos, comprend les notions du « grammar engineering » et l'organisation de la grammaire. Au niveau pratique, on a extrait la partie considérable de la notation, qui convient bien pour réalisation technique.

En outre, tout l'exposé de la grammaire et son implémentation ont contribué beaucoup dans notre compréhension de fonctionnement d'une théorie sémantique.

II.2. La théorie de Kratzer & Heim

Le livre de Kratzer & Heim « *Semantics in Generative Grammar* » (1998) est un exposé conséquent de la théorie de l'anglais dans l'approche de Montague. Les auteurs ont trouvé un compromis idéal entre le formalisme strict et l'évidence linguistique. Les travaux de Montague (« *English as a Formal Language* » et « *The Proper Treatment of Quantification in Ordinary English* ») sont très concis et formels, mais il n'est pas resté beaucoup de place pour les explications linguistiques ; en conséquence, ce n'est pas facile, à notre avis, du premier regard de voir une liaison entre le formalisme présenté et la langue anglaise. Au contraire, la théorie de Kratzer & Heim, en continuant la tradition de Montague, propose une théorie très équilibrée.

Comme cette théorie est devenue la base de la grammaire du français, que nous allons présenter dans la section suivante, on en parlera en détail.

Description générale

Le but principal, comme on s'y est déjà habitué, est de concevoir le sens d'une phrase :

« To know the meaning of a sentence is to know its truth-condition. A theory of meaning, then, pairs sentences with their truth-conditions. »

Le schéma adopté par Kratzer & Heim est très simple :

La phase «_____» est vrai, si et seulement si _____.

Donc, « le ciel était bleu le mardi», ssi le ciel était bleu le mardi.

Ce schéma a été proposé dans l'ouvrage de Alfred Tarski « *The concept of Truth in Formalized Languages* » (1935). Au premier abord, cette solution serait triviale, s'il n'existait pas une autre particularité de la langue humaine – la possibilité de comprendre les phrases, qu'on n'a jamais entendues. Partant de là, on peut concevoir le sens d'une phrase à partir des sens de ses parties. Donc chaque partie significative contribue aux conditions de la vérité d'une manière systématique (le Principe de Compositionnalité). On peut citer Donald Davidson (« *Inquiries into Truth and Interpretation* »):

« The theory reveals nothing new about the conditions under which an individual sentence is true; it does not make those conditions any clearer than the sentence itself does. The work of the theory is in relating the known truth conditions of each sentence to those aspects ("words") of the sentence that recur in other sentences, and can be assigned identical roles in other sentences. Empirical power in such a theory depends on success in recovering the structure of a very complicated ability – the ability to speak and understand a language. »

En définissant les éléments d'analyse, Kratzer & Heim manient les notions *saturées/non-saturées*, qui viennent de Frege :

« Statements in general, just like equations or inequalities or expressions in Analysis, can be imagined to be split up into two parts; one complete in itself, and the other in need of supplementation, or "unsaturated." Thus, e.g., we split up the sentence "Caesar conquered Gaul" into "Caesar" and "conquered Gaul." The second part is "unsaturated" - it contains an empty place; only when this place is filled up with a proper name, or with an expression that replaces a proper name, does a complete sense appear. Here too I give the name "function" to what this "unsaturated" part stands for. In this case the argument is "Caesar". »

Il y a deux types d'éléments saturés dans le sens de Frege : des phrases complètes et des noms propres, parce qu'ils ne prennent pas d'arguments et ils ne sont pas des fonctions. Toutes les autres unités sont considérées comme non-saturées. Un exemple trivial : *Pierre aime* est une phrase incomplète par rapport au sens. Saturation est, à notre avis, un synonyme en sémantique de la valence

syntaxique de Tesnière. Si toutes les positions sont occupées, la valence est complète, sinon, il faut les remplir.

Kratzer & Heim proposent une théorie extensionnelle, elles ne s'occupent pas de l'intensionnalité (sauf quelques exemples dans le dernier chapitre). Leur but est de décrire formellement les extensions des phrases, elles sont désignées par des crochets, donc l'extension (le terme *extension* est utilisé pour *dénotation* de Frege) d'une expression α est marquée comme [[α]].

extension(Pierre) = [[Pierre]]

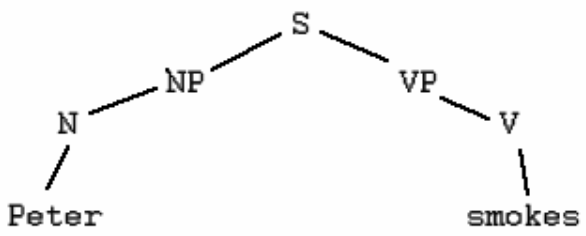
extension(Pierre aime Marie) = [[Pierre aime Marie]]

L'extension d'une phrase correspond à sa valeur de la vérité avec les conditions de la vérité. Les valeurs de la vérité sont dans la manière habituelle : 1 (*true*) et 0 (*false*).

Parfois, on mélange les exemples anglais et les exemples français, parce que pour l'instant il n'y a pas de différence en ce qui concerne le raisonnement.

Structure de la grammaire

Kratzer & Heim définissent la structure de la grammaire en trois parties : l'inventaire de dénotations, la lexique et l'ensemble des règles syntaxico-sémantiques pour chaque combinaison d'application possible. La grammaire à chaque étape de construction couvre un certain domaine des faits linguistiques. Kratzer & Heim élargissent leur grammaire de l'anglais successivement chapitre par chapitre, mais on peut commencer par un exemple assez simple *Peter smokes* et citer la grammaire pour cette exemple:



1. L'inventaire de dénotations

Soit D un ensemble de tous les individus, qui existent dans le monde. Les dénotations possibles sont :

Les éléments de D, l'ensemble des individus.

Les éléments de {0, 1}, l'ensemble des valeurs de la vérité.

Les fonctions de D vers {0, 1}.

2. Lexique

Pour les noms propres

[[Peter]] = Pierre

[[Moscow]] = Moscou etc.

Pour des autres mots, il faut les définir pour chaque catégorie, mais par exemple, les verbes intransitifs sont définis comme cela :

[[work]] = f : D → {0, 1}

Pour tous les x∈D, f(x) = 1, ssi x travaille.

[[smoke]] = f : D → {0, 1}

Pour tous les x∈D, f(x) = 1, ssi x fume.

Les verbes intransitifs sont définis comme cela :

[[love]] = f : D → {g : g est la fonction de D vers {0, 1}}

Pour tous les x∈D, f(x) = g_x : D → {0, 1}

Pour tous les y∈D, g_x(y) = 1, ssi y aime x.

Ou encore plus formel :

[[love]] = f : D → {g : D → {0, 1}}

Pour tous les x,y∈D, f(x)(y) = 1, ssi y aime x.

3. Règles pour les nœuds non-terminaux

Si α est de la forme S → β γ, alors [[α]] = [[γ]]([[β]])

Si α est de la forme NP → β, alors [[α]] = [[β]]

Si α est de la forme VP → β, alors [[α]] = [[β]]

Si α est de la forme N → β, alors [[α]] = [[β]]

Si α est de la forme V → β, alors [[α]] = [[β]]

Cette petite grammaire est évidente, elle fonctionne directement. On fait surgir les nœuds terminaux [[Peter]] et [[smokes]] jusqu'au dernier nœud S, où l'application se passe selon la première règle de notre grammaire: [[smokes]]([[Peter]]). Si l'on regarde la définition de [[smokes]], c'est une fonction, qui attend un argument. Donc on obtient :

[[Peter smokes]] = [[smokes]]([[Peter]]) =

{Pour tous les x∈D, smoke(x) = 1, ssi x fume, ou x est Peter} =

{ smoke(Peter) = 1, ssi Pierre fume ; smoke(Peter) = 0, ssi Pierre ne fume pas }

On a obtenu la valeur de vérité, et aussi les conditions de la vérité de cette phrase, ce qui était notre objectif.

Pour l'instant, cette dérivation pourrait paraître une sorte de tautologie. Mais on mâche cet exemple simple pour montrer la grammaire dans l'action et pour expliquer les raisonnements, qui sont derrière le formalisme, que Kratzer & Heim développent.

Lambda-calcul

Avant que nous allions en détails dans la conception de Kratzer & Heim, il nous reste à examiner une technique, qui s'appelle *lambda-calcul*, à l'aide de laquelle toutes les formules logico-sémantiques sont construites.

Le lambda-calcul est un formalisme, un langage de programmation théorique inventé par Alonzo Church dans les années 1930. Ce formalisme trouve plusieurs application dans l'informatique théorique (il est égal au Machine de Turing dans le capacité). On peut distinguer deux grandes classes de lambda-calculs : les lambda-calculs non typés et les lambda-calculs typés. Dans un lambda-calcul typé un certain type s'attribue à chaque expression de ce formalisme ; un lambda-calcul non-typé n'a pas de ce mécanisme. Cette distinction est très essentielle, parce que, en fait, ce sont deux formalismes différents.

En sémantique formelle, on utilise conventionnellement lambda-calcul typé, qui peut être considéré comme une extension fonctionnelle du calcul des

prédicats par l'introduction de *lambda opérateur*, ainsi donc la tradition classique de Montague continue.

Les avantages du lambda-calcul sont dans la flexibilité, la simplicité et l'uniformité. Ce formalisme est devenu un moyen technique standard dans les théories sémantiques modernes : Blackburn & Bos, Kratzer & Heim, Ronnie Cann etc. Le langage de notre propre grammaire emploie λ-calcul. En conséquence, nous allons consacrer ce paragraphe aux points cruciaux de lambda notation pour n'y plus retourner dans les sections suivantes.

Lambda opérateur donne plus de contrôle sur des variables liées dans une expression. Normalement, λ-opérateur transforme une expression dans un prédicat à un argument. Lambda opérateurs peuvent se combiner, chaque λ ajoute un paramètre substituable au prédicat. La lambda notation est extrêmement utile dans des expressions longues avec plusieurs variables.

Quelques exemples :

`λx.(man(x))` ou plus simple `λx.man(x)`

`λx.(λy.(love(x,y)))` ou plus simple `λxλy.love(x,y)`

Pour l'instant, ce n'est qu'une réécriture des formules déjà connues.

En outre, les expressions complexes peuvent aussi être abstraites par lambda opérateur :

`λP.λQ.∀x.(P(x)>Q(x))` c'est la formule pour « every », qui prend comme arguments deux expressions complexes. On voit, que λ détermine la portée d'effet d'une variable, pareillement aux quantificateurs.

Le symbole lambda nous montre qu'il y a des éléments d'information, qui nous manquent, et les occurrences de la variable liée par lambda marquent les endroits, où on attend cette information.

Donc on a affaire à une reformulation de la notion « fonction », qui est appliquée à un argument. C'est exactement « l'application fonctionnelle » décrite dans le premier chapitre.

Il y a deux opérations importantes dans lambda-calcul, qui assurent l'application fonctionnelle correcte: alpha réduction et bêta réduction.

Alpha réduction

Cette procédure a un caractère technique. Il y a intérêt à utiliser lambda notation, seulement si l'on peut garantir, que la portée des variables est bien déterminée et qu'il n'y pas des erreurs dans liaison entre les λ-opérateurs et les paramètres substituables. De telles erreurs peuvent se produire, si l'on rassemble l'expression à partir des expressions mineures. Par exemple, la phrase :

```
Mia likes the policeman.
```

se compose de quatre éléments `Mia, likes, the, policeman`:

```
Mia
λx.λy.like(x,y)
λx.policeman(x)
```

On utilise `x` dans toutes les entrées, mais évidemment les variables dénotées par `x` ne réfèrent pas toujours les mêmes objets, `x` réfère à `Mia` et pas à `policeman`.

L'alpha-réduction est une procédure de changement du nom des variables dans une expression rassemblée, afin qu'elles soient correctes au niveau de référence et liaison.

Bêta réduction

Bêta réduction (ou lambda conversion) est une procédure de transfert d'un argument à la fonction. Cette application sera désignée par le symbole `@`. Par exemple, la formule pour la phrase :

```
Peter love Marie
```

est générée d'une manière suivante :

```
((λx.(λy.(love(y,x))))@(Marie))@(Peter)
(λy.(love(y,Marie)))@(Peter)
love(Peter,Marie)
```

Lambda-calcul est très sensitif à l'ordre d'application des arguments, le résultat en dépend dans une grande mesure, parce que on obtient un autre sens dans l'ordre inverse :

```
((λx.(λy.(love(y,x))))@(Peter))@(Marie)
(λy.(love(y,Peter)))@(Marie)
love(Marie,Peter)
```

Lambda-calcul typé

En sémantique formelle, normalement, les types sont utilisés pour enlever les cas, où l'application fonctionnelle n'est pas possible (les types ne sont pas compatibles). Mais, par exemple, Blackburn & Bos ont implémenté lambda-calcul non-typé. Ils se passent de types en raison de la spécificité de son travail : ils sont plus intéressés dans les techniques computationnelles, que dans la construction de théorie complète et rigoureuse, et ils sont obligés de suivre la ligne de programmation de Prolog. L'introduction de chaque mécanisme clef en théorèmes de Prolog demande la révision de tout le programme. Et l'introduction des types surchargerait le programme, qui est déjà assez difficile à comprendre entièrement.

Kratzer & Heim ont développé la théorie avec les types sémantiques (c'est-à-dire, lambda-calcul typé).

La définition des types sémantiques reste inchangée depuis Montague. Seulement la notation est légèrement modifiée. On emploie deux types basiques :

1. `e` est le type des individus

   $D_e := D$

2. `t` est le type des valeurs de vérité

   $D_t := \{0, 1\}$

Ces types permettent de générer un nombre indéfini de types dérivés. Généralement, D_τ est l'ensemble de dénotations possibles de type τ. En plus des types basiques `e` et `t`, qui correspondent aux dénotations saturées de Frege, il y a des types dérivés, qui sont dénotations non-saturées de Frege. On peut les désigner par l'ensemble de paires ordonnés $<\sigma,\tau>$, ce qui correspond aux fonctions, dont les arguments sont de type σ et dont les valeurs sont de type τ. Par exemple,

$D_{<e,t>}$ = {f : f est une fonction de D_e vers D_t}

$D_{<e,<e,t>>}$ = {f : f est une fonction de D_e vers $D_{<e,t>}$}

Afin de lier ces notions avec les faits linguistiques, on peut dire que $D_{<e,t>}$ est le domaine pour les verbes intransitifs et $D_{<e,<e,t>>}$ est le domaine pour les verbes transitifs. Apparemment, type ∈ marche comme une valence ouverte dans l'expression, qu'il faut remplir. Donc $D_{<e,t>}$ doit trouver un élément, $D_{<e,<e,t>>}$ – deux éléments etc.

Plus de la linguistique

Maintenant, quand on possède tout l'appareil technique, on peut aborder finalement la sémantique computationnelle.

Généralement, pour décrire les extensions des phrases naturelles Kratzer & Heim définissent deux parties : le dictionnaire et les règles de combinaison pour chaque catégorie syntaxique.

Les règles les plus importantes, qui définissent l'application sont les suivantes :

1. Les nœuds terminaux

   Si α est un nœud terminal, [[α]] est défini en dictionnaire.

2. Les nœuds non branchant

   Si α est un nœud non branchant et β est son fils, alors [[α]] = [[β]].

3. L'application fonctionnelle

   Si α est un nœud branchant, {β, γ} sont ses fils et [[β]] est une fonction, dont le domaine contient [[γ]], alors [[α]] = [[β]]([[γ]]).

Evidemment, il existe plusieurs cas, où ces règles ne suffisent pas pour résoudre les problèmes. Dans ces cas-là, Kratzer & Heim introduisent les règles précisant, mais nous n'allons pas répéter ici tous les cas, parce qu'il faut donner la vision générale de la théorie.

Faisons le résumé du lexique, qui est en considération pendant quelques chapitres :

0. les noms propres (type : ∈) :

   [[Pierre]] = Peter

[[Moscou]] = Moscow

1. les noms (type : <e, t>) :

   [[maison]] = λx. x is a house

   [[homme]] = λx. x is a man

2. les verbes d'un argument (type : <e, t>) :

   [[travailler]] = λx. x works

   [[fumer]] = λx. x smokes

3. les verbes de deux arguments (type : <e, <e, t>>) :

   [[aimer]] = λx.λy. y loves x

   [[manger]] = λx.λy. y eats x

4. les verbes de trois arguments (type : <e, <e, <e, t>>>) :

   [[donner]] = λx.λy.λz. z gives x to y

   [[itroduire]] = λx.λy.λz. z itroduces x to y

5. les adjectifs (type : <<e, t>, <e, t>>) :

   [[gris]] = λP.λx.(x is grey & P(x))

   [[heureux]] = λP.λx.(x is happy & P(x))

6. les pronoms (type : <<e, t>, t>) :

   [[personne]] = [[nobody]] = λP.!∃x.P(x)

   [[quelqu'un]] = [[somebody]] = λP.∃x.P(x)

   [[chaqu'un]] = [[everybody]] = λP.∀x. P(x)

7. les pronoms (type : <<e, t>, e>) :

   [[aucun]] = [[no]] =λP.λQ.!∃x.(P(x)>Q(x))

   [[chaque]] = [[every]] = λP.λQ.∀x.(P(x)>Q(x))

8. les déterminants (type : <<e, t>, e>) :

   [[le]] = [[the]] = λP.λQ.∃x∀y.((P(x)=x=y)&Q(x))

9. la coordination (type : <t, <t, t>>) :

   [[et]] = [[and]] = λP.λQ.(P=Q=1)

10. les mots sémantiquement vides (dans la qualité des fonctions, qui rendent eux-mêmes)

$$[[de]] = [[of]] = \lambda x \in D_e.\ x\ ([[de\ John]] = [[John]])$$

$$[[be]] = [[être]] = \lambda f \in D_{<e,t>}.\ f\ ([[be\ rich]] = [[rich]])$$

$$[[une]] = [[a]] = \lambda f \in D_{<e,t>}.\ f\ ([[a\ cat]] = [[cat]])$$

On répète, que ce n'est pas une liste exhaustive. Kratzer & Heim ont examiné bon nombre de situations linguistiques. Il y a plusieurs astuces dans l'utilisation de ces définitions, il faut les réviser chaque fois, quand on rencontre le cas particulier, mais on peut avoir ce dictionnaire pour point de départ. Certains de cas plus difficiles seront examinés dans la grammaire du français, qu'on va présenter dans la section suivante, qui est pour beaucoup est fondée sur la théorie de Kratzer & Heim.

II.3. La grammaire construite du français

La construction d'une vraie grammaire du français est un objectif très complexe même avec des grandes restrictions. A notre avis, il est inaccessible même en hypothèse, sans compter le temps limité de ce travail. On peut se rappeler le travail de Montague « *Universal Grammar* » (1970), qui commence par la thèse :

« It is clear that no adequate and comprehensive semantical theory has yet been constructed, and arguable that no comprehensive and semantically significant syntactical theory yet exists. »

Presque quarante ans passés, il n'existe pas de théorie linguistique, qui décrirait complètement la syntaxe et la sémantique d'une langue avec succès.

Quand même il y a au moins deux faits, qui nous ont encouragé :

1. Nous avons un but précis : écrire la grammaire du français, qui sera réalisé sur ordinateur.
2. Nous avons examiné plusieurs théories de ce paradigme dans les sections précédentes.

Encore une difficulté consistait en ce, qu'il existe peu de théories montagoviennes, élaborées sur le matériau de la langue français. Le livre de

Michel Galmiche « *Sémantique linguistique et logique* » nous a aidé dans cet aspect, ainsi que le livre de Michel Chambreuil « *Grammaire de Montague* ». Mais comme nous avons déjà dit, nous avons emprunté le maximum de Blackburn & Bos et Kratzer & Heim.

Après avoir pris en compte toutes ces stipulations, commençons par les principes de construction de cette grammaire.

Principes de construction

Un avantage de la construction de sa propre grammaire est dans le fait, qu'on peut choisir, quels termes utiliser parmi des centaines de termes déjà inventés, ou redéfinir ces termes. Nous comprenons sous la notion « grammar engineering » emprunté à Blackburn & Bos, tel organisation de la grammaire, qui dépend de l'objectif de la grammaire. Dans notre cas, c'est un logiciel, qui construise des formules logico-sémantiques à partir des phrases françaises dans le cadre de l'approche de Montague. La question, pour quel but on crée ce logiciel, est laissée hors de la discussion. Mais nous le considérons d'avoir au moins la valeur pédagogique assez intéressante.

En conséquence, nous avons dû trouver la balance entre la valeur scientifique (la « profondeur » de la grammaire) et l'évidence de la grammaire (la « largeur » de la grammaire), parce qu'elles sont inversement proportionnelles. En outre, le temps de son implémentation d'une grammaire est malheureusement directement proportionnelle à la « taille » de la grammaire.

A notre avis, cela sera intéressant d'assembler la grammaire dans une manière « bottom-up », puisque toutes les théories antécédentes étaient construites d'une façon « up-bottom ». En outre, « bottom-up » raisonnement répond au principe de « grammar engineering ».

Dans notre cas, le « bottom » est une phrase naturelle et le « up » est la formule correspondante. Si l'on prend comme un exemple la phrase :

les chiens tachetés courent dans la rue

on peut essayer de construire la grammaire « ponctuelle », en la généralisant ensuite autant que possible.

C'est clair, que si l'on veut traiter de cette phrase d'un bout à l'autre, il faut passer tous les niveaux d'analyse, ce qui n'est pas le but de ce mémoire, donc on va faire passer ou réduire certaines parties d'analyse, en se concentrant sur les points importants.

Heureusement, nous avons auparavant discuté plusieurs pièces de la grammaire computationnelle, nous allons utiliser ces pièces au fur et à mesure.

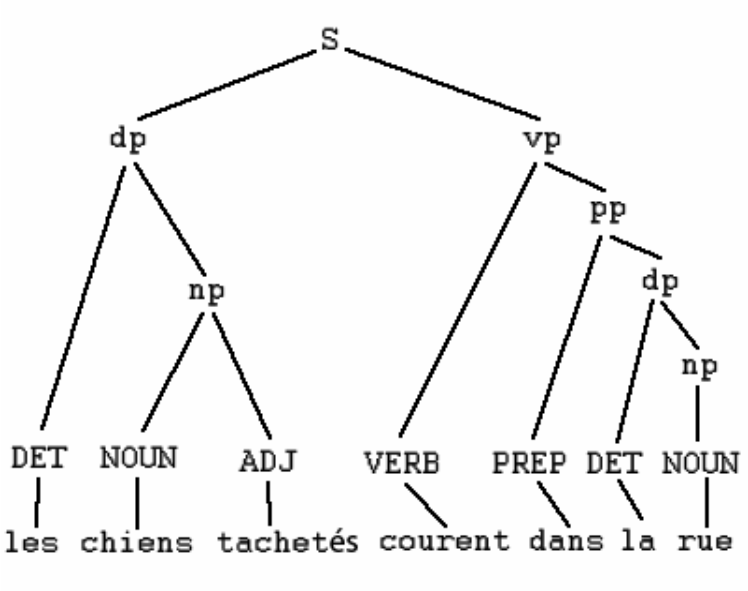
Syntaxe de la grammaire

Du point de vue d'un praticien, premièrement, on a besoin d'extraire la structure de cette phrase, donc il faut formaliser le niveau de syntaxe. S'étant familiarisé avec quelques théories de sémantique computationnelle, on impose la première restriction : des arbres syntaxiques doivent être binaires.

La solution entendue est de diviser cette phrase comme ça :

$[_{s}[_{dp}[_{det}\textit{les}][_{np}[_{noun}\textit{chiens}][_{adj}\textit{tachetés}]]]][_{vp}[_{v}\textit{courent}][_{pp}[_{prep}\textit{dans}]\ [_{np}[_{det}\textit{la}]\ [_{noun}\textit{rue}]]]]]$

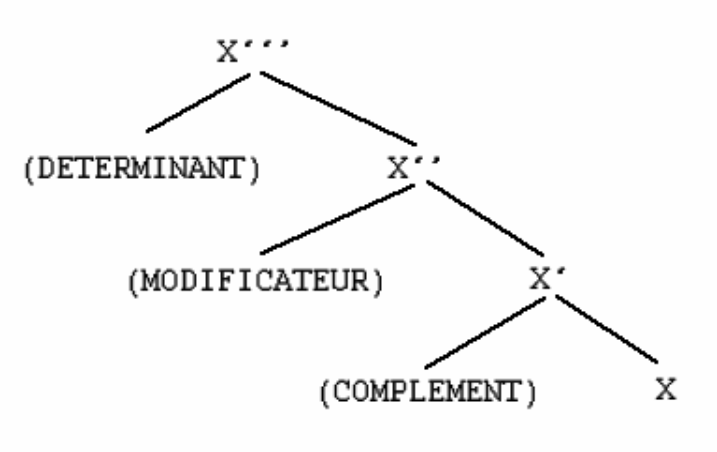
Et l'arborescence :



On voit, que c'est une analyse standard de constituants immédiats, initialisée encore par Bloomfield. Son conventionnalisme nous attire dans notre

mission pratique, parce qu'il ne faut pas expliquer son essence. Les pour et les contre sont bien connus et étudiés. Cela ne vaut pas la peine de tout répéter ici.

Il existe plusieurs théories de constituants immédiats. Nous allons s'en tenir à la théorie bien connue « X-bar theory », proposée par Chomsky et élaborée par Jackendoff. On en récupère la notation et l'envie de présenter toutes les structures linguistiques d'une façon universelle (emprunté à Henk von Reimsdijk, « *Theory of Grammar* ») :



On peut expliquer la simplicité, avec laquelle nous prenons tel ou tel morceau d'une théorie, par le fait, que pour l'instant on se prépare à la partie essentielle – la construction des formules, et on n'a pas besoin prendre en considération tous les astuces de la théorie de syntaxe. On ne veut avoir qu'un mécanisme, qui génère des arborescences correctes de phrases testées. Ainsi donc, toutes les implications philosophiques de la théorie X-barre restent inexploitées, car on n'en a pas besoin ici.

La binarité d'arbre est une conséquence naturelle, car l'on construit une grammaire formalisée autant que possible (pour ensuite la réaliser sur ordinateur, ce qui est important aussi). Chaque arbre non binaire peut être transformé en l'arbre binaire, c'est un fait informatique bien connu (A. Aho, R. Sethi, J. Ullman « *Compilers: Principles, Techniques, and Tools* »), donc la capacité de la grammaire ne se rétrécit pas.

Les règles de réécriture pour notre exemple auront le format suivant :

S → dp vp ;

dp → DET? np ;

np → NOUN ADJ? ;

vp → VERB pp? ;

pp → PREP dp ;

ou DET, NOUN, VERB, PREP sont les nœuds terminaux,

dp, vp, np, pp sont les nœuds non terminaux, S étant le symbole initial.

Ces règles font dériver la phrase testée et l'ensemble infini de phrases de cette structure, mais comment pourrait-on généraliser ces règles pour avoir une « boite noire », qui reconnaîtrait toutes les phrases naturelles ? Evidemment, cette question de la syntaxe ultime reste ouverte. Si on parle d'applications de TAL réelles, les modèles statistiques ou les théories mixtes (quand toute l'information sur phrase est examinée au même temps) sont utilisés plus souvent.

Heureusement, ce sujet est au-delà de la problématique de ce travail. Mais il faut résoudre la question de la génération des arbres syntaxiques d'une manière régulière. En projetant cette grammaire pour l'ordinateur, il vient l'idée d'utiliser Context-Free Grammars (les grammaires hors-contexte), car elles sont simples à un certain degré, bien formalisées, étant facilement réalisées sur ordinateur.

Formellement, la grammaire hors-contexte est une grammaire formelle dans laquelle chaque règle de production est de la forme

$$V \rightarrow w$$

où V est un symbole non terminal et w est une chaîne composée de terminaux et/ou de non-terminaux. Le terme « hors-contexte » provient du fait qu'un non-terminal V peut toujours être remplacé par w, sans prendre en compte son contexte. Un langage formel est hors-contexte s'il existe une grammaire hors-contexte qui le génère.

Le problème avec les grammaires hors-contexte appliquées à la langue naturelle est qu'elles ne conviennent pas très bien pour refléter toutes les dépendances de syntaxe de cette langue. Elles peuvent définir la structure d'une phrase séparée, même définir une partie de la langue. Par exemple, la grammaire

au-dessus est de type hors-contexte, et elle décrit toutes les phrases du français des mots avec les catégories syntaxiques correspondantes :

les chiens tachetés courent dans la rue

les chats gris courent dans la rue

les chats gris dorment dans la rue

les chats gris dorment dans les maisons etc.

C'est déjà pas mal. Mais un petit écart de ce pattern suffit pour que la grammaire ne marche plus :

**les chats gris et blanc dorment dans la rue*

En plus, généralement, on s'intéresse à des cas plus difficiles. Comment résoudre ce problème ? Il n'y a pas de solution prête, mais nous, nous proposons l'issue suivant. En se souvenant, que notre objectif est la sémantique et la construction pratique des formules, il faut avoir non une seule grammaire décrivant toute la langue, mais avoir un constructeur des grammaires hors-contexte applicables dans le cas propre :

G = {l'ensemble des grammaires g_i, où *i* parcourt les domaines d'une langue naturelle, chaque domaine ayant une grammaire hors-contexte g définissant ce domaine}

A notre connaissance, c'est une idée originale, qui n'a pas été exploitée. On comprend très bien les défauts de cette approche :

1. Les domaines sont difficiles à limiter. En fait, on divise la langue en domaines, où les grammaires hors-contexte sont heureusement applicables. Ce n'est pas évident de savoir comment choisir la grammaire correcte dans un cas concret. Ce n'est pas évident : qu'est ce qui va passer aux bornes des domaines ?

2. L'objectif traditionnel de la linguistique est de chercher des théories universelles, c'est-à-dire d'unifier les appareils et d'essayer de décrire tout d'une façon identique. L'idée d'avoir un ensemble des grammaires est contraire à l'intention générale de la linguistique à l'uniformité.

Nous n'avons pas de réponses à ces problèmes, mais nous avons des commentaires :

1. On ne se concentre pas sur l'élaboration de cette idée, parce que nous étudions la sémantique dans ce travail et pas la syntaxe.
2. Probablement, cette idée est sans issue générale scientifique, mais conformément aux tâches locales elle résout le problème d'incapacité des grammaires hors-contexte d'écrire complètement une langue.
3. Sur le plan pratique cette idée nous a permis de construire un appareil intéressant, qui donne la possibilité de tester cette idée. Le résultat est présenté dans la partie pratique de ce travail, consacrée à l'interface syntaxico-sémantique GANSS (le chapitre III).

On admet, que c'est possible avec certaines restrictions d'obtenir un fractionnement des phrases d'une langue naturelle aux ensembles, où chaque ensemble serait décrit par une grammaire hors-contexte. On admet, qu'il est possible d'élaborer un mécanisme, qui choisirait la grammaire nécessaire dans les points de choix. Dans une certaine approximation, une telle organisation pourrait exister. Le degré d'approximation dépend de l'objectif final.

Dans notre réalisation tout le problème du choix repose sur l'utilisateur, qui souhaite obtenir la formule logico-sémantique pour sa phrase. Il doit choisir g_i courant (ou la définir, si elle n'existe pas). Nous nous occupons de la sémantique et créons la carcasse syntaxique ; il reste à l'utilisateur de faire son choix dans les points, qui dépassent notre capacité à les résoudre.

Cette approche se distingue considérablement de la position de la linguistique théorique, mais c'est exactement ce qu'on comprend sous « grammar engineering ». Nous reviendrons à cette question dans la section III.

La question suivante est celle de la nature des nœuds terminaux. L'arbre syntaxique se construit selon les règles de réécriture, qui sont basées sur les catégories syntaxiques, qui s'attribuent à chaque mot ou à un groupe de mots encadré par guillemets, si l'on veut traiter de toute l'expression intégralement :

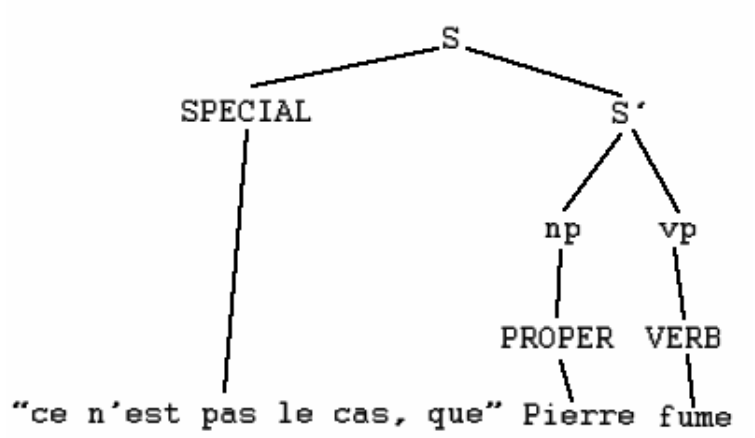
ce sont les mots ordinaires, *chacun ayant sa propre catégorie*

et *« un groupe des mots, ayant une seule catégorie pour tout le groupe »*.

Nous allons utiliser des catégories syntaxiques traditionnelles : NOUN, VERB, ADJ, PREP, PROPER etc. (une liste complète est donnée dans la description du GANSS). Comment attribuer les catégories syntaxiques est une question de réalisation d'un *analyseur lexical*. Nous n'irons pas dans le niveau inférieur de la grammaire (analyse du lexique). Ce niveau ne contribue en rien à la construction de la grammaire de notre objectif, car ce niveau est résolu par les moyens informatiques (construction des classes de mots selon leur type de déclinaison).

En outre, nous introduisons une catégorie spéciale SPECIAL, qui peut contenir les actions sémantiques spéciales. Nous l'avons incorporée pour la négation, mais nous croyons, que cette catégorie peut être utilisée pour les autres buts aussi.

Par exemple, « *ce n'est pas le cas, que » Pierre fume*.



C'est une catégorie abstraite, qui nous aide de résoudre les difficultés pratiques. L'utilisation de cette catégorie est assez astucieuse, on l'élucidera dans la section pratique. Mais au fond elle correspond à certaines transformations sémantiques.

Sémantique de la grammaire

A présent, nous avons obtenu le dispositif syntaxique, qui génère les arborescences des phrases naturelles. Nous sommes prêt à définir le bloc sémantique.

Comme nous l'avons dit, le repère dans notre édification scientifique est la théorie de Kratzer & Heim, donc nous adopterons plusieurs solutions déjà trouvées par ces auteurs, qui à leur tour continuent la stratégie de Montague.

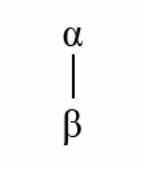
Application fonctionnelle

L'étape suivante est l'application fonctionnelle, qui implique, que les objets participant dans cette opération sont les *fonctions*. Du point de vue technique l'application est une opération dans laquelle on obtient la formule logico-sémantique d'un nœud composé des formules logico-sémantiques de ses fils. Pour l'effectuer, on a besoin des formules initiales et des règles sémantiques d'application.

Commençons par les règles.

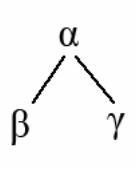
Avec la syntaxe, qu'on a définie, il y a deux cas structurels, qui correspondent aux trois interprétations :

i)



$\alpha \rightarrow \beta$

ii)



$\alpha \rightarrow \gamma(\beta)$

$\alpha \rightarrow \beta(\gamma)$.

Le cas i) est simple, on assigne la sémantique de β à α.

Le cas ii) accepte deux interprétations, puisqu'il y a deux coopérandes : un *foncteur* et un *argument*, soit γ étant un foncteur et β étant un argument, soit vice-versa :

si [[β]] est dans le domaine de [[γ]], alors [[α]] = [[γ]]([[β]])

si [[γ]] est dans le domaine de [[β]], alors [[α]] = [[β]]([[γ]])

sinon les constituants sont incompatibles.

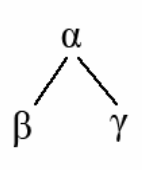
Ici surgissent les types sémantiques de Montague, car ils déterminent les domaines des expressions. Quand deux nœuds se combinent pour constituer la

sémantique du nœud supérieur, leurs types sont comparés et la configuration correcte est choisie : un élément devient foncteur, et l'autre – son argument.

Ces règles sont locales : la sémantique du père est déterminée par les sémantiques de ses fils, c'est une conséquence du Principe de Compositionnalité, qui nous permet de ne pas prendre en considération tout ce qui se passe hors de la structure locale. Il faut définir ce qu'on comprend sous « la sémantique » d'un élément. Dans notre grammaire, la sémantique d'un élément se compose de deux parties :

1. La formule logico-sémantique, qui est une partie essentielle.
2. Le type sémantique, qui est une partie auxiliaire. Les types nous donnent le contrôle des règles sémantiques.

Par rapport au fonctionnement des types, si l'on a



ou

$\beta \in D_{<p,q>}$

$\gamma \in D_{<m,n>}$

1) β devient le foncteur et γ – l'argument, si `p=<m,n>`, alors α reçoit le type `q`.

2) γ devient le foncteur et β – l'argument, si `m=<p,q>`, alors α reçoit le type `n`.

Si ni 1), ni 2) ne marchent, c'est une erreur de type, donc l'application sémantique n'est pas possible.

Apparemment, c'est exclu, que `p=<m,n>` et `m=<p,q>` au même temps, car ils sont cyclés.

L'algèbre de types est très simple. Il existe une seule règle : `<a,b>@a=b`

Donc, on a

```
<e,t>@e=t
<t,t>@t=t
<e,<e,t>>@e=<e,t>
<<e,t>,e>@<e,t>=e
```

etc.

Toute autre application n'est pas possible.

A présent, on sait, comment combiner les nœuds, il nous reste à comprendre, ce qui se passe avec les formules logico-sémantiques à l'application, d'où viennent ces formules et comment elles sont organisées.

Formules logico-sémantiques

Chaque expression linguistique jouit d'un certain sens. Comme le mot « sens » est très provocateur, on le changera par le terme « extension », examiné auparavant. Notre modèle est extensionnel, donc il s'agit des extensions, qui doivent être captées par les formules.

Traditionnellement, l'unité minimale remplie sémantiquement est le mot. C'est au niveau des mots, qu'il faut définir les premières formules, à la base desquelles les formules intégrales de l'expression complexes seront construites.

Dans la construction des formules nous avons adopté la notation de Blackburn & Bos, parce qu'elle convient très bien à la réalisation sur ordinateur.

Ces formules représentent le résultat, qu'on veut obtenir à la fin du processus de la composition sémantique. Comme on l'a déjà vu, ces formules se construisent à partir des formules minimales, à partir des formules de mots. Donc notre but est de définir le vocabulaire, c'est-à-dire de créer un dictionnaire de λ-formules (un λ-dictionnaire). Une entrée dans un tel dictionnaire est caractérisée par 4 champs : lemma, catégorie syntaxique, type est formule.

En fait, on peut grouper les mots selon leurs catégories syntaxiques : les noms comme « maison », « ville », « homme » etc. sont des « enveloppes » référentielles aux objets d'un classe. Si on parle dans les termes des informaticiens, de tels objets sont « générés » (en anglais *to instantiate*) dans la parole (à l'opposition de la langue dans le sens de Saussure) par des moyens différents (normalement par un déterminant, mais d'autres variants sont aussi possibles). Evidemment, ces « enveloppes » fonctionnent de la même manière et elles ont une structure sémantique identique.

On peut dire la même chose sur les verbes, classés selon la structure de leurs arguments. Les verbes transitifs « aimer », « adorer », « haïr » et « acheter »

ne se distinguent que par l'action. Mais cette action est enveloppée dans une structure sémantique identique : il faut avoir deux arguments et le même contexte syntaxique.

De tels « enveloppes » sont directement saisis par les « semantic macros » chez Blackburn & Bos.

En fait, dans tout ce travail il s'agit des « enveloppes » sémantiques, des « enveloppes » pour capter et garder le sens. Notre travail consiste en élaboration d'un mécanisme pour transmettre le sens dans les « enveloppes » et combiner ces « enveloppes » dans les « paquets » de telle manière qu'ils gardent la structure des sens compris et afin qu'ils soient « combinables » eux-mêmes. En termes imagés, à la fin, ces « paquets » constituent le « colis » de sens fini, qui peut être envoyé à un homme ou à une machine (le dernier est beaucoup plus intéressant).

Chaque phénomène linguistique peut être décrit par les divers moyens. Notre grammaire est orientée vers lexique. C'est-à-dire que s'il existe des méthodes différentes pour résoudre un certain problème, on choisit toujours le moyen au niveau du lexique, si possible : on essaie de mettre toute l'information nécessaire dans les définitions des mots. De cette façon, le lexique est aussi un mécanisme de modéliser les phénomènes linguistiques, comme les structures syntaxiques auparavant.

Revenons au vocabulaire. L'idée générale est que on peut changer les définitions des mots, si on considère un certain fait linguistique. Mais pour avoir un dictionnaire initial nous adoptons le dictionnaire de Kratzer & Heim. Le niveau d'extension nous permet d'utiliser les mêmes formules de l'anglais pour le français. En cas de la spécifité française, le mot peut être aussitôt redéfini.

En faisant le total, prenons un exemple de l'inférence d'une expression classique : *Chaque homme aime une femme.*

Chaque homme : DET + NOUN → np

$[(\lambda Q.(\lambda P.(\forall x.(Q(x)>P(x)))))@(homme(y))] =$

$= [(\lambda P.(\forall x.(homme(x)>P(x))))] =\alpha$

une femme : DET + NOUN → np

[(λQ.(λP.∃y(Q(y)&P(y))))@(femme(z))] =

= [(λP.∃y.(femme(y)&P(y)))] = β

aime [une femme] : VERB + np → vp

[(λK.(λx.(K@(λy.(aimer(y,x))))))@(β)] =

= [(λK.(λx.(K@(λy.(aimer(y,x))))))@([(λP.∃z.(femme(z)&P(z)))])] =

= [(λx.(λP.∃z.(femme(z)&P(z)))@(λy.(aimer(y,x)))) =

= [(λx.(∃z.(femme(z)&aimer(y,x)))] = γ

[Chaque homme] [aime [une femme]] : dp + vp → S

[α@(γ)] = [([(λP.(∀x.(homme(x)>P(x))))])@(γ)] =

= [(λP.(∀x.(homme(x)>P(x))))@(λr.(∃z.(femme(z)&aimer(y,r)))] =

= [(∀x.(homme(x)>(λr.(∃z.(femme(z)&aimer(z,r))))))@(x)] =

= [(∀x.(homme(x)>(∃z.(femme(z)&aimer(z,x)))))]

La formule finale est : (∀x.(homme(x)>(∃z.(femme(z)&aimer(z,x)))))

Nous avons construit la grammaire à trois parties. Le mécanisme sémantique est durement fixé par le Principe de Compositionnalité et les types sémantiques, mais le composant syntaxique et le composant lexique sont assez flexibles pour couvrir certain nombre des expressions linguistiques.

Jusqu'ici, la grammaire construite semble assez abstraite, mais le troisième chapitre consacré à sa réalisation informatique devrait être plus pratique.

III.

Réalisation informatique de l'interface syntaxico-sémantique

Un des problèmes généraux de la linguistique (dont une partie est sans doute la sémantique computationnelle) est la difficulté dans l'application pratique des résultats d'une théorie et l'absence d'évidence linguistique. Très peu de théories linguistiques sont utilisées dans l'industrie de TAL, donc les tâches pratiques ne peuvent pas servir comme critère enlevant des théories injustifiées. Apparemment, nous ne parlons pas de l'aspect cognitif d'une théorie, ce qu'on ne peut pas mesurer par applicabilité d'une théorie.

Malheureusement, il est très fréquent qu'il existe une rupture entre la théorie construite et l'expérience linguistique, une rupture dans le sens, que derrière l'édification théorique et le perfectionnement d'outillage des faits linguistiques d'une langue soient pris en considération très abstraitement, et on ne peut « valider » la théorie donnée que par des exemples ponctuels. A notre avis, c'est un défaut grave dans le développement d'une théorie. En conséquence, depuis le début des travaux sur les problèmes posés dans les sections précédentes, un des buts principaux consistait à construire une interface d'ordinateur réalisant notre grammaire linguistique. A notre avis, c'est dans cette partie, que nous avons obtenu les résultats les plus intéressants.

III.1. Description générale de l'initiative

L'objectif principal de l'interface syntaxico-sémantique consiste en démonstration des idées de la sémantique computationnelle dans l'action. Nous avons voulu donner à l'utilisateur la possibilité de voir toutes les étapes d'analyse, qui se passent en une grammaire construite dans le cadre de l'approche de Montague. Bien que nous comprenions, que la valeur pédagogique de notre travail est plus évidente, nous attribuons aussi à notre interface certaine valeur

scientifique, sans compter l'expérience obtenue dans le travail sur l'interface. La valeur scientifique est en ce, que l'utilisateur peut construire lui-même ses propres grammaires et les tester par notre interface. Nous allons en parler en détails plus tard. Mais avant d'aborder notre interface, il faut étudier des programmes analogues.

Programmes analogues

Les recherches d'implémentations des théories sémantiques sur ordinateur n'ont pas donné beaucoup de résultats. On peut alléguer les programmes suivants, qui traitent de l'anglais :

1. Blackburn & Bos, la grammaire en Prolog réalisé partiellement en DORIS (Discourse Oriented Representation and Inference System) http://www.coli.uni-saarland.de/~bos/doris/
2. C&C tools (James Curran, Stephen Clark, Johan Bos) http://svn.ask.it.usyd.edu.au/trac/candc/wiki/
3. NTLK toolkit (Steven Bird, Edward Loper, Ewan Klein) http://nltk.sourceforge.net/

On ne parle pas de plusieurs grammaires de teste en Prolog, qui sont normalement développées dans le cadre des cours d'études, parce qu'elles doivent mettent des étudiants au courant de Prolog et son application en linguistiques, habituellement pas plus.

DORIS

Nous avons déjà décrit le travail de Blackburn & Bos (la section II.1). DORIS est une interface en ligne développé en 2001, dont les buts étaient :

1. étudier les inférences en sémantique computationnelle ;
2. illustrer les outils de sémantique computationnelle introduits dans le livre « *Representation and Inference for Natural Language* ».

Malheureusement, cette interface n'est plus supportée. A présent, il ne marche pas, alors qu'il se trouve dans l'Internet. Donc il ne nous reste, qu'à prendre connaissance de sa documentation :

« Implemented in Prolog, DORIS uses a left-corner parser to cover a (admittingly rather small, but continually growing) fragment of English syntax and a lexicon of a few thousand words based on the *Pulp Fiction* script. It computes an underspecified discourse representation (UDR) of the input sentence, followed by resolving ambiguities – thereby taking into account the previous discourse and some background knowledge. »

DORIS a été abandonnée au dire de Johan Bos en faveur de l'outil C&C.

C&C

Nous n'avons pas eu possibilité d'apprécier et de tenir compte les avantages de ce système entièrement, parce que C&C est apparu dans l'accès public très récemment (le 1 juin 2007), mais à en juger d'après les rapports d'autres chercheurs, cet ensemble d'outils est très puissant, surtout au niveau de la vitesse.

Les outils dans C&C sont taggers, CCG parser et le module sémantique Boxer. Aussi l'outil morphologique externe `morpha` est utilisé. Ils sont développé en format de *command script.*

Nous nous sommes surtout intéressés au module sémantique Boxer, développé par Johan Bos. Sa distinction technique d'autres composants est ce qu'il a été réalisé en Prolog (les autres sont en C++). Il construit des DRS sur la base d'analyse syntaxique. Prenons un exemple du site, l'input est `Every man runs` et l'output comme cela :

```
%%% Every man runs .

sem(1,

    [
     word(1001, 'Every'),
     word(1002, man),
     word(1003, runs),
     word(1004, '.')
    ],

    [
     pos(1001, 'DT'),
     pos(1002, 'NN'),
     pos(1003, 'VBZ'),
     pos(1004, '.')
    ],

    [
    ],
```

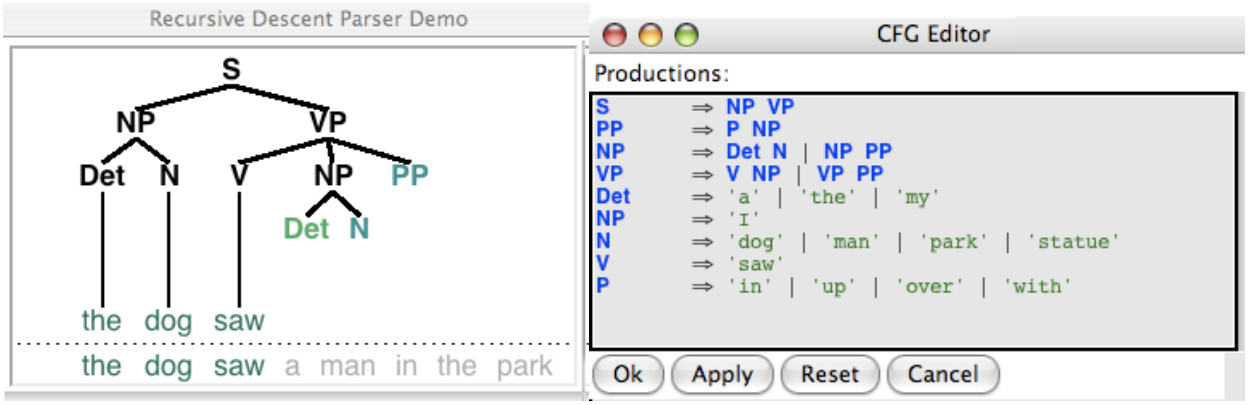
```
     [
      l0:drs([], [l1]),
      l1:[1001]:imp(l2, l4),
      l4:drs([[1003]:_G3858], [l5, l6, l7]),
      l5:[1003]:pred(_G3858, run, v, 0),
      l6:[]:pred(_G3858, event, n, 1),
      l7:[1003]:rel(_G3858, _G3808, agent, 0),
      l2:drs([[1001]:_G3808], [l3]),
      l3:[1002]:pred(_G3808, man, n, 0)
     ]
    ).

/* ===== DRS (pretty print) =====

 ____________________________________
|                                    |
|____________________________________|
|  ___________      _______________  |
| | x1        |     | x2            | |
| |___________|     |_______________| |
| | man(x1)   | ==> | run(x2)       | |
| |___________|     | event(x2)     | |
|                   | agent(x2,x1)  | |
|                   |_______________| |
|____________________________________|

============================== */
```

On croit, que cet outil peut être utilisé dans plusieurs tâches, qui demandent la construction de DRS.

NLTK

NLTK est un ensemble d'outils linguistiques, réalisés en Python, dont la première version est apparue en 2001. Pour l'instant (version 0.8), il n'existe pas de partie sémantique, mais les niveaux précédents sont très bien élaborés : *tokenization, tagging, chunking* et *parsing*. Ce fait nous permet de faire une supposition, que l'analyse sémantique dans ce toolkit peut rapidement arriver.

Nous payons l'attention à NLTK, parce que ce toolkit est un projet *open source* très connu parmi linguistes. On peut dire, qu'il est devenu un standard pour d'autres logiciels de ce domaine. Outre son utilisation dans plusieurs projets de TAL, NLTK est une base pour les nombreux cours d'études de linguistique computationnelle, ce qui le fait intéressant pour notre travail aussi. Une autre analogie est dans les visualisations, que NLTK propose, et la possibilité de définir les grammaires au vol.



A notre avis, c'est indispensable d'étudier ce toolkit, si l'on veut développer des programmes analogues.

Pour et contre de Prolog

Prolog est un système unique. Plusieurs programmes de ce domaine sont écrits en Prolog, généralement, ils modélisent quelques exemples linguistiques et présentent des grammaires d'exercice. Au sens strict, ces programmes sont des réalisations des grammaires en Prolog, ils ne sont pas des logiciels indépendants.

L'avantage des programmes, basés sur Prolog, est la rapidité de développement, parce que pour tester quelque point de la théorie, il suffit d'ajouter quelques règles à la grammaire, et la puissance de Prolog comme un système des inférences.

Du point de vue technique, Prolog est un système assez capricieux : il y a plusieurs versions qui ne sont pas toujours compatibles. C'est assez difficile de l'intégrer un autre logiciel, de le connecter à l'Internet ou d'aboutir à l'interface plus amicale.

Le deuxième défaut des systèmes, basés sur Prolog, est une absence d'évidence, l'input et l'output en Prolog est spécifique. Les textes des programmes sérieux sont assez lourds, leur compréhension est difficile, en conséquence, il est difficile de partager l'expérience obtenue grâce à de tels programmes ou les réutiliser.

En somme, on dirait, que le Prolog est un système très utile dans la vérification des points ponctuels de la théorie dans l'intérêt du chercheur, mais que

c'est un choix manqué pour présenter le fonctionnement de la théorie au public. Ce choix est obligatoire, parce que si l'on veut construire un programme, qui implémenterait la théorie assez large à l'échelle réelle, c'est-à-dire pas une grammaire de test, mais en le faisant approcher à de l'expérience linguistique naturelle et en réduisant les restrictions autant que possible, le travail commence à ressembler au développement d'une application de TAL commercial sous le rapport de la complexité et du temps de production.

Exigences de programme

Un des objectifs de ce travail était d'essayer de changer le stéréotype dans l'écriture de logiciel linguistique. Ayant tenu compte de l'expérience des grammaires en Prolog, nous avons pris une autre conception d'implémentation sur ordinateur de la théorie proposée. Parmi les exigences les plus importantes, il faut énumérer les suivants :

a. Accessibilité au programme
b. Spectacularité (évidence) du programme
c. Fonctionnement approché de la réalité
d. Extensibilité du programme

Les trois premières conditions ont défini le choix principal du fondement technique : l'interface a dû être réalisé dans le format d'une web-application, ce qui assure l'accès public, qui garantit les moyens pour visualiser la théorie dans l'action, qui permet de l'utiliser au même degré que le logiciel TAL en ligne.

Comme le projet a été prévu assez lourd au niveau des ressources et assez difficile au niveau du fonctionnement, Java est devenu le langage de programmation principal.

L'objectif initial (idéal et inabordable) était de construire le système, qui prend la phrase du français naturel en input et rend la formule logique à la fin du travail selon la conception, les principes et les règles de la théorie proposée.

Evidemment, ce but n'est pas faisable dans l'extension complète par définition, mais on avait envie d'approcher le fonctionnement du programme du

fonctionnement d'une application TAL réelle. La restriction de temps n'a pas permis de tout réaliser, mais nous avons obtenu une application intéressante.

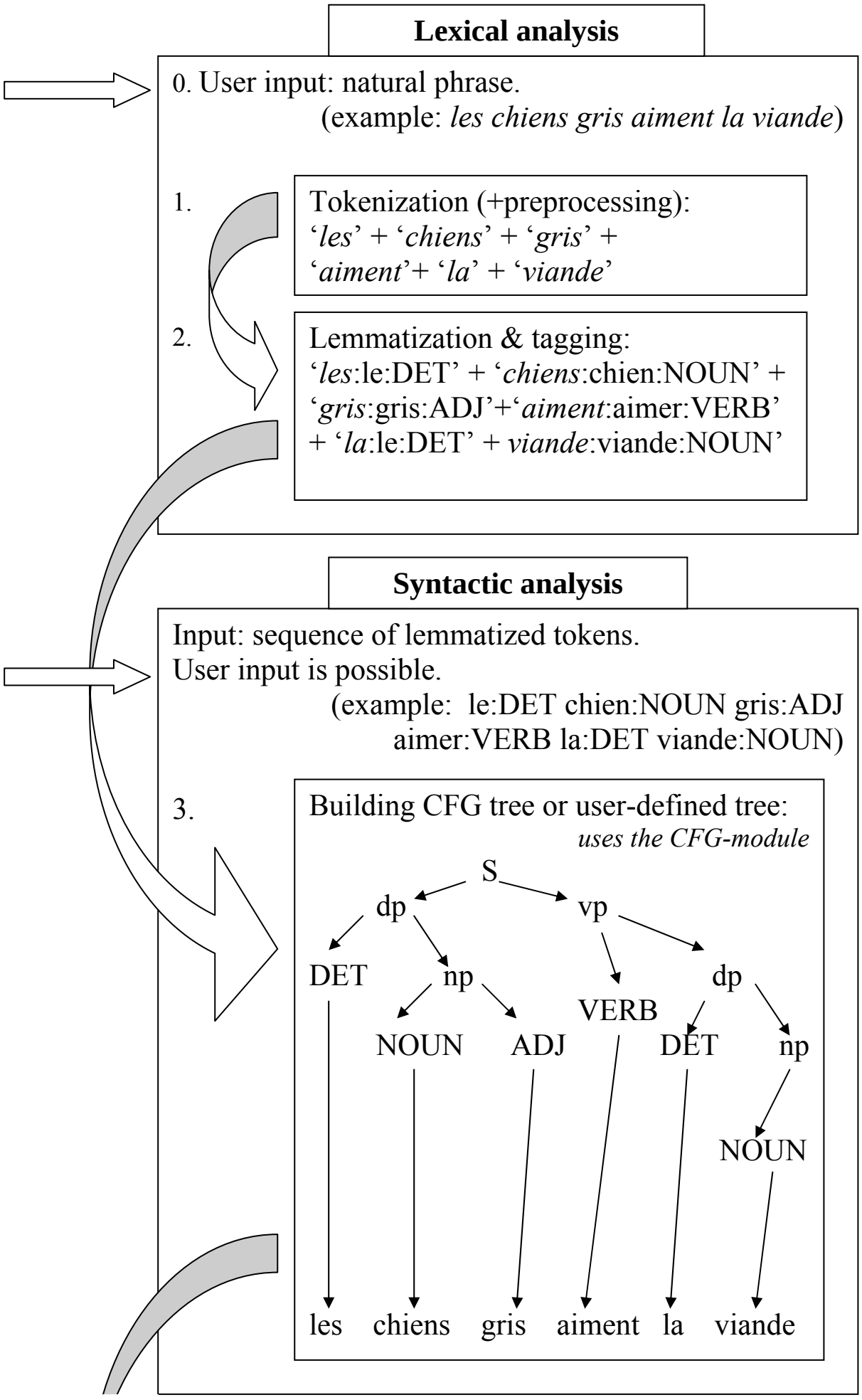
Le programme, qui s'appelle sans prétentions ni peu ni prou GANSS (Grammar ANalyzer, Syntax & Semantics) construit les formules logiques des phrases à la base des dictionnaires sémantiques définis par l'utilisateur selon soit les grammaires syntaxiques CFG, soit les structures syntaxique définies par l'utilisateur. A la rigueur on peut considérer ce logiciel comme « constructeur des grammaires », car deux parties principales sur trois sont redéfinissables : lexique et règles syntaxiques.

III.2. Structure du GANSS

GANSS est une web-application en ligne, réalisé en grande partie dans un ensemble de `Java Servlets`. Java servlet est une application Java qui permet de générer dynamiquement des données au sein d'un serveur HTTP, ils s'exécutent dynamiquement sur le serveur Web. GANSS marche sous un serveur standard Apache Tomcat. Technologiquement, GANSS est fait en plusieurs langages de web-programmation : perl, php, javascript, mais la partie essentielle est écrite en Java.

L'organisation du GANSS est présentée dans le schéma suivant.

servlet `application`



Semantic application

Input: binary syntactic tree
User input is possible.
(example: [[le:DET] [[chien:NOUN] [gris:ADJ]] [[aimer:VERB] [[le:DET] [[viande:NOUN]]]]])

4.

Functional application on the syntax tree
uses the λ-module

<<e,t>, e> : [[le]]

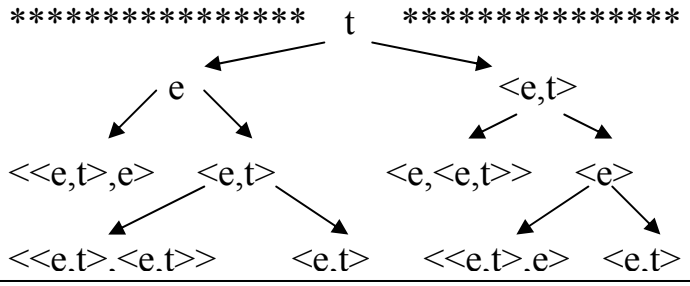
= λP.λQ.∃x∀y.((P(x)=x=y)&Q(x))

<e,t> : [[chien]] = λx.(chien(x))

<<e,t>, <e,t>> : [[gris]] = λx.(gris(x))

<e, <e,t>> : [[aimer]] = λy.(λx.(aimer(x,y)))

<e,t> : [[viande]] = λx.(viande(x))



Context-Free Grammars

(*CFG-module*)
servlet `grammar`

- Creates and manages CFG-grammars, which are used in syntax parsing.
- User input grammar is of conventional linguist friendly syntax.
- Each grammar is compiled as reusable standalone java-class.

Lambda-dictionnaries

(λ-module)
servlet `lambdar`

- Creates and manages λ-dictionnaries, which are used in semantic application.
- Typed lambda calculus is implemented as a logic language of entries.
- λ-dictionnaries are reusable and stored in MySQL tables.

Il y a trois parties principales :

1. Module des lambda-dictionnaires qui est réalisé dans le servlet `lambda`. Le but de ce module est de gérer les dictionnaires des mots avec leurs formules logiques.
2. Module des grammaires qui se compose du servlet `grammar` et du programme `JavaCCC`. Le but de ce module est de générer les parsers des grammaires CFG.
3. Module d'application dans lequel on fait la construction des formules logiques des phrases (le servlet `application`).

On voit, que cette organisation ressemble à celle de l'application de Blackburn & Bos : « lexicon » correspond au module des lambda-dictionnaires, « rules » correspond au module des grammaires et « semantic macros » correspond au module d'application. Mais la différence conceptuelle est que grammaire du GANSS est orientée vers lexique et pas vers « semantic macros ».

En outre, il y a plusieurs sections auxiliaires (réalisant upload, download, filtrage, output de pages statiques etc.), qui sont importantes pour le fonctionnement intégral du logiciel, mais elles n'ont pas de grande importance du point de vue conceptuel.

Toutes les données sont stockées dans les tableaux de MySQL, notamment, les grammaires CFG, le dictionnaire des lemmes, les dictionnaires de lambda, l'information sur utilisateurs.

Module des lambda-dictionnaires

Ce module gère des lambda-dictionnaires : création des dictionnaires, recherches dans les dictionnaires, insertion des entrées etc. L'interface est plus ou moins intuitive, on a besoin de peu de temps pour se faire aux opérations.

Ce module est responsable de « lexique » de notre grammaire. Donc la fonction générale est d'ajouter les entrées, qu'on voudra utiliser ultérieurement.

Premièrement, il faut choisir le dictionnaire courant ou créer un nouveau dictionnaire, avec lequel on travaillera. L'utilisateur peut soit chercher, soit ajouter des entrées dans le dictionnaire.

L'entrée contient quatre champs : `pos` (part of speech), `functype` (input et output), `lemma` et `formula`. L'index consiste en `pos` et `lemma`, c'est-à-dire que s'il existe déjà une telle entrée avec `pos` et `lemma`, elle est écrasée par une nouvelle.

Le champ `pos` est présenté dans l'énumération de catégories syntaxiques, qui sont reconnaissable par GANSS. Le champ `lemma` correspond au lemme du mot ou on peut introduire une expression complète, qui se compose de quelques mots, encadrée par guillemets.

Les deux champs restants sont plus intéressants. Le *functype complet* d'une entrée consiste en deux souschamps : *functype input* et *functype output*, étant les projections de la notion *type sémantique*. La syntaxe utilisée est intuitivement claire, il y a cinq lettres dans l'alphabet : '`e`', '`t`', '`<`', '`>`', '`,`', qui peuvent former les expressions de types (par exemple, pour l'entrée de *le* functype complet est `<<<e>,<t>>,<e>>` où `<<e>,<t>>` est functype input et `<e>` est functype output). Les expressions introduites par utilisateur sont contrôlées par la grammaire de functype, qui est listée dans l'annexe.

Le champ `formula` est pour la formule logique du lemma. Il y a trois mots spéciaux : `lambda`, `forall`, `exists` qui désignent respectivement λ, ∀, ∃. Les opérations logiques classiques : `&` (disjonction), `|` (conjonction), `>` (implication), `!` (négation). Les majuscules sont utilisées pour les expressions complexes (`P`, `Q`), les minuscules sont utilisées pour les variables (`x`, `y`, `z`). Toutes les expressions doivent être bien encadrées par les parenthèses.

Après avoir rempli tous les champs, l'entrée est ajoute dans le dictionnaire.

Module des grammaires

Ce module a le caractère principal, car il assure l'extensibilité du programme – le point crucial de la conception. Le but de ce module est de générer des grammaires hors-contexte, qui en leur tour génèrent des arbres syntaxiques, sur lesquels on fait l'application sémantique. Par défaut on travail avec des arbres binaires. La binarité des arbres permet de simplifier les règles d'application sémantiques.

Il y a deux possibilités pour construire des arbres syntaxiques:

1. L'utilisateur peut définir une grammaire CFG (Context Free Grammar) dans la notation conventionnelle, compiler cette grammaire et ensuite l'utiliser pour construire des arbres syntaxiques, quand on veut.
2. L'utilisateur peut donner la structure d'un arbre au moment de l'input de la phrase. Ces structures des constituants immédiats définies d'une telle manière sont plus amples, que les structures définies par CFG. Mais apparemment il faut payer pour cette liberté : on ne peut pas sauvegarder ces structures, on a besoin de les définir chaque fois à l'application.

Context Free Grammars

Il existe un servlet spécial pour générer les CFG `grammar`. Il permet d'introduire les grammaires nouvelles, de compiler les grammaires, de gérer les grammaires précédentes. Le grammaire est définie par une notation conventionnelle linguistique, une notation de production EBNF (extended Backus–Naur form). On va appeler le format de telle grammaire `grm`. Un exemple de telle grammaire est

```
S : dp vp;
dp : DET? np;
np : NOUN ADJ?;
vp : VERB np?;
```

Les éléments en majuscules sont les terminaux, les éléments minuscules sont non terminaux. Chaque règle contient deux parties séparées par deux-points, un non terminal est à la gauche et une production de deux éléments à la droite terminaux ou non terminaux. L'alphabète de terminaux est :

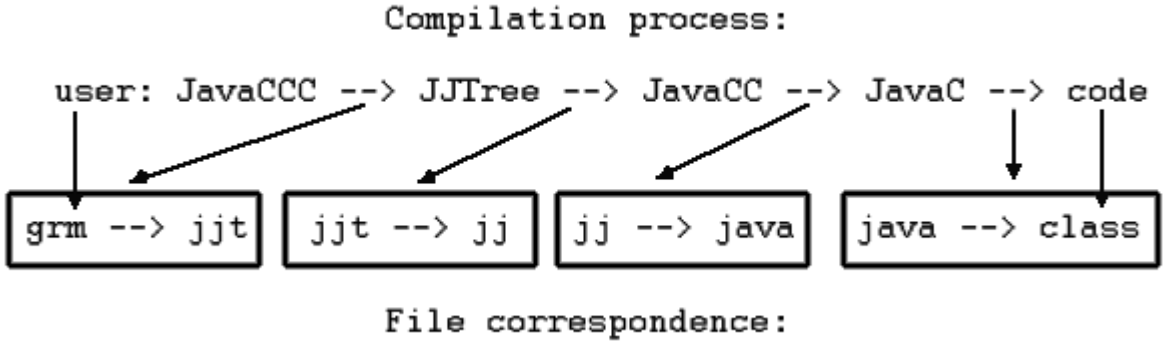
< TERM: "ADV" | "ADJ" | "VERB" | "NOUN" | "DET" | "ACR" | "PREP" | "PROPER" | "PRO" | "COORDCONJ" | "SUBCONJ" | "INT" | "WEAKPRO" | "PUNCT" | "COMPLEX" | "POINT">

Les règles peuvent être modifiées par les symboles standard d'expressions régulières : `*, +, ?, (), |`.

La syntaxe complète est présentée dans l'annexe.

Compilation

Du point de vue technique, afin qu'on puisse utiliser ces grammaires dans le module sémantique, on a besoin de compiler un `class` fichier de Java. La compilation est faite en quatre étapes et cette procédure a été la plus difficile à réaliser. La chaîne de compilation est la suivante :



L'utilisateur introduit sa grammaire dans le champ textuel de web-browser et l'envoie au serveur, ensuite sur le serveur il lance le programme `JavaCCC` (étant une part du GANSS) qui traduit la syntaxe `grm` dans le format de l'input de `JJTree (jjt)` et ainsi de suite jusqu'au fichier `class`. Le processus de compilation est administré par le logiciels `Apqche Ant`. La compilation se lance dans le `thread (fork)` parallèle.

A l'output on reçoit le fichier `nomdegrammaire.class` qui est le parser de la grammaire spécifiée (en fait, on reçoit beaucoup de fichier, mais celui-ci est le plus important). Ensuite on peut utiliser ce parser pour générer des arbres.

Dans les outils comme `JavaCC,` il y a normalement deux sections : *scaner* et *parser* (le premier pour l'analyse lexicale et le deuxième pour l'analyse syntaxique), par exemple, `yacc` et `flex`. `JavaCC` aussi peut faire l'analyse lexicale, mais on utilise notre propre `TokenManager` en raison de son efficacité, car l'analyse lexicale linguistique est beaucoup fine, que l'analyse lexicale générale.

A notre avis cette solution technique est dans une certaine mesure unique, car elle permet de générer dynamiquement les parsers des arbres syntaxiques sur le serveur sans participation de l'utilisateur. Cette organisation donne la possibilité de compiler et appliquer les grammaires sans relancer le serveur, sans relancer le programme.

La pièce principale de cette organisation est le programme `JavaCCC` écrit à son tour en utilisant le `JavaCC` (Java Compiler Compiler). C'est un logiciel indépendant assez solide pour gérer la notation conventionnelle linguistique de CFG (EBNF). Nous avons dépassé le temps considérable pour le créer. A notre avis, on peut l'utiliser chaque fois, quand on a besoin de travailler avec les grammaires CFG.

`JJTree, JavaCC` et `JavaC` sont les outils standard de Java. `JJTree` ajoute l'information nécessaire pour sauvegarder des arbres dans le parsing et génère les fichiers de l'input de `JavaCC`, qui à son tour construit les parsers/scaners et génère les fichiers de l'input de `JavaC`, dont l'output est les fichiers `java`, qui se compilent et s'ajoutent dynamiquement dans le `deployment` de GANSS. Depuis là la CFG est prête à utiliser, c'est-à-dire parser les phrases d'utilisateur en construisant des arbres syntaxiques. Si la grammaire viole la syntaxe ou s'ils arrivent des erreurs dans la compilation, le log de processus est disponible.

Les structures définies par utilisateur

Un autre moyen de construire des arbres syntaxiques est d'indiquer la structure au moment de l'introduction d'une phrase dans le servlet `application` (c'est-à-dire, au moment de construction de la formule).

Par exemple, si on a la phrase *les chiens du garçon courtent dans la rue*, on peut indiquer la structure par les crochets :

[[*les* [*chiens* [*de* [*la fille*]]]] [*courtent* [*dans* [*la rue*]]]]

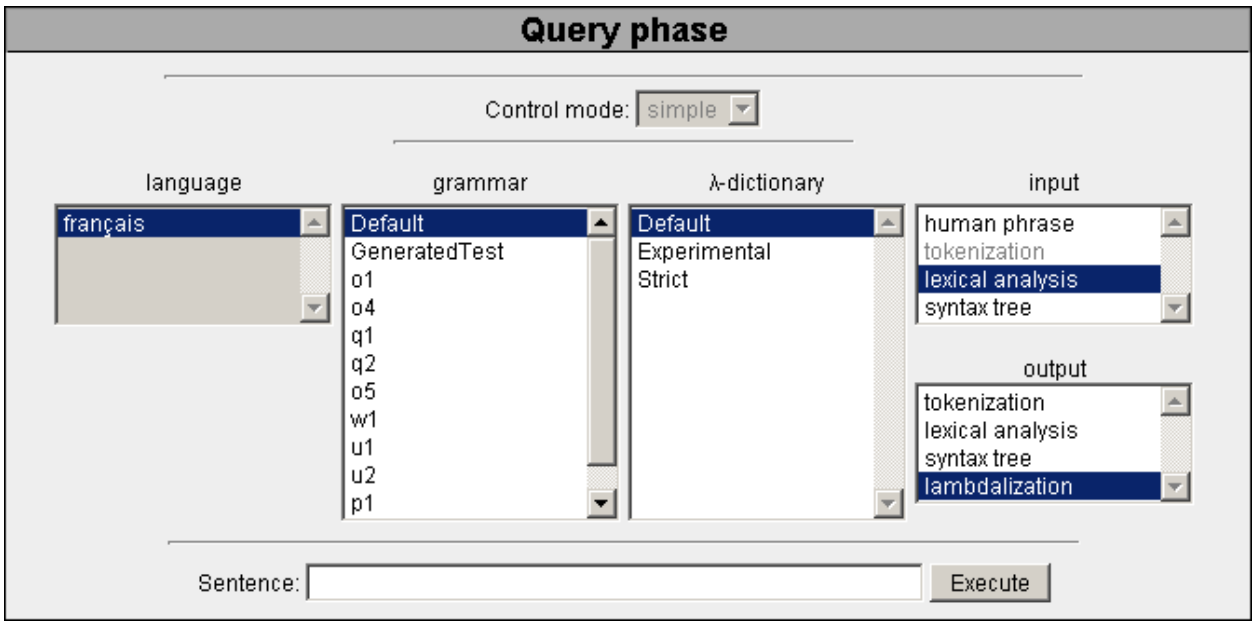
Le mécanisme est intuitivement claire : chaque paire des crochets désigne un nœud syntaxique intermédiaire contenant les fils. Les mots sont des nœuds terminaux. Pour avoir un arbre binaire chaque nœud intermédiaire ne contient pas plus que deux fils. La syntaxe de l'input est contrôlée par la grammaire de ce type (donné dans l'annexe).

Si l'on utilise cette notation, il n'y a pas de restrictions de CFG (récursivité à gauche). L'application des règles sémantiques se fait directement sur la structure donnée, mais le prix est que cette option n'est pas rigoureusement formalisée dans la théorie, elle dépend de l'utilisateur, mais dans tous les cas elle ne contredit pas la conception.

III.3. Fonctionnement du GANSS

La construction des formules logiques est le but principal du GANSS, elle se fait dans le module d'application.

Ce module permet de régler les options, qui conduisent le processus. Il y a quatre options : `grammar`, `λ-dictionnary`, `input`, `output` et champ d'entrée pour la phrase à parser `sentence`.



Par `grammar` on choisit, quel ensemble de règles utiliser ; par `λ-dictionnary` respectivement, quel λ-dictionnaire utiliser. L'option `input` détermine le type de l'input, les valeurs possibles sont :

1. `human phrase` – une phrase comme elle est ;
2. `lexical analysis` – une séquence de tokens avec la catégorie syntaxique résolue ;
3. `syntax tree` – une séquence de tokens avec la catégorie syntaxique plus avec l'information sur l'arbre syntaxique à construire.

Les exemples complets sont donnés ci-dessous, dans les sections suivantes.

L'option `output` détermine le type de l'output, qu'on veut recevoir, les valeurs possibles sont :

1. `lexical analysis` – une séquence de tokens avec la catégorie syntaxique résolue ;
2. `syntax tree` – un arbre syntaxique.
3. `lambdalization` – une formule logique.

Evidemment, ces options correspondent aux niveaux classiques d'analyse d'une phrase :

1. Analyse lexicale ;
2. Analyse syntaxique ;

3. Analyse sémantique.

GANSS permet de commencer et finir par l'étape souhaitable. Le schéma ci-dessus décrit le processus d'analyse réalisée. Nous allons le considérer l'étape par l'étape.

Etape lexicale

Cette étape a le caractère auxiliaire, car la partie essentielle de la théorie, comme on a déjà vu, commence à partir d'un arbre syntaxique et consiste en l'application des règles sémantiques au nœud par nœud. Ainsi donc la mission du module lexical est de fournir le module syntaxique de l'information nécessaire pour construire un arbre. Dans notre cas, si on reçoit le lemme, la catégorie syntaxique d'un mot suffit pour construire un arbre. Par exemple,

si l'input est

les garçons courent dans la rue

l'output devrait être la séquence de tokens

'*les*:le:DET' + '*garçon*:garçon:NOUN' +
'*courent*:courir:VERB' + '*dans*:dans:PREP'+
'*la*:le:DET' + '*rue*:rue:NOUN'

Techniquement, le travail du module est divisé à deux sections : tokenization et lemmatization. La tokenization est au fond une procédure de preprocessing de la phrase, c'est-à-dire, fractionnement et préparation de tokens à analyser. Lemmatization consiste en rétablissement d'un lemme.

La morphologie est réalisée par les tableaux de déclinaison, qui contient 158 paradigmes de verbes, noms et adjectifs français. Ils sont les ressources externes (c'est un mélange compilé de sources différentes). Si le programme ne peut pas résoudre l'ambiguïté morphologique, il propose les options possibles à l'utilisateur, afin qu'il fasse le choix final.

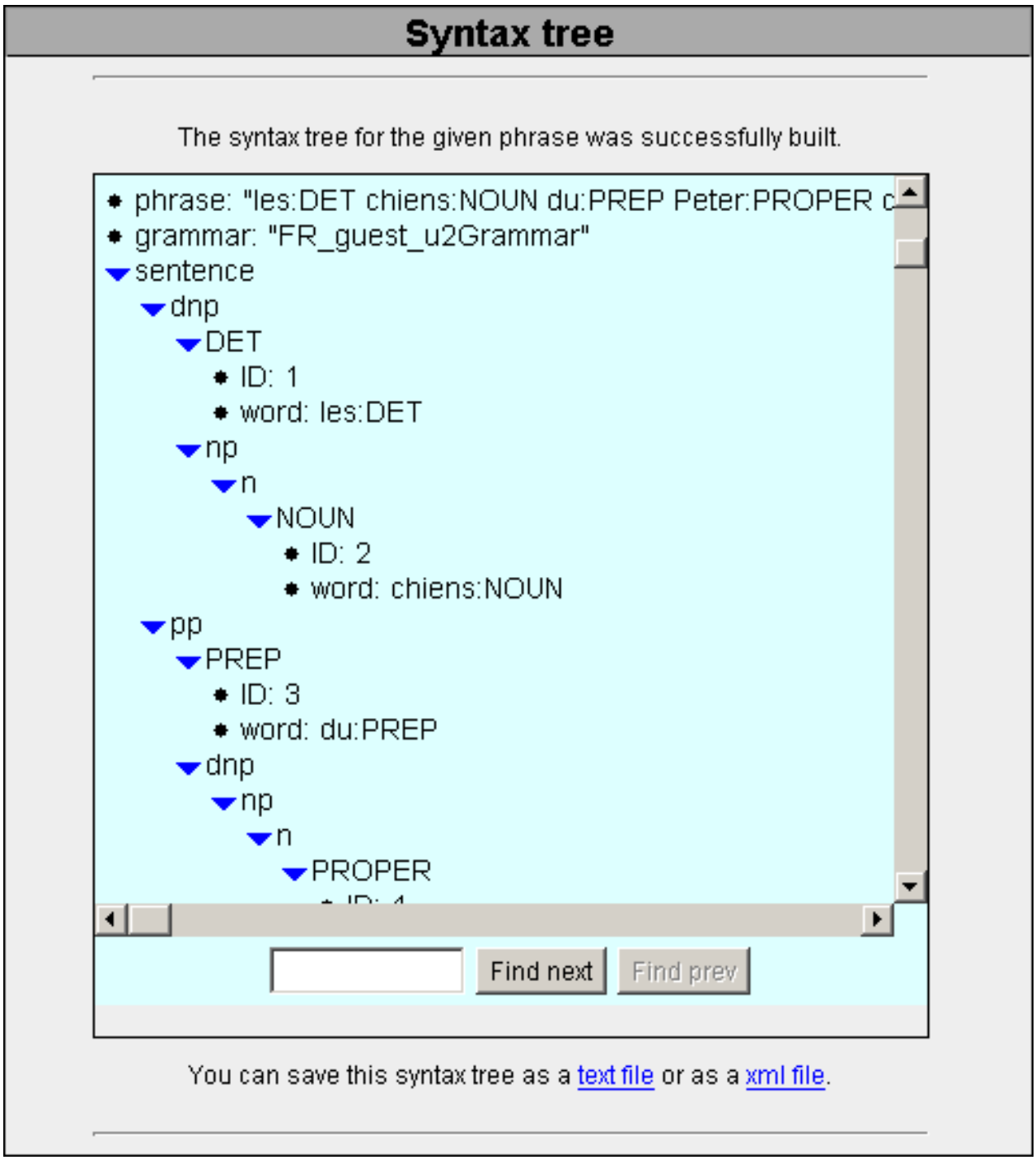
Le dictionnaire des lemmes utilisé est une compilation de dictionnaires pris de l'Internet. Le nombre de mots dans le dictionnaire est à peu près 70000. Comme tous les dictionnaires de GANSS, c'est stocké dans les tableaux de MySQL.

Le module lexical est dans l'état « brut », ce qu'on peut expliquer par la concentration de notre attention sur les autres modules et par le fait, qu'il y a une dépendance de sources externes et par la complexité générale de résolution de problèmes morphologiques (si l'on fait à l'échelle réelle). Mais dans tous les cas on peut l'utiliser.

Etape syntaxique

Après avoir reçu la chaîne de tokens, on procède à l'étape syntaxique. Ici, on assume par le *token* un élément avec une catégorie syntaxique résolue, en plus, il contient le lemme, dont on aura besoin dans l'étape sémantique.

Selon la grammaire choisie ou selon la structure proposée par utilisateur GANSS construit un arbre sur la chaîne de tokens donnée. Si la chaîne ne correspond pas à la grammaire, l'utilisateur reçoit une erreur, dans ce cas-là il doit réviser soit sa grammaire, soit sa phrase. C'est une restriction syntaxique assez générale, quand même elle peut éliminer des phrases notoirement fausses. Si le *parsing* est fini avec succès, l'arbre est publié à l'aide d'un applet. L'utilisateur peut sauvegarder le fichier de *parsing* dans le format de `txt` ou `xml`.



Etape sémantique

Dès que l'arbre est disponible, on aborde le processus, qu'on peut appeler *lambdalization*, c'est-à-dire l'application des règles sémantiques et la construction de la formule intégrale à partir des formules des mots.

En première étape, on extrait les formules logiques et les functypes du dictionnaire choisi pour chaque lemme de la phrase (c'est-à-dire, pour chaque nœud terminal). Si l'on ne trouve pas quelque lemme dans le dictionnaire, une

erreur se produit. Dans ce cas-là il faut ajouter cette entrée dans le dictionnaire et relancer le programme.

Normalement, les formules dans le dictionnaire sont présentées dans le format suivant :

[[aimer]] = `lambdax.(lambday.(x AIMER y))`

[[chaque]] = `lambdaP.(lambdaQ.(forallx.(P(x)>Q(x)))`

Après avoir obtenu les formules pour les nœuds terminaux, on lance l'opération de alpha-réduction pour résoudre tous les conflits avec lettres identiques dans les formules extraites.

Maintenant, on est prêt à construire la formule finale. Comme l'arbre est toujours binaire, on a trois cas :

1. Si le nœud est terminal, on récupère sa formule du dictionnaire.
2. Si le nœud a un seul fils, on attribue la formule du fils au nœud parent.
3. Si le nœud a deux fils, l'application fonctionnelle se produit : le nœud parent obtient la « combinaison » des formules des fils.

Le mécanisme d'établissement d'un foncteur et d'un argument est basé sur les types sémantiques. On compare les functypes en cherchant la convergence. Si le functype complet d'un nœud coïncide avec le functype input d'un autre, le premier devient l'argument et le deuxième devient le foncteur, sinon, on fait cette comparaison à l'envers, en essaient de trouver l'égalité de types au nouveau. Si l'on ne réussit pas avec deux combinaisons, cela veut dire que l'application n'est pas possible et l'utilisateur reçoit une erreur. Les functypes marchent comme des limiteurs, qui ne laissent pas passer les applications impossibles au niveau de la sémantique.

Si le fils-foncteur et le fils-argument sont déterminés, l'application est faite dans une manière droite : les formules sont encadrées par les parenthèses et l'argument est concaténé au foncteur par un symbole d'application `@`.

Par exemple, pour « chaque homme » on obtient l'expression suivante :

```
(lambdaP.(lambdaQ.(forallx.(P(x)>Q(x))))@(lambdax.(homme(x))
```

Après avoir concaténé les formules, on essaie de faire la β-réduction, c'est-à-dire de réduire le nombre des variables liés. C'est une tâche plus fine. On ne veut pas aller profondément dans les détails de l'algorithme, mais l'idée générale est qu'on confronte les arguments aux variables et réduit tous les deux, si possible.

L'expression précédente se réduit, par exemple, à la forme suivante:

```
lambdaQ.(forallx.(homme(x)>Q(x))
```

Quand la formule du nœud est construite, on attribue le functype complet au nœud, qui est égal au functype output du foncteur.

De cette façon on visite tous les nœuds de l'arbre en assemblant la formule finale, qui est notre objectif principal.

Conclusion

Résultats du travail

Nous croyons, que le résultat le plus important est la création d'un logiciel (un constructeur des grammaires), réalisant les principes de la théorie montagovienne. Au niveau du contenu, GANSS réalise toutes les idées principales de la sémantique computationnelle. Au niveau du fonctionnement, il exécute toutes les tâches qui lui incombent. Au niveau de la forme, ce logiciel correspond aux conditions exigées (accessibilité, évidence, extensibilité).

Nous apprécions avec modération la profondeur de nos recherches théoriques pour les raisons suivantes :

1. Nous avons eu un autre but prioritaire concret, qui consistait en la réalisation sur ordinateur des idées de sémantique computationnelle ;
2. Nous avons essayé de dépasser la « profondeur » par la « largeur » de faits et de théories attirés ;
3. La conception de « grammar engineering » impose certaines restrictions : l'implémentation doit suivre toute l'édification théorique. Mais en réalité ce n'est pas possible dans la limite des conditions posées, car la réalisation de chaque idée prend énormément de temps.

La théorie qui sous-tend le GANSS est basée sur des travaux récents, bien connus dans le domaine de la sémantique computationnelle. La nouveauté de cette théorie consiste en deux points :

1. La théorie est pour la langue française ;
2. La théorie est orientée vers la réalisation pratique, car elle était construite parallèlement à son implémentation.

Ces particularités ont des pour et les contre, nous avons exposé notre position dans ce mémoire. Mais dans tous les cas pendant notre travail on n'a pas

rencontré des essais semblables, ce qui rend, à notre avis, ce travail intéressant et unique dans un certain sens.

Contribution

On peut distinguer deux types de contribution que ce travail fait : contribution scientifique et contribution pédagogique.

Contribution scientifique :

1. L'expérience reçue et exposée dans l'implémentation de la grammaire de sémantique computationnelle ;
2. Adaptation des théories de l'anglais au français (notamment, Kratzer & Heim, Blackburn & Bos) ;
3. Création d'un instrument pour construire et vérifier des grammaires potentielles dans le cadre de la conception de la sémantique computationnelle ;
4. Innovation dans l'approche de la construction des interfaces linguistiques.

La contribution pédagogique consiste, évidemment, dans la création d'un instrument ouvert, facilement accessible, spectaculaire et puissant pour construire des grammaires dans le cadre de la conception de Montague.

Perspectives

La structure du programme est modulaire, tel qu'il avait été projeté depuis le début du travail. Cette organisation permet facilement d'élargir le programme. Par exemple, pour ajouter la fonctionnalité de l'anglais, il est nécessaire simplement de créer certains tableaux en MySQL, parce que tout autre travail doit être fait par utilisateur, car il définit le lexique et les structures.

Prochainement, nous voulons fixer et finir toutes les options déjà proposées, tester notre interface largement dans le sens de son applicabilité aux tâches différentes de la sémantique computationnelle. Apparemment, il y a

plusieurs améliorations à faire et plusieurs de dispositifs sémantiques à implémenter, surtout, les variables dans le sens Kratzer & Heim.

Nous espérons, que ce projet éveillera l'intérêt des gens, qui s'occupent de la sémantique computationnelle.

Bibliographie

Références principales

1. Angelika Kratzer & Irene Heim, *Semantics in Generative Grammar*, Blackwell Publishing, 1997
2. Patrick Blackburn & Johan Bos, *Representation and Inference for Natural Language*, Center for the Study of Language and Inf, 2005
3. Richard Montague, *The Proper Treatment of Quantification in Ordinary English*, The Journal of Philosophy, 1970
4. Richard Montague, *English as a Formal Language*, The Journal of Philosophy, 1970
5. Ronnie Cann, *Formal Semantics*, Cambridge University Press, 1993
6. *Montague Grammar* edited by Barbara H. Partee, Academic Press, 1976
7. A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Educational Publishers, 1986
8. Michel Galmiche, *Sémantique linguistique et logique*, Presses Universitaires de France, 1991
9. Robert D. van Valin, *An Introduction to Syntax*, Cambridge University Press, 2001
10. Henk van Reimsdijk, *Introduction to the Theory of Grammar*, The MIT Press, 1986
11. Gennaro Chierchia & Sally McConnell-Ginet, *Meaning & Grammar*, The MIT Press, 1990
12. Robin Cooper, *Quantification and Syntactic Theory*
13. Fred Landman, *Structures for Semantics*, Springer, 1991
14. D. Dowty, R. Wall, S. Peter, *Introduction to Montague Semantics*, Springer, 1980

Références secondaires

1. Richard Montague, *Universal Grammar*, The Journal of Philosophy, 1970
2. Michel Chambreuil, *Grammaire de Montague*, ADOSA, 1989
3. Charles Fillmore, *Universals in Linguistic Theory*, Holt, Rinehart, and Winston, 1968
4. Donald Davidson, *Inquiries into Truth and Interpretation*, Oxford University Press, 2001
5. G. E. Hughes, M. J. Cresswell, *Introduction to Modal Logic*, Routledge, 1996
6. J. J. Katz, J. A. Fodor, *The structure of a semantic theory*
7. Alfred Tarski, *The concept of truth in formalized languages*, Cambridge University Press, 2004
8. Jane Grimshaw, *Argument Structure*, The MIT Press, 1992
9. Alonzo Church, *A Formulation of a Simple Theory of Types*
10. Ray Jackendoff, *Semantic Interpretation in Generative Grammar,* The MIT Press, 1974
11. David Dowty, *Thematic Proto-Roles and Argument Selection*, *Language*, 67, 3, 1991
12. Ray Jackendoff, *Semantic Structures,* The MIT Press, 1991
13. Barbara Partee, *Some Transformational Extensions of Montague Grammar*
14. David Lewis, *General Semantics*, Oxford University Press, 1987
15. Norbert Hornstein, *Logic as Grammar*, The MIT Press, 1987
16. Rudolf Carnap, *Meaning and necessity*, University Of Chicago Press, 1988
17. Karen Zagona, *Verb Phrase Syntax*, Springer, 1988
18. Donald Knuth, The Art of Computer Programming, Addison-Wesley Professional, 1998
19. M. J. Cresswell, Entities and Indices, Springer, 1990
20. D. R. Dowty, Word Meaning and Montague Grammar, Springer, 1990

Annexes