Rapport de stage

Grammaires d'arbres adjoints à large couverture et grammaires catégorielles abstraites

Maxime GUILLAUME

Année 2017–2018

UFD63 Stage de licence 3 MIASHS réalisé dans le Laboratoire Lorrain de Recherche en Informatique et ses Applications au sein de l'équipe Sémagramme

Maîtres de stage : **Maxime Amblard et Sylvain Pogodalla**

Encadrant universitaire : **Sylvain Castagnos**

Remerciements

En premier lieu, je tiens à remercier vivement mes maîtres de stage, Monsieur Amblard et Monsieur Pogodalla pour m'avoir donné l'opportunité de travailler sur cette thématique, de m'avoir guidé et aidé tout au long de ce stage.

Je saisis cette occasion pour aussi adresser mes profonds remerciements à Clément, Maria, Mathieu, Pierre et Valentin pour l'aide, le cadre de travail et les parenthèses.

Enfin, je remercie toute l'équipe Sémagramme pour l'accueil chaleureux.

Table des matières

Introduction

Le traitement automatique du langage naturel (TALN) est une discipline qui prend racine au début des années 1950 dans la confrontation américano-soviétique avec les débuts de la traduction automatique. Le TALN, c'est l'étude et la modélisation de la communication linguistique par le biais de la machine. C'est-à-dire étudier un langage qui n'est pas défini par une grammaire formelle avec une machine à calculer. Travailler sur des questions du TALN, c'est tenter de mieux comprendre la communication et les mécanismes cognitifs qui lui sont liés. C'est également construire des outils qui permettent de traiter et produire de l'information.

Avec l'arrivée récente de quantités de données textuelles et informatisées, l'objet d'étude du TALN apparaît en figure de proue. Comment traiter de l'information non structurée ? C'est une question ouverte et qui fait intervenir de multiples disciplines comme la linguistique, la psycholinguistique, la logique, l'informatique théorique, l'intelligence artificielle ou encore les mathématiques.

Le traitement de la langue est divisé en plusieurs niveaux :

phonétique L'analyse des sons utilisés dans la communication parlée ;

morphologique L'étude des types et la forme des mots ;

syntaxique L'étude de la combinaison des mots pour former des phrases ou des énoncés, c'est-à-dire de la forme de l'énoncé ;

sémantique L'étude du sens des énoncés, le signifiant ;

pragmatique L'analyse de la signification en fonction d'un contexte.

Cette présentation est ici décrite de manière séquentielle. Mais chaque niveau possède son lot d'ambiguïtés et la frontière entre les niveaux n'est pas si distincte que cela.

Ce stage s'inscrit dans le cadre du travail sur la relation entre le niveau syntaxique et le niveau sémantique. Comment donner du sens à des combinaisons de mots ?

La figure 1 illustre une représentation de ce lien. Elle présente d'un côté la syntaxe de la phrase " *Who does Paul think John said Bill liked* " et l'une des traductions sémantiques possibles.

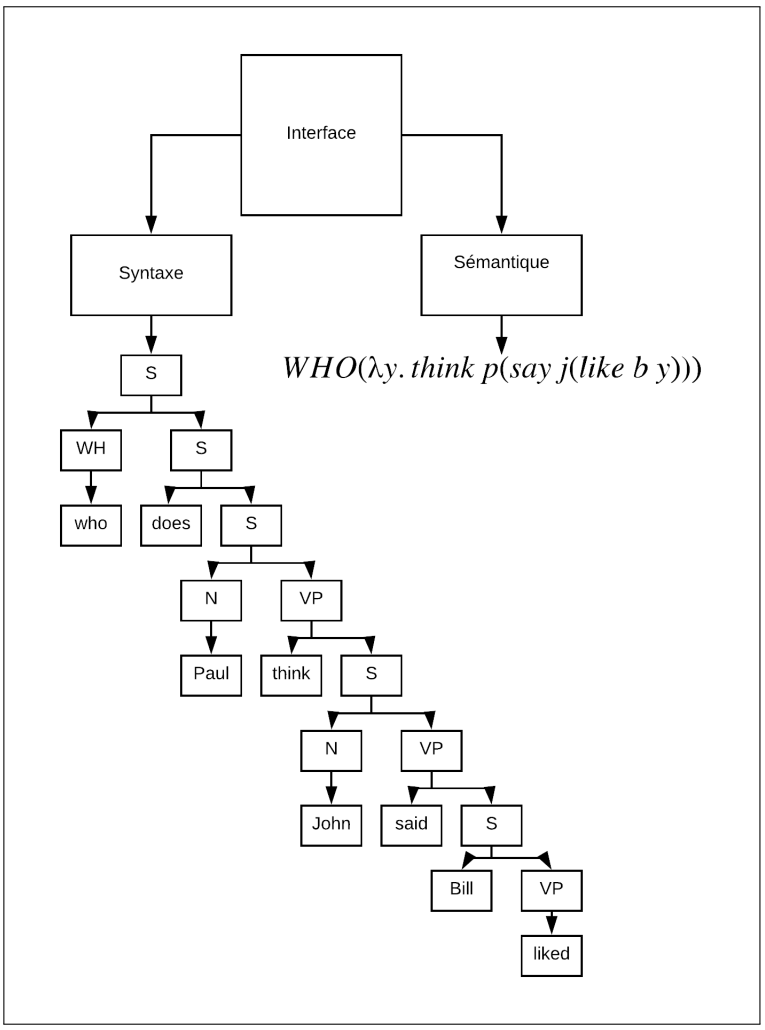


FIGURE 1 – Interface syntaxe-sémantique

Pour pouvoir travailler sur la dimension syntaxique et par extension le niveau sémantique, il est nécessaire d'avoir un outil qui permet de mettre en évidence la structure des phrases. C'est le rôle de l'analyseur syntaxique. Pour en définir un, il est nécessaire d'avoir un formalisme grammatical. Il en existe un grand nombre, parmi eux, on peut citer les grammaires d'arbres adjoints (TAG) et plus récemment les grammaires catégorielles abstraites (ACG) développées au sein de l'équipe Sémagramme. Ces cadres diffèrent, mais ont des points communs qui permettent d'établir des correspondances.

À partir d'un formalisme, on peut définir une grammaire. L'un des objectifs du travail sur la syntaxe est de produire des grammaires à large couverture, les plus fidèles possible aux langues. C'est un travail très complexe et coûteux. Du côté des TAG, ces ressources existent pour le français et l'anglais notamment. Pour les ACG, pour le moment, il n'existe pas de grammaire à large couverture.

L'objectif de ce stage est donc d'utiliser les similarités entre les deux formalismes et les ressources existantes des TAG pour reproduire ces grammaires de manière automatique pour les ACG. Ce rapport définit dans un premier temps les formalismes grammaticaux, explicite le principe de la traduction TAG en ACG, puis décrit l'environnement logiciel dans lequel le stage se situe et enfin décrit le développement du logiciel TAG2ACG qui implémente cette traduction.

1 Laboratoire Lorrain de recherche en informatique et ses applications

Le laboratoire lorrain de recherche en informatique et ses applications (LORIA)[1] est une unité mixte de recherche. Les différents organismes qui le composent sont L'Institut National de Recherche en Informatique et Automatique (INRIA)[2], le Centre National de Recherche Scientifique (CNRS)[3] et l'Université de Lorraine[4]. Il est situé en périphérie de Nancy à Vandœuvre-lès-Nancy. Le Loria compte 31 équipes de recherche divisées en 5 départements pour un total de plus de 290 chercheurs et plus de 400 personnes en effectif total.

1.1 Présentation des départements

Chaque département traite d'un aspect particulier de la recherche en informatique actuelle.

Algorithmique, calcul, image et géométrie Ce département s'articule autour des différentes facettes des problèmes algorithmiques en particulier dans le cadre de la géométrie ou du calcul symbolique.

Méthodes formelles Cette division s'intéresse à différentes notions telles que la logique, les modèles de calcul et de programmation, la modélisation, la spécification, la sûreté et enfin la sécurité et la vérification.

Réseaux, systèmes et services Le travail porte sur les questions d'informatique, parallèle et distribuée, et sur le paradigme de programmation temps réel.

Traitement automatique des langues et des connaissances Trois concepts y sont étudiés : la langue naturelle, les connaissances et les documents. Du côté de la langue naturelle, la recherche est centrée sur :

— La parole ;

1. www.loria.fr/fr/
2. www.inria.fr/fr
3. www.cnrs.fr
4. www.univ-lorraine.fr

— La syntaxe ;
— La sémantique ;
— Le discours.

Le travail sur les connaissances porte sur :

— Sa représentation ;
— La formalisation du raisonnement ;
— La découverte de connaissance.

Quant aux documents, les thématiques sont :

— La reconnaissance de formes et symboles ;
— La modélisation de l'écriture manuelle.

Systèmes complexes, intelligence artificielle et robotique Ce département a pour objectif d'étudier les systèmes complexes et leurs interactions, l'intelligence artificielle et la robotique.

1.2 Équipe Sémagramme

Sémagramme [5] est une équipe du département quatre qui s'intéresse à la linguistique computationnelle. L'objectif global est la conception de modèles et d'outils logiques pour l'analyse de la sémantique de la langue naturelle. Au moment de mon stage, elle était composée de cinq permanents, trois doctorants et trois stagiaires.

1.2.1 Axes de recherche

Sémagramme s'organise autour de trois axes de recherche :

— La modélisation de l'interface syntaxe-sémantique ;
— La dynamique des discours ;
— La construction de ressources lexicales et grammaticales.

Modélisation de l'interface syntaxe-sémantique

L'objectif global de l'équipe Sémagramme est de travailler sur la représentation sémantique de la langue naturelle. Or, la construction de la sémantique est guidée par la syntaxe et il existe une multitude de formalismes pour la représenter. Il est donc nécessaire d'avoir un cadre générique qui permet de travailler sur ces différents formalismes. L'autre principe directeur, énoncé notamment par Montague (Montague 1970) ; considère que la sémantique est une image homomorphe de la syntaxe. C'est à partir de cela que le formalisme des ACG a été introduit.

Dynamique des discours

Une des questions du traitement automatique des langues est comment interpréter un discours. Si on prend une partie du discours, on doit pouvoir l'analyser à partir d'un

5. http://semagramme.loria.fr/

contexte mais cette partie a également le pouvoir de modifier ce contexte. Il semble donc que l'interprétation d'un discours possède une notion de dynamique. En effet, la représentation sémantique s'appuie généralement sur des représentations logiques, notamment si l'on s'intéresse aux propriétés inférentielles véhiculées par la langue. Lorsque l'on passe de l'analyse sémantique d'une phrase seule à l'analyse d'une suite de phrases, ou discours, de nouveaux phénomènes apparaissent, en particulier ceux liés à la dynamique de ce discours et à l'évolution des modèles. Ainsi, dans la phrase "il dort", l'interprétation du pronom ne peut se faire qu'à l'aide d'une interprétation des phrases précédentes qui construit non seulement une représentation à l'aide d'une formule logique, mais aussi un contexte dans lequel des entités peuvent être référencées par la suite.

Construction de ressources

De par la nature de construction de la sémantique, il est nécessaire d'avoir à disposition des ressources morphosyntaxiques comme des grammaires et des lexiques. Sémagramme a produit diverses ressources notamment des grammaires et des corpus annotés. Le présent travail s'inscrit dans le cadre de la construction de ressources pour les ACG.

2 Contexte scientifique

Dans ce chapitre, nous portons notre regard sur les TAG, l'entrée de notre programme. Ensuite, nous explicitons les différents constituants des ACG, le point de sortie. Cela nécessite avant tout chose de faire une brève explication du lambda-calcul puisque c'est un élément central des ACG. Enfin, nous explicitons l'encodage des TAG dans les ACG qui est la principale fonction du logiciel TAG2ACG.

2.1 Grammaires d'arbres adjoints

Les grammaires d'arbres adjoints (Aravind K. Joshi, Levy et Takahashi 1975) sont un formalisme qui engendre un langage faiblement dépendant du contexte situé entre les grammaires algébriques et les grammaires contextuelles dans la hiérarchie de Chomsky (Chomsky 1958). Il est fondé sur l'utilisation d'arbres et de réécritures d'arbres. Il est notamment utilisé en traitement automatique des langues et linguistique formelle pour traiter principalement le niveau syntaxique. C'est un système bien étudié tant d'un point de vue linguistique que computationnel. En effet, il existe des grammaires à large couverture fabriquées avec ce formalisme et son utilisation dans le cadre de l'analyse syntaxique est de complexité polynomiale.

2.1.1 Définitions

Définition 2.1.1.1. Grammaire TAG
Une grammaire TAG est définie par un quintuplet $\langle \Sigma, NT, I, A, S \rangle$ où :

- $\boldsymbol{\Sigma}$ L'ensemble fini de symboles terminaux ;
- $\boldsymbol{NT}$ L'ensemble fini de symboles non terminaux ;
- $\boldsymbol{I}$ L'ensemble des arbres initiaux ;
- $\boldsymbol{A}$ L'ensemble des arbres auxiliaires ;
- $\boldsymbol{S}$ Un symbole de départ appartenant à NT.

Remarque 2.1.1.1. Nomenclature des arbres
Pour parler des arbres, nous utilisons la nomenclature suivante :

- α_{ancre} Arbre initial associé au mot, ou *ancré* par le mot *ancre* ;
- β_{ancre} Arbre auxiliaire associé au mot, ou *ancré* par le mot *ancre* ;
- **Arbre élementaire** Arbre initial ou arbre auxiliaire ;
- **Arbre dérivé** Arbre engendré par une ou plusieurs opérations entre arbres élémentaires ;

Arbre de dérivation Arbre qui explicite la construction d'un arbre dérivé.

Définition 2.1.1.2. Arbre initial
Un arbre initial est un arbre élémentaire qui possède ces trois propriétés :

Propriété 1.
Les nœuds feuilles ont un label dans Σ ou NT.

Propriété 2.
Les nœuds feuilles avec un label dans NT sont marqués avec une flèche $\downarrow$ pour l' opération de substitution définie plus tard.

Propriété 3.
Les nœuds qui ne sont pas des feuilles ont un label dans NT.

La figure 2.1 présente un exemple d'arbre initial ancré par *mange*. Celui-ci possède deux feuilles : NP dans NT marqué par une flèche et son ancre dans Σ.

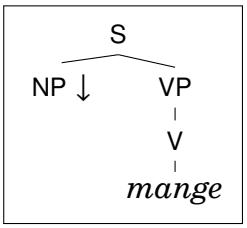


FIGURE 2.1 – Arbre initial α_{mange}

Définition 2.1.1.3. Arbre auxiliaire
Un arbre auxiliaire est un arbre élémentaire qui possède ces quatre propriétés :

Propriété 4.
Les nœuds feuilles ont un label dans Σ ou NT.

Propriété 5.
Il existe un nœud étiqueté par un symbole non terminal appelé nœud pied marqué avec *. Ce nœud se doit d'être de la même catégorie que la racine.

Propriété 6.
Les nœuds feuilles avec un label dans NT sont marqués avec $\downarrow$ pour la substitution, à l'exception du nœud pied.

Propriété 7.
Les nœuds qui ne sont pas des feuilles ont un label dans NT.

La figure 2.2 montre un exemple d'arbre auxiliaire ancré par *apparemment*. Celui-ci possède deux feuilles le nœud pied VP^* et son ancre.

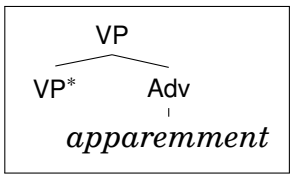


FIGURE 2.2 – Arbre auxiliaire $\beta_{apparemment}$

Une grammaire TAG est munie de deux opérations : la substitution et l'adjonction.

Définition 2.1.1.4. Substitution
La substitution est une opération qui construit un nouvel arbre à partir d'un arbre α et d'un arbre δ.
α doit être un arbre initial ou un arbre dérivé d'un arbre initial. δ doit être un arbre élémentaire ou un arbre dérivé.
Si α possède un nœud k avec un label X marqué avec $\downarrow$ et si δ a pour label à sa racine X alors on remplace le nœud k par l'arbre δ.

La figure 2.3 donne un exemple de substitution où α est l'arbre ancré par *mange* et δ l'arbre ancré par *Jean*.

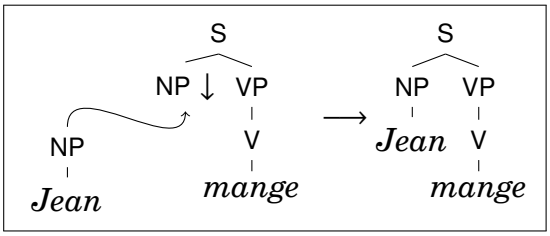


FIGURE 2.3 – Substitution

Définition 2.1.1.5. Adjonction
L'adjonction est une opération qui construit un nouvel arbre à partir d'un arbre auxiliaire β et d'un arbre initial ou dérivé α.
Il n'est pas possible de faire une adjonction sur un nœud marqué $\downarrow$.
Si on considère un arbre β avec à sa racine un label noté X et un arbre α qui contient un nœud k avec pour label X qui n'est pas marqué par la substitution alors on remplace le sous-arbre γ au nœud k de l'arbre α par l'arbre β auquel on a substitué l'arbre γ à son nœud pied.

La figure 2.4 donne un exemple où β est l'arbre ancré par *apparemment*, α est l'arbre ancré par *mange*, et γ est le sous-arbre de α au nœud VP

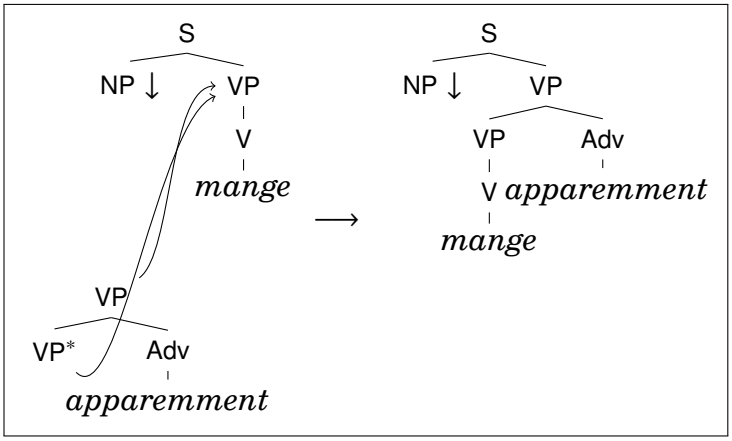


FIGURE 2.4 – Adjonction

Définition 2.1.1.6. Contraintes d'adjonction
Il est possible de donner des contraintes concernant l'adjonction sur tous les nœuds des arbres élémentaires. On peut spécifier 3 types de contraintes :

Adjonction nulle Il n'est pas possible de procéder à une adjonction sur ce nœud ;

Adjonction obligatoire Il est nécessaire de procéder à une adjonction ;

Adjonction sélective Il est possible de procéder à une adjonction quand l'arbre appartient à un ensemble A.

Définition 2.1.1.7. Arbre de dérivation
Un arbre de dérivation permet de spécifier comment l'arbre dérivé a été construit.

Les arbres de dérivation possèdent 3 propriétés :

Propriété 8.
Un arbre de dérivation a pour label à sa racine un arbre initial.

Propriété 9.
Tous les nœuds ont pour label un arbre auxiliaire ou initial

Propriété 10.
Les branches contiennent les adresses des nœuds où ont eu lieu les opérations.

La figure 2.5 montre un exemple d'arbre de dérivation pour la substitution présentée figure 2.3.

La figure 2.6 montre l'arbre de dérivation pour l'adjonction présentée figure 2.4

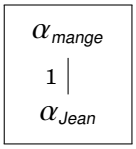


FIGURE 2.5 – Arbre de dérivation pour la substitution

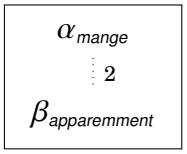


FIGURE 2.6 – Exemple arbre de dérivation adjonction

2.1.2 Variantes des TAG

Définition 2.1.2.1. LTAG
Une grammaire est lexicalisée (SCHABES et Aravind K. JOSHI 1988) si et seulement si :
- Les structures (un ensemble fini) sont associées avec un élément lexical ;
- Il existe une ou des opérations pour effectuer des compositions avec les structures.

Chaque élément lexical sera appelé l'ancre de la structure.

Définition 2.1.2.2. FB-TAG ou FTAG
Il est aisé d'ajouter les propriétés des grammaires d'unification dans le formalisme des TAG , l'idée est de remplacer les symboles NT ou Σ par des structures de traits. Ces structures de traits sont des ensembles de paires attribut-valeur. Cela permet de rajouter des contraintes autres que la catégorie syntaxique comme le genre ou le nombre sur la lexicalisation et la construction d'arbres dérivés.

Dans le cas des TAG, chaque nœud non terminal présente deux structures de traits : les traits amont et les traits aval. Les structures de traits permettent d'affiner la description des arbres syntaxiques mais cela ne change pas l'expressivité en terme de hiérarchie de Chomsky.

Nous présentons figure 2.7 un exemple d'arbre avec structures de traits. Chaque nœud possède une paire d'attribut-valeur pour la catégorie syntaxique (cat), une structure de traits amont (top) et une structure de trait aval (bot). Dans ces structures, on retrouve des choses comme le nombre (num), la personne (pers).

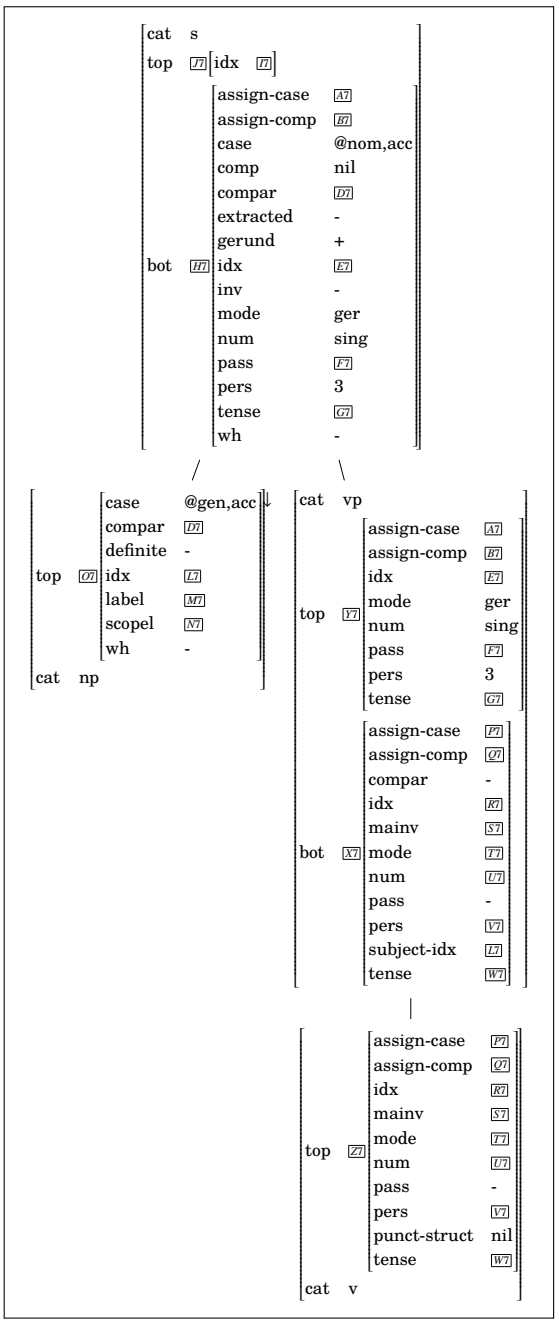


FIGURE 2.7 – α_{mange} avec structures de traits, sans le nœud terminal mange

Unification

Dans le cas des FB-TAG, il est nécessaire d'avoir une opération d'unification pour travailler avec ce formalisme. De manière simplifiée, l'unification est une opération qui combine des structures de traits et en crée une nouvelle qui contient les informations précédentes des parties et rien de plus. Il est possible qu'une unification échoue, c'est donc une opération partielle.

Conséquences sur les opérations

Substitution Les traits amont de la racine de l'arbre substitué s'unifient avec les traits amont du nœud de substitution.

Adjonction Les traits amont de la racine de l'arbre auxiliaire s'unifient avec les traits amont du nœud adjoint. Les traits aval du nœud pied sont unifiés avec les traits aval du nœud adjoint.

2.2 Lambda-calcul

Le lambda-calcul est un formalisme introduit par Alonzo Church (Church 1936). Il s'articule autour de la notion de fonction et de calcul. Il est équivalent aux machines de Turing et est très utilisé dans le domaine de la calculabilité.

2.2.1 Lambda-calcul non typé

Définition 2.2.1.1. Syntaxe du lambda-calcul
La syntaxe du lambda-calcul peut être décrite sous la forme de Backus-Naur suivante :

$$M ::= x|\lambda x.\, M|M\, N$$

où :

x est une variable ;

λx. M est une abstraction avec M un terme et x est dit lié dans M ;

M N est une application avec M et N des termes.

Exemple 1. $I = \lambda x.\, x$ correspond à la fonction identité.

Exemple 2. $K = \lambda xy.\, x$ prend deux arguments et renvoie le premier.

Définition 2.2.1.2. Substitution
L'opération de substitution se définit de manière inductive comme ceci :

— $a[N/a] \equiv N$
— $b[N/a] \equiv b$
— $(L\, M)[N/x] \equiv (L[N/x])(M\,[N/x])$
— $(\lambda x.\, M)\,[N/x] \equiv \lambda x.\, M$
— $(\lambda y.\, M)\,[N/x] \equiv \lambda y.\, M\,[N/x]$ avec y non libre dans N

Définition 2.2.1.3. bêta-réduction
La bêta-réduction est une opération qui permet de remplacer une variable liée par le paramètre auquel est appliqué le terme fonctionnel.
$((\lambda x.\, M))\, N \longrightarrow_\beta M\,[N/x]$

la définition de l'alpha-conversion qui stipule que l'on peut renommer les variables liées comme on le souhaite et l'η-équivalence qui permet de retirer une abstraction redondante ne sont pas présentées dans ce rapport.

2.2.2 Lambda-calcul simplement typé

Cette sous-section présente le λ-calcul simplement typé. Il est nécessaire d'introduire dans le langage : les variables linéaires, l'application linéaire et l'abstraction linéaire qui sont issues de la logique linéaire (Girard 1987), et enfin les constantes, car elles sont présentes dans le formalisme des ACG. L'intuition derrière la logique linéaire est d'avoir un moyen de spécifier l'usage des ressources.

Dans le λ-calcul typé, la notion de contexte de typage est essentielle. Un contexte de typage, c'est une suite de déclarations de types et de variables comme ceci : $x : \alpha$. On distingue dans notre système deux contextes distincts. La notation Γ est utilisée pour le contexte non-linéaire et Δ pour le linéaire.

Pour définir un système de typage, il est nécessaire d'introduire le séquent un objet de la forme $\Gamma; \Delta \vdash t : \alpha$. Il est alors possible de construire des règles d'inférences pour chaque catégorie de lambda-termes :

$$\frac{}{\Gamma; \vdash_\Sigma c : \tau(c)}\ \text{(const.)}$$

$$\frac{}{\Gamma; x : \alpha \vdash_\Sigma x : \alpha}\ \text{(lin. var.)} \qquad \frac{}{\Gamma, x : \alpha; \vdash_\Sigma x : \alpha}\ \text{(var.)}$$

$$\frac{\Gamma; \Delta, x : \alpha \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda^{\circ} x.t : \alpha \multimap \beta}\ \text{(lin. abs.)} \qquad \frac{\Gamma; \Delta_1 \vdash_\Sigma t : \alpha \multimap \beta \quad \Gamma; \Delta_2 \vdash_\Sigma u : \alpha}{\Gamma; \Delta_1, \Delta_2 \vdash_\Sigma (t\, u) : \beta}\ \text{(lin. app.)}$$

$$\frac{\Gamma, x : \alpha; \Delta \vdash_\Sigma t : \beta}{\Gamma; \Delta \vdash_\Sigma \lambda x. .t : \alpha \to \beta}\ \text{(abs.)} \qquad \frac{\Gamma; \Delta \vdash_\Sigma t : \alpha \to \beta \qquad \Gamma; \vdash_\Sigma u : \alpha}{\Gamma; \Delta \vdash_\Sigma (t\, u) : \beta}\ \text{(app.)}$$

Table 2.1 – Règles de typage

Ces types portent de l'information. Cela permet notamment d'avoir une idée sur ce que fait une fonction et permet de vérifier qu'une composition de fonction est possible. Outre cet aspect, cela permet également de s'assurer que les calculs terminent. En effet, si on prend le terme $\Omega = \lambda x.\, xx$ pour lequel la réduction $\Omega\Omega \to \Omega\Omega$ montre qu'il existe une réduction infinie, il n'est pas possible de le typer.

Exemple 3. Soit $x, y, estNegatif$ et $aleatoirePositif$ des λ-termes et $\mathbb{N}$ (Entiers), $\mathbb{B}$ (Booléens) des types.

estNegatif renvoie vrai si l'entier est négatif et *aleatoirePositif* renvoie un entier positif si la valeur est vrai sinon renvoie un entier négatif.

— ; $estNegatif : \mathbb{N} \to \mathbb{B} \vdash estNegatif : \mathbb{N} \to \mathbb{B}$ (var.)
— ; $x : \mathbb{N} \vdash x : \mathbb{N}$ (var.)
— ; $y : \mathbb{B} \vdash y : \mathbb{B}$ (var.)
— ; $aleatoirePositif : \mathbb{B} \to \mathbb{N} \vdash$: $aleatoirePositif : \mathbb{B} \to \mathbb{N}$ (var.)

on a ainsi :

— ; $estNegatif : \mathbb{N} \to \mathbb{B};\ x : \mathbb{N} \vdash (estNegatif\ x) : \mathbb{B}$ (app.)
— ; $AleatoirePositif : \mathbb{B} \to \mathbb{N};\ y : \mathbb{B} \vdash (aleatoirePositif\ y) : \mathbb{N}$ (app.)
— ; $estNegatif : \mathbb{N} \to \mathbb{B}; (aleatoirePositif\ y) : \mathbb{N} \vdash (estNegatif\ (aleatoirePositif\ y)) : \mathbb{B}$ (app.)
— $(estNegatif\ y)$ n'est pas typable. Il faudrait utiliser (app.) ou (lin. app.) mais les prémisses ne sont pas typables étant donnée les hypothèses sur les types de *estNegatif* et y.

2.3 Grammaires catégorielles abstraites

Les grammaires catégorielles abstraites (De Groote 2001) sont un cadre grammatical qui permet d'encoder et d'étendre des formalismes grammaticaux de manière uniforme comme les TAG par exemple. Il permet, par le biais de la vérification de type et par la composition, d'être modulable et flexible. Pour l'aborder, il est nécessaire de définir ses différents constituants.

2.3.1 Constituants d'une ACG

Définition 2.3.1.1. Signature d'ordre supérieur
Une signature d'ordre supérieur est un triplet (A, C, τ) où :

A Un ensemble de types atomiques ;

C Un ensemble de constantes ;

τ Une fonction qui pour chaque constante associe un type.

L'ensemble des lambda-termes construits à partir des constantes de C, des variables, de l'application et des abstractions est noté $\Lambda\Sigma$.

Définition 2.3.1.2. Types implicatifs
l'ensemble des types implicatifs $\mathscr{T}_A$ est défini de la manière suivante :

— Si $a \in A$ alors $a \in \mathscr{T}_A$;
— Si $\alpha, \beta \in \mathscr{T}_A$ alors $\alpha \to \beta \in \mathscr{T}_A$;
— Si $\alpha, \beta \in \mathscr{T}_A$ alors $\alpha \multimap \beta \in \mathscr{T}_A$ (implication linéaire, utilise une seule fois l'hypothèse α).

Définition 2.3.1.3. Lexique
Un lexique est une paire (F,G) où :

F : $\mathscr{T}_{A_1} \to \mathscr{T}_{A_2}$ est un morphisme entre les types construits sur A_1 et les types construits sur A_2 : $F(\alpha \to \beta) = F(\alpha) \to F(\beta)$;

$G : \wedge(\Sigma_1) \rightarrow \wedge(\Sigma_2)$ est un morphisme entre les termes construits sur σ_1 et les termes construits sur $\sigma_2 : G(x) = x, G(\lambda x.t) = \lambda x.G(t), G(t\ u) = G(t)\ G(u), G(\lambda^o x.t) = \lambda^o x.G(t)$;

F et G tels que $\forall c \in C_1, \vdash_{\Sigma_2} G(c) : F(\tau_1(c))$ est prouvable. C'est-à-dire que le type de l'interprétation d'un c abstrait, c'est l'interprétation du type de c.

2.3.2 ACG

Définition 2.3.2.1. ACG
Une grammaire abstraite catégorielle (notée $\mathscr{G}$) est un quadruplet $\langle \Sigma_1, \Sigma_2, \mathscr{L}, s \rangle$ où :

Σ_1 une signature d'ordre supérieur ;

Σ_2 une autre signature d'ordre supérieur ;

$\mathscr{L}$ un lexique ;

s un type $\in \mathscr{T}_{A_1}$.

2.3.3 Langages

Une ACG génère deux langages (ensembles de lambda-termes) le langage abstrait et le langage objet.

Définition 2.3.3.1. Langage abstrait
Le langage abstrait noté $\mathscr{A}(\mathscr{G})$ représente les structures admissibles, il est défini comme l'ensemble des $t \in \wedge(\Sigma_1)$ où $\vdash_{\Sigma_1} t : s$ est dérivable.

Définition 2.3.3.2. Langage objet
Le langage objet noté $\mathscr{O}(\mathscr{G})$ représente la réalisation des structures admissibles définie par le langage abstrait, il est défini comme l'ensemble des $u \in \wedge(\Sigma_2)$ où $\exists t \in \mathscr{A}(\mathscr{G})$ tel que $u = \mathscr{L}(t)$.

Composition des ACG
Le lexique définit comment les structures sont interprétées. Cela permet d'une part, de faire en sorte qu'on puisse définir deux interprétations issues d'un même vocabulaire abstrait. Cela est notamment utilisé pour modéliser l'interface entre la syntaxe et la sémantique. Un exemple de ce partage est représenté figure 2.8 où $\Sigma_{derivations}$ (représentant les arbres de dérivations des TAG) est le langage abstrait de la grammaire $\mathscr{G}_{derived\ trees}$ (syntaxe) et celui de $\mathscr{G}_{sem.}$ (sémantique) .

D'autre part, cela permet d'effectuer des compositions de lexiques. C'est-à-dire que la première interprétation devient le vocabulaire abstrait de la seconde interprétation. Cela permet notamment d'avoir de la modularité et un moyen de contrôler les structures générées. La figure 2.8 montre un exemple de composition de lexique avec $\mathscr{G}_{yield} \circ \mathscr{G}_{derived\ trees}$.

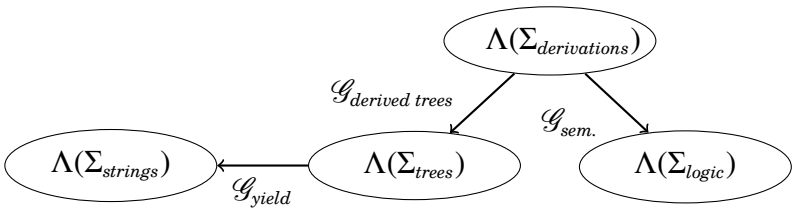


FIGURE 2.8 – Modélisation de l'interface syntaxe-sémantique

2.4 Encodage des TAG dans les ACG

Cette section présente la conversion des TAG en ACG (POGODALLA 2017). Nous abordons en premier lieu la représentation des chaînes, puis la construction des arbres dérivés et enfin, les arbres de dérivations.

2.4.1 Encodage des chaînes

La nomenclature utilisée pour la signature des chaînes est $\Sigma_{strings}$. Sa définition est $\Sigma_{strings} = \langle A_\sigma, C, \tau_\sigma \rangle$ avec :

A un ensemble composé d'un seul élément o, un type atomique ;

τ_σ une fonction qui pour chaque élément de C attribue le type $(o \multimap o)$;

C un alphabet.

un type σ qui n'est pas atomique est défini tel que : $\sigma = (o \multimap o)$.

On se munit de deux opérateurs qui n'appartiennent pas à la signature la concaténation de chaînes et la fonction identité :
— $+ = \lambda^o f\ g\ z.f\ (g\ z) : \sigma \to \sigma \to \sigma$
— $\epsilon = \lambda^o x.x : \sigma \to \sigma$
on peut démontrer que ϵ est l'élément neutre de + et que + est associatif :

Démonstration. ϵ l'élément neutre de +
on pose $a : \sigma$.

on cherche à montrer que : $a + \epsilon = \epsilon + a$

$$\begin{aligned}
\epsilon + a &= + \epsilon\ a \\
&= (\lambda^o f\ g\ z.f\ (g\ z))\ \epsilon\ a \\
&= (\lambda^o g\ z.\epsilon\ (g\ z))\ a \\
&= (\lambda^o g\ z.(\lambda^o x.x)\ (g\ z))\ a \\
&= (\lambda^o g\ z.g\ z)\ a \\
&= \lambda^o z.a\ z
\end{aligned}$$

et :

$$
\begin{aligned}
a + \epsilon &= + \; a \; \epsilon \\
&= (\lambda^o f \; g \; z.f \; (g \; z)) \; a \; \epsilon \\
&= (\lambda^o g \; z.a \; (g \; z)) \; \epsilon \\
&= \lambda^o z.a \; (\epsilon \; z) \\
&= \lambda^o z.a \; ((\lambda^o x.x) \; z) \\
&= \lambda^o z.a \; z
\end{aligned}
$$

Par η-équivalence la fonction a est égale à la fonction $\lambda z.\, a\, z$, donc :

$a + \epsilon = \epsilon + a = a$

Lemme. $(f + g)\ h \rightarrow_\beta f\ (g\ h)$
Pour tout $f, g : \sigma$ et $h : o$, $(f + g)\ h \rightarrow_\beta f\ (g\ h)$

On le montre ici :

$$
\begin{aligned}
(f + g)\ h &= (+ \; f \; g)\ h \\
&= ((\lambda^o F \; G \; Z.F \; (G \; Z)) \; f \; g)\ h \\
&= ((\lambda^o G \; Z.f \; (G \; Z)) \; g)\ h \\
&= (\lambda^o Z.f \; (g \; Z))\ h \\
&= f \; (g \; h)
\end{aligned}
$$

Démonstration. Associativité de +
on pose f, g et $h : \sigma$
on cherche à montrer que : $f + (g + h) = (f + g) + h$

on a :

$$
\begin{aligned}
f + (g + h) &= (+) \; f \; (+ \; g \; h) \\
&= (\lambda^o F \; G \; z.F \; (G \; z)) \; f \; (+ \; g \; h) \\
&= (\lambda^o G \; z.f \; (G \; z)) \; (+ \; g \; h) \\
&= \lambda^o z.f \; ((+ \; g \; h) \; z) \\
&= \lambda^o z.f \; (g \; (h \; z))
\end{aligned}
$$

et :

$$
\begin{aligned}
(f + g) + h &= (+) \; (+ \; f \; g)\ h \\
&= (\lambda^o F \; G \; z.F \; (G \; z)) \; (+ \; f \; g)\ h \\
&= (\lambda^o G \; z.(+ \; f \; g) \; (G \; z))\ h \\
&= \lambda^o z.(+ \; f \; g) \; (h \; z) \\
&= \lambda^o z.f \; (g \; (h \; z))
\end{aligned}
$$

donc $f + (g + h) = (f + g) + h$

On utilise le lemme précédent pour passer à la dernière réduction dans les deux cas.
$(h\ z)$ est bien de type o dans la seconde partie.

2.4.2 Encodage des arbres dérivés

On appelle la signature des arbres dérivés Σ_{trees}.

On pose T un type atomique qui représente les arbres.

On utilise un alphabet indicé. C'est-à-dire que les non terminaux sont marqués avec leur nombre de fils. Les nœuds correspondent à notre ensemble de constantes.

La figure 2.9 montre un exemple. la racine S possède deux fils, on la note donc S_2.VP et V n'ont qu'un fils. Ils deviennent respectivement VP_1 et V_1 . *mange* est un terminal et un élément du lexique, son arité est 0. NP est une feuille et est marqué par une substitution.

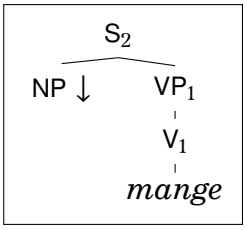
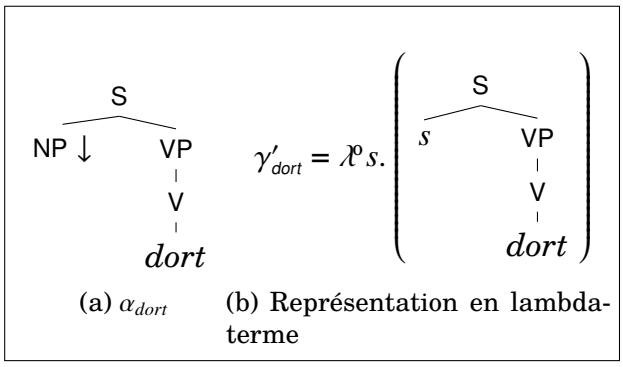


FIGURE 2.9 – Exemple : Indice α_{mange}

Au niveau du typage, les nœuds terminaux ont le type T et les non terminaux une implication linéaire de n+1 fois le type T.

Il reste les cas spécifiques d'un nœud où il est possible d'avoir une substitution ou une adjonction.

Substitution Quand un arbre contient un nœud de substitution. Il peut être vu comme une fonction qui prend un autre arbre à la place de ce nœud. La figure 2.10 montre l'exemple d'un arbre qui contient un nœud VP marqué par la substitution et son interprétation.

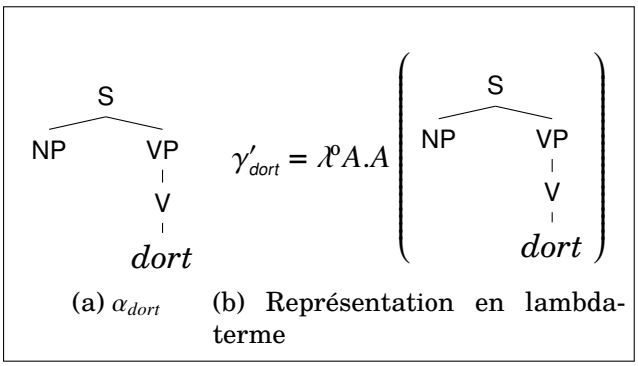


(a) α_{dort} (b) Représentation en lambda-terme

FIGURE 2.10 – Interprétation fonctionnelle de l'opération de substitution sur le nœud NP

Un arbre avec juste un non-terminal (substitution possible) avec une contrainte d'adjonction nulle sera donc typé comme ceci : $T \multimap T$.

Adjonction Quand un arbre contient un nœud n où il est possible d'effectuer une adjonction. Il peut être vu comme une fonction qui prend une fonction de type $T \multimap T$ appliquée au sous-terme dont la racine est le nœud n. La figure 2.11 l'illustre avec le nœud n = S.



(a) α_{dort} (b) Représentation en lambda-terme

FIGURE 2.11 – Interprétation fonctionnelle de l'opération d'adjonction sur le nœud S

Il est également nécessaire d'introduire la fonction identité de type $T \multimap T$ pour ne pas rendre obligatoire l'adjonction.

Un arbre avec juste un nœud où l'adjonction est possible sera donc typé comme ceci : $(T \multimap T) \multimap T$.

Lexique Il est possible de construire le lexique qui va des arbres dérivés vers les chaînes en faisant en sorte qu'il renvoie simplement la concaténation des feuilles des arbres. Par exemple, s'il existe un nœud non terminal S_2, sa déclaration dans le lexique est $S_2 := \lambda xy.\, x + y$.

2.4.3 Gestion des catégories syntaxiques

Dérivations

L'explication précédente montre comment on peut construire les arbres dérivés. La problématique est qu'on ne prend pas en compte la catégorie syntaxique des nœuds et on souhaite modéliser les arbres de dérivation. Pour traiter cela, on définit une nouvelle signature $\Sigma_{derivations}$.

Les types de cette signature sont les catégories syntaxiques.

Une fonction identité est définie pour chaque type et une constante pour chaque arbre.

Le typage d'une constante c_{ancre} se fait de la manière suivante :
$(A^1 \multimap A^1) \multimap \cdots \multimap (A^n \multimap A^n) \multimap S^1 \multimap \cdots \multimap S^m \multimap \alpha$
avec

- A^i Les noms des n nœuds non terminaux où une adjonction est possible. La convention explicitée dans les articles de l'ordre est issue du parcours en largeur ;
- S^i les noms des m feuilles avec un non terminal, à l'exception du nœud pied si l'arbre est auxiliaire. L'ordre est défini par un parcours en profondeur (droite-gauche) ;
- α Il existe une distinction entre deux cas, on note R la catégorie de la racine de l'arbre :
  - Si la constante est issue d'un arbre initial alors $\alpha = R$;
  - Sinon, on note R' la catégorie du nœud pied et $\alpha = R' \multimap R$.

Lexique Pour construire le lexique qui va de $\Sigma_{derivations}$ vers Σ_{trees}, chaque type de notre signature devient le type T. Les fonctions identités ont pour terme objet : la fonction identité. Les constantes des arbres ont pour terme objet, la représentation d'un arbre en lambda-calcul.

$$
\begin{array}{ll}
c_{dort} & : (\mathsf{S} \multimap \mathsf{S}) \multimap (\mathsf{VP} \multimap \mathsf{VP}) \multimap (\mathsf{V} \multimap \mathsf{V}) \multimap \mathsf{NP} \multimap \mathsf{S} \\
:=_{\text{derived trees}}\ \gamma_{dort} & \quad = \lambda^{0} S\ a\ b\ s.S\ (\mathsf{S}_2\ s\ (a\ (\mathsf{VP}_1\ (b\ (\mathsf{V}_1\ dort)))))) \\
 & : (T \multimap T) \multimap (T \multimap T) \multimap (T \multimap T) \multimap T \multimap T
\end{array}
$$

FIGURE 2.12 – Inteprétation d'une constante de $\Sigma_{derivations}$

Ordre et Dérivations

Grâce à $\Sigma_{derivations}$, les catégories syntaxiques sont prises en compte. Mais un problème reste encore en suspens. Il est impossible, par exemple, de différencier les constantes abstraites issues d'un arbre initial avec un nœud de substitution S de même type que celui de la racine et celle d'un arbre auxiliaire contenant une racine S (et donc un nœud pied de même type).

Il est possible de déterminer l'ordre des ACG à partir du maximum de l'ordre du type des constantes abstraites et la complexité à partir de l'ordre maximum de la réalisation de ses types atomiques. On peut donc constituer une hiérarchie. Celle-ci a été étudiée et nous informe sur les propriétés d'une ACG. Dans notre contexte, celle qui nous intéressent sont les ACG d'ordre 2 qui ont un parsing polynomial. Or, dans nos signatures courantes, les constantes dépassent l'ordre 2.

Ce sont les raisons pour lesquelles, on introduit une nouvelle signature Σ_{TAG}.

Pour tous les arbres initiaux, on définit une constante C_{ancre} tel que : $A_A^1 \multimap \cdots \multimap A_A^n \multimap S^1 \multimap \cdots \multimap S^m \multimap \alpha$
avec

- A_A^i Les noms des n nœuds non terminaux où une adjonction est possible. La convention explicitée dans les articles de l'ordre est issue du parcours en largeur ;
- S^i les noms des m feuilles avec un non-terminal, à l'exception du nœud pied si l'arbre est auxiliaire. L'ordre est défini par un parcours en profondeur (droite-gauche) ;
- α Il existe une distinction entre deux cas, on note R la catégorie de la racine de l'arbre :
  - Si la constante est issue d'un arbre initial alors $\alpha = R$;
  - Sinon, on note R' la catégorie du nœud pied et $\alpha = R'_A$.

Il est nécessaire d'associer pour chaque type X_A une fonction identité de type X_A pour ne pas rendre l'adjonction obligatoire. La distinction entre les arbres est désormais possible.

3 Environnement de travail

Dans ce chapitre, l'environnement de travail du stage est introduit. Les explications concernent tout d'abord, l'environnement de programmation de TAG2ACG, ensuite les différents logiciels qui sont en relation avec lui sont détaillés et enfin une présentation des ressources grammaticales est effectuée. L'objectif est donc de présenter le passage du cadre scientifique vers les implémentations matérielles.

3.1 Environnement de programmation

3.1.1 Ocaml

Ocaml [1] est un langage de programmation écrit par Xavier Leroy, Damien Doligez, Jérôme Vouillon et Didier Rémy au sein de L'INRIA. Il fait partie de la famille des langages ML (Méta-langage). Il est multiparadigme et est utilisé principalement de manière fonctionnelle. Ses caractéristiques principales sont :

— Le typage statique ;
— L'inférence de types ;
— Le polymorphisme paramétrique ;
— Le filtrage par motif.

C'est le langage utilisé pour écrire le programme TAG2ACG.

3.1.2 XML

Le XML [2] est un langage de balisage générique créé au sein de la W3C. Sa création a été menée par Jon Bosak. Il permet de structurer de l'information en général et en particulier textuelle. Il est :

— Standard dans le monde de l'informatique ;
— Défini par une syntaxe stricte ;
— Utile pour décrire des structures arborescentes.

Les ressources linguistiques à notre disposition sont encodées dans ce format. On en présente une description dans la section 3.3

1. https://ocaml.org/
2. https://www.w3.org/XML/

3.1.3 GNU Make

GNU Make [3] est un moteur de production créé par Roland McGrath et Richard Stallman. Il permet l'automatisation des opérations répétitives comme la génération de documentation ou bien la compilation. Il est indépendant du langage de programmation et est basé sur le prince de règles et de cibles.

On l'exploite pour faire la liaison entre la conversion effectuée par le programme et la boîte à outils Acgtk décrite dans la sous-section 3.2.3.

3.1.4 Dune

Dune [4] est un moteur de production spécifique à Ocaml et Reason. Il a été conçu par Jérémie Dimino. Il est utilisé pour la compilation des sources de TAG2ACG.

3.1.5 Git

Le code source du programme TAG2ACG est hébergé sur la forge Gitlab Inria [5] qui utilise Git comme gestionnaire de version.

Git [6] est un logiciel crée par Linus Torvalds en 2005 qui permet de gérer des versions de manière décentralisée. Il est principalement dédié à la gestion de code source et au travail collaboratif.

3.2 Environnement logiciel

Pour comprendre les actions du programme TAG2ACG, il est nécessaire de présenter les deux logiciels qui génèrent les entrées du programme, à savoir XMG et Lex2all. Ensuite, il faut s'intéresser à l'utilisation de ses sorties qui sont des fichiers utilisables par Acgtk. Avec ces précisions, on peut décrire de manière plus précise le fonctionnement du programme.

3.2.1 XMG

Une des problématiques lorsqu'on construit une grammaire à large couverture de manière manuelle, c'est la maintenance. Comment opérer sur un ensemble de descriptions syntaxiques complètes et complexes ? Une des approches possibles est de construire un outil de haut niveau qui permet non pas de décrire la totalité, mais de définir des fragments. On introduit des contraintes pour pouvoir les composer et cela génère de manière automatique l'ensemble des descriptions à partir d'éléments

3. https://gnu.org/software/make/
4. https://dune.readthedocs.io/en/latest/
5. https://gitlab.inria.fr/
6. https://git-scm.com/

beaucoup moins complexes. Ce genre d'outil propose donc un méta langage et un compilateur. Il existe plusieurs implémentations de ce principe.

Parmi eux, on peut trouver XMG (Crabbé et al. 2013) qui est un compilateur de métagrammaires qui permet de construire des grammaires TAG de tailles conséquentes pour le langage naturel. Il est possible de définir des grammaires TAG avec structures de traits et une sémantique basée sur la logique des prédicats. La sortie de ce programme est un fichier XML qui décrit la totalité des descriptions. C'est une entrée de notre programme.

3.2.2 Lex2all

Une partie des ressources dont nous disposons sont issues de ce XMG. Celles-ci ne présentent toutefois pas de lexique. C'est-à-dire que l'on a un ensemble de descriptions syntaxique et sémantique, mais celles-ci ne sont pas associées à des mots. Certaines ressources possèdent un lexique associé mais dans un autre format.

Lex2all est un programme écrit par Yannick Parmentier et Eric Kow qui permet de prendre une description de lexiques qui est lisible pour un humain et de la transformer vers différents types de fichiers notamment une description XML qu'on utilise dans TAG2ACG. Avec ces deux ressources liées, on obtient une grammaire complète.

3.2.3 Acgtk

L'objectif global de ce stage est de convertir des TAG en ACG mais aussi de pouvoir manipuler ces ressources dans ce cadre. On peut par exemple vouloir procéder à l'analyse syntaxique d'une phrase à partir des ressources converties. Pour cela, on dispose de l'outil Acgtk.

Acgtk [7] (Pogodalla 2016) est une boîte à outils développée au sein de l'équipe Sémagramme par Sylvain Pogodalla pour le test et le développement des ACG. Il comprend deux logiciels :

— Acgc ;
— Acg.

Acg

Acg est un interpréteur qui permet de travailler avec ce cadre. Il dispose de fonctionnalités comme l'analyse ACG qui permet de retrouver les termes abstraits possibles à partir d'un terme objet. On peut par exemple vouloir connaître les arbres dérivés possibles de la phrase "Paul dort". Cela se fait par le biais de cette analyse. On peut également faire l'opération inverse qu'on appellera la réalisation qui à partir du langage abstrait produit l'image dans le langage objet. Par exemple, on peut vouloir déterminer la représentation logique associée à un arbre de dérivation.

7. http://calligramme.loria.fr/acg/

Acgc

Acgc est un compilateur qui vérifie la construction des signatures et lexiques ACG et produit un binaire. Dans le cadre de ce stage, on l'utilise pour vérifier que la conversion produite est cohérente. C'est également très utile dans le lancement d'Acg puisqu'il ne reproduira pas cette étape avec un fichier déjà compilé. Ce qui peut engendrer selon la taille des ressources un gain de temps conséquent.

3.2.4 TAG2ACG

TAG2ACG c'est donc le programme qui permet de convertir des fichiers représentant des FB-LTAG en ACG. Il prend en entrée un fichier XML de lexique et un fichier XML qui représente la grammaire TAG et renvoie un ensemble de fichiers représentant une architecture ACG qui correspond à la conversion des TAG en ACG. Si l'on intègre les explications précédentes concernant l'environnement logiciel, on peut voir TAG2ACG comme un membre d'une chaîne de traitement (voir figure : 3.1).

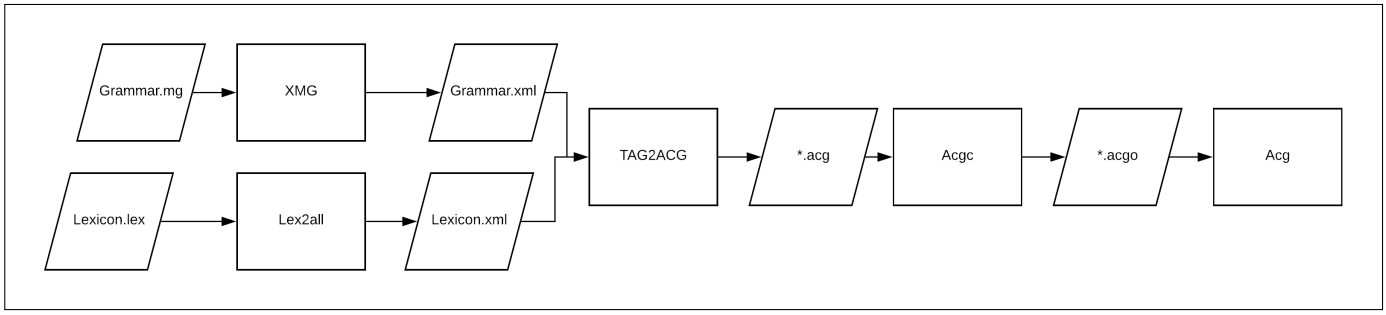


FIGURE 3.1 – TAG2ACG, une interface entre plusieurs programmes

Une première version a été réalisée par Boussidan Aaron, un stagiaire précédent de l'équipe Sémagramme, qui depuis le fichier de grammaire construit une partie de la conversion (voir figure : 3.2). La grammaire sans lexique est convertie pour produire $\mathscr{G}_{TAG}$ et $\mathscr{G}_{derived\ trees}$

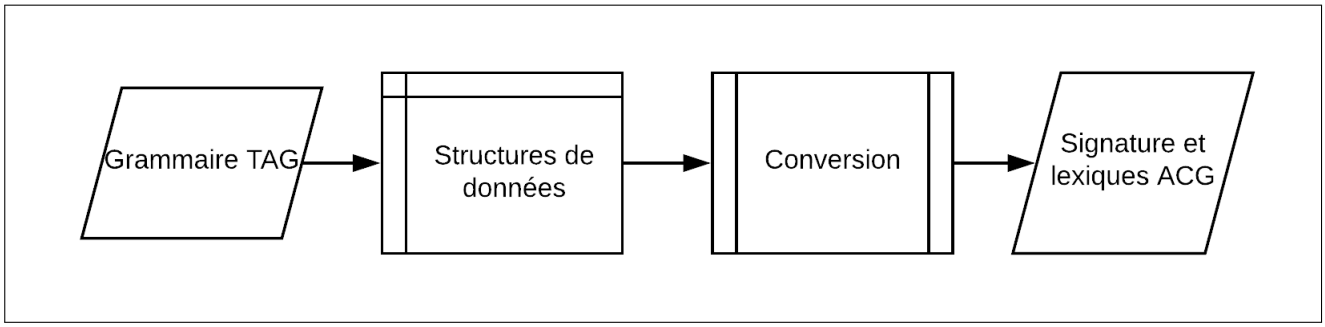


FIGURE 3.2 – Architecture de la première version

La version développée durant ce stage construit une partie plus conséquente de la conversion en prenant en compte le lexique (voir figure : 3.3). On ajoute également $\mathscr{G}_{yield}$. Le développement est explicité dans le chapitre 4.

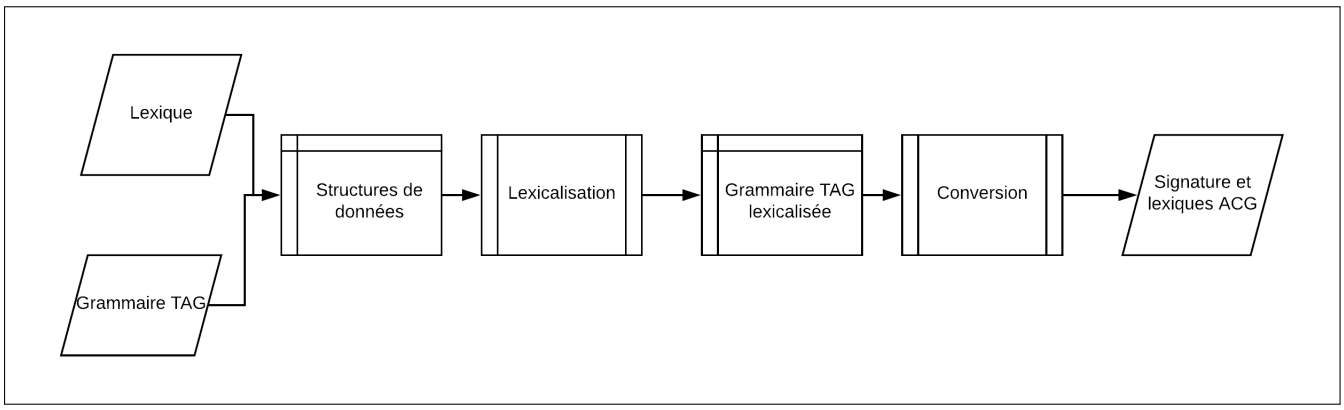


FIGURE 3.3 – Architecture de la version courante

3.3 Ressources linguistiques

On présente dans cette section l'encodage XML des entrées de TAG2ACG. Il est nécessaire, ici, de l'aborder sans pour autant le couvrir complètement dans la mesure où cela permet de mieux comprendre ce qui a été réalisé.

On introduit quelques conventions pour faciliter la lecture des figures :

***** Contient 0 ou n nœuds XML de ce type ;

+ Contient 1 ou n nœuds XML de ce type ;

? Contient 0 ou 1 nœud XML de ce type ;

| Il n'y a qu'un nœud XML parmi les nœuds de la même hauteur marqués par ce préfixe qui est utilisé. C'est un ou exclusif ;

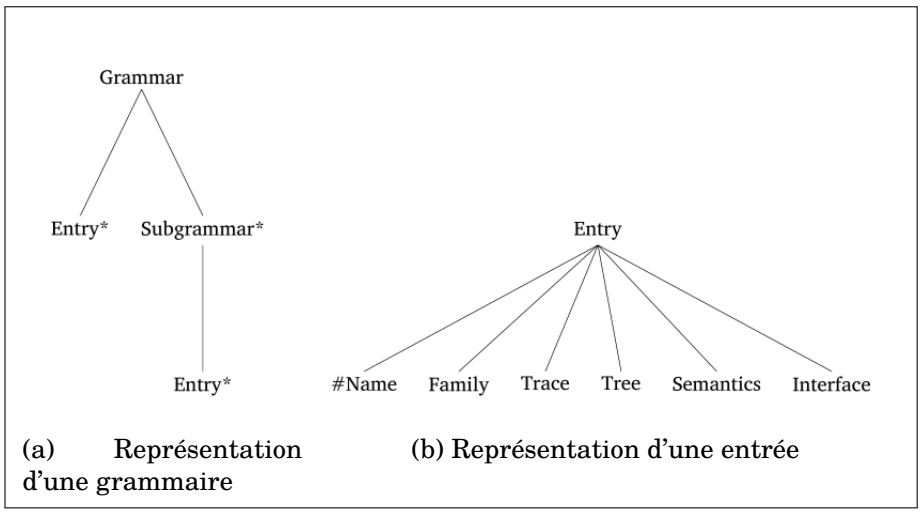
Les attributs d'un nœud sont marqués par ce préfixe.

Pour une description plus fine, on introduit la définition formelle des documents XML (pour la grammaire, voir figure : 5.2 et pour le lexique, voir figure : 5.1). Il n'est cependant pas nécessaire de s'y attarder pour comprendre la suite de ce rapport.

3.3.1 Grammaire

Une grammaire (voir figure : 3.4a) est un ensemble d'entrées ou un ensemble de sous-grammaires.

Une entrée (voir figure : 3.4b) est ce qui enveloppe la description syntaxique et sémantique. Elle porte également des informations sur son appartenance à une famille, et comment elle a été construite. Une famille est un ensemble d'arbres qui partagent des caractéristiques syntaxiques communes.



(a) Représentation d'une grammaire (b) Représentation d'une entrée

FIGURE 3.4 – Représentation partielle du XML des grammaires sous forme arborescente

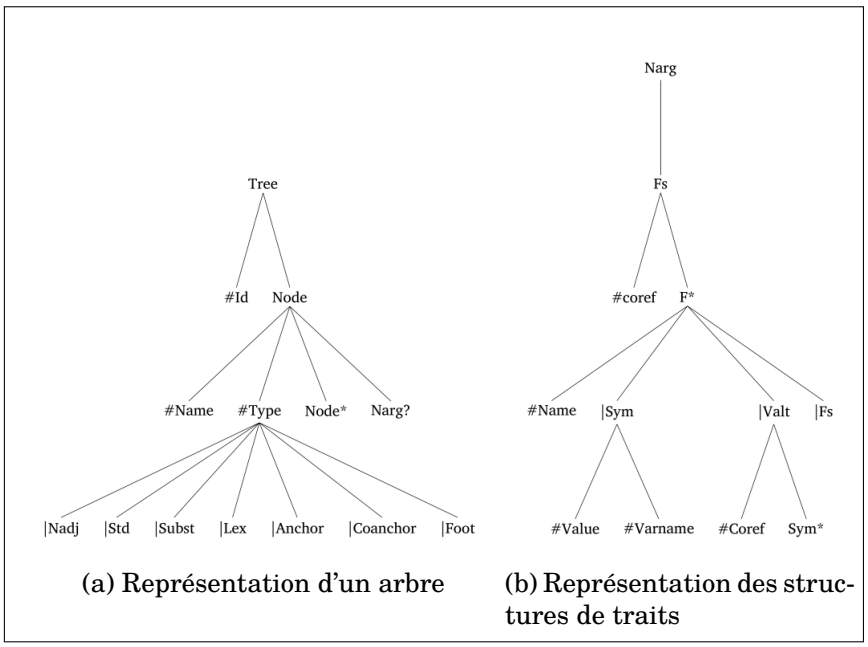
La description syntaxique se fait par le biais du nœud XML arbre (voir figure 3.5a). Celle-ci est composée d'un identifiant et d'une racine qui est un nœud (en tant que composant d'un arbre et non d'un nœud XML). Celui-ci possède un nom et un type qui appartient aux différentes catégories de nœuds présentes dans les TAG. Ce type permet notamment de distinguer s'il est un terminal ou non et s'il peut subir une opération. Ce nœud peut contenir une liste d'enfants (définition récursive).

Il présente également une structure pour contenir les structures de traits qu'on appelle Narg ici (voir figure : 3.5b). Narg contient une structure de traits (Fs) qui enveloppe un ensemble de traits (F).

Un trait (F) peut être soit une structure de traits , soit un trait atomique (Sym) ou bien un ensemble de traits atomiques (Valt). Cela permet de représenter l'incertitude. Par exemple , on peut reprendre l'exemple de Text Encoding Initiative [8] qui définit un trait de genre qui contient masculin, féminin et neutre.

Cette façon de définir les structures de traits correspond à la figure 2.7 présentée dans la description des TAG avec structures de traits.

8. http://www.tei-c.org/release/doc/tei-p5-doc/en/html/ref-vAlt.html



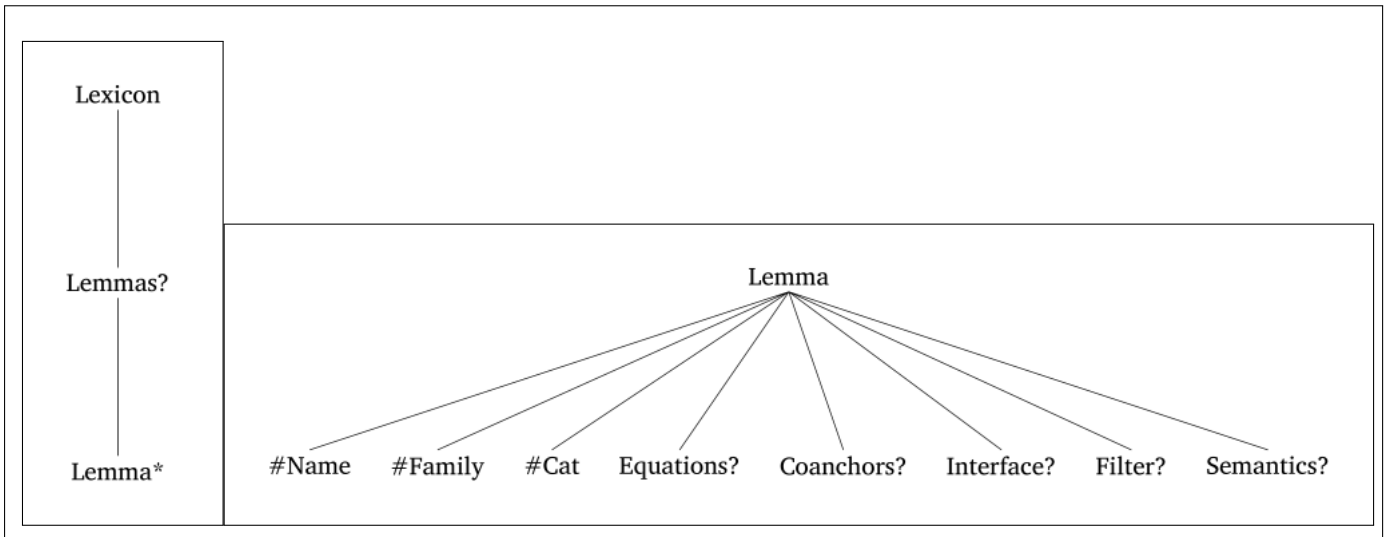
(a) Représentation d'un arbre

(b) Représentation des structures de traits

FIGURE 3.5 – Représentation partielle du XML de la description syntaxique sous forme arborescente

3.3.2 Lexique

Un lexique est un ensemble de lemmes (voir figure : 3.6a). Chaque lemme est ancré à une famille d'arbres (voir figure : 3.6b). Il possède une catégorie syntaxique et une nomenclature. Il dispose de potentielles coancres (des terminaux qui lui sont liés) qui lui sont associées et une sémantique. On omet intentionnellement ici les éléments de filtrage et d'interfaces.



(a) Représentation d'un lexique

(b) Représentation d'un lemme

FIGURE 3.6 – Représentation partielle du XML des lexiques sous forme arborescente

4 Travail réalisé

Les différentes étapes de ce stage et ses limites sont décrites dans ce chapitre.

4.1 Appréhension des formalismes et Ocaml

Pour pouvoir travailler sur le logiciel TAG2ACG, il a été nécessaire de comprendre les différents formalismes décrits précédemment. Cela a nécessité un travail d'environ 2 semaines. Pour les TAG, les principales sources sont (Abeillé 1993 ; Aravind K. Joshi, Levy et Takahashi 1975 ; Aravind K Joshi 1991), pour les ACG et enfin pour l'encodage des TAG (De Groote 2001 ; Pogodalla 2015 ; Pogodalla 2017).

Après avoir compris les bases théoriques nécessaires dans lesquelles le stage s'inscrit. Il a été essentiel d'apprendre le langage de programmation dans lequel la première ébauche a été construite. C'est-à-dire OCaml. Je me suis basé sur le livre "Real World OCaml" de (Hickey, Madhavapeddy et Minsky 2014).

4.2 Travail sur l'existant et appréhension des ressources grammaticales

Après avoir étudié le cadre scientifique et le langage du programme, il a été nécessaire de comprendre le code source du programme et les ressources linguistiques. La problématique majeure a été de saisir l'organisation des ressources et comment elles s'articulaient. En effet, elles diffèrent selon les versions et sont très peu documentées. Ce fût une tâche quelque peu déroutante dans la mesure où cela se rapproche de la rétro conception.

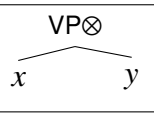
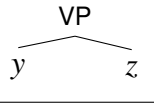
4.3 Reconstruction des lambda-termes

Ma première tâche de développement fut de trouver pourquoi la compilation ACG ne marchait pas avec la première version du programme et de résoudre les différents problèmes liés à cette erreur de compilation. Il s'avère que c'était les lambda-termes dans le lexique qui va de $\Sigma_{derivations}$ à Σ_{trees} qui ne correspondaient pas avec le typage et plus précisément l'ordre de déclaration des arguments dans les abstractions.

En effet, si on prend :

$f = \lambda x\ y.x\ y$ et $g = \lambda y\ x.x\ y$
le typage de g et f est différent.

Il a donc été nécessaire de modifier cette construction. Si on s'appuie sur le tableau de conversion des nœuds ci-dessous avec x une variable fraîche [1].

Type de noeud	Exemple	Lambda-terme associé
Lexème	*si*	*si*
Substitution	V ↓	$\lambda x.\ x$
Noeud pied	V*	$\lambda x.\ x$
Ancre et coancre	V◇	$\lambda x.\ \mathsf{V}_1 x$
Adjonction nulle		$\mathsf{VP}_2\ x\ y$
Adjonction		$\lambda x.x\ \ (\mathsf{VP}_2\ y\ z)$

Il faut que les variables introduites soient déclarées dans un ordre particulier pour correspondre à la façon de typer les arbres décrits dans la sous-section 2.4.3.

De plus, les constantes construites à partir des lexèmes n'étaient pas déclarées dans la signature ACG correspondante. C'est-à-dire qu'une partie des nœuds terminaux n'existaient pas dans Σ_{trees}. Ce qui signifie que les lambda-termes faisaient un appel à des constantes non définies. Cela déclenche une erreur fatale de Acg. Il a donc été nécessaire de les extraire et de les rajouter.

4.4 Ancrage

À ce moment du stage, le programme est capable de transformer une grammaire sans lexique en signature et lexique ACG, sans problème de compilation. Or, ce qui est intéressant dans les ressources TAG présentes, c'est qu'elles sont lexicalisées. Il a donc fallu considérer les lexiques dans la conversion.

4.4.1 Parsing

En premier lieu, il faut récupérer les structures de données du lexique XML. Pour cela, on utilise la librairie OCaml PXP [2] qui est déjà utilisée pour les grammaires. Cette librairie permet la validation d'un XML conformément à un DTD et la récupération sous forme d'objets des structures du XML. Après avoir écrit ce module, on récupère les informations qui nous intéressent à savoir le nom du lemme, la famille à laquelle il appartient et les coancres associées.

1. Une variable qui n'est pas liée ou libre.
2. `http://projects.camlcity.org/projects/dl/pxp-1.2.9/doc/manual/html/ref/index.html`

4.4.2 Intégration

La problématique est désormais de pouvoir lier les lemmes à un ensemble d'arbres. On dispose pour cela de deux informations cruciales : la famille et les coancres.

On commence d'abord, par associer chaque lemme l avec sa famille, c'est-à-dire ensemble F d'arbres. De telle sorte qu'on a un ensemble de paires $\langle l, F \rangle$.

Après avoir fait cette première association, on regarde si le lemme dispose de coancres associées et on filtre parmi F les arbres qui contiennent exactement le même nombre de coancres et si cela correspond bien.

On crée ensuite pour chaque arbre et lemme, une constante dans Σ_{TAG} et $\Sigma_{derivations}$ et on donne leur interprétation dans les lexiques correspondants.

4.5 Construction du langage des chaînes

Après avoir introduit les lemmes, on dispose d'une grammaire lexicalisée dans le format des ACG : $\mathscr{G}_{TAG}$ et $\mathscr{G}_{derived\ trees}$ (voir figure : 4.1).

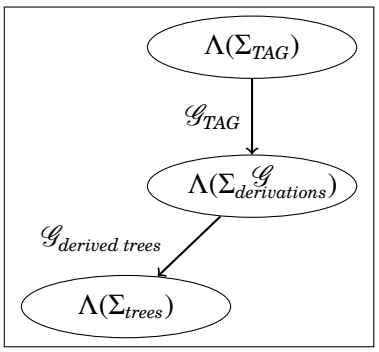


FIGURE 4.1 – Encodage de TAG2ACG sans les chaînes

On peut donc vérifier si oui ou non une phrase appartient à la grammaire.

La difficulté c'est que pour le moment, nous sommes obligés de donner la description d'une phrase en arbre pour voir s'il existe un antécédent.

Exemple 4. Si on souhaite vérifier que la phrase "Jean mange" existe on doit donner quelque chose comme "S_2 (N_1 *Jean*) (VP_1 *mange*) ". On souhaiterait pouvoir s'abstraire des catégories syntaxiques, en donnant, une chaîne de caractère comme "$Jean + mange$".

C'est la raison pour laquelle, on m'a demandé d'ajouter le langage des chaînes dans la conversion de TAG2ACG. De telle sorte qu'on a désormais $\mathscr{G}_{yield}$ (voir figure : 4.2).

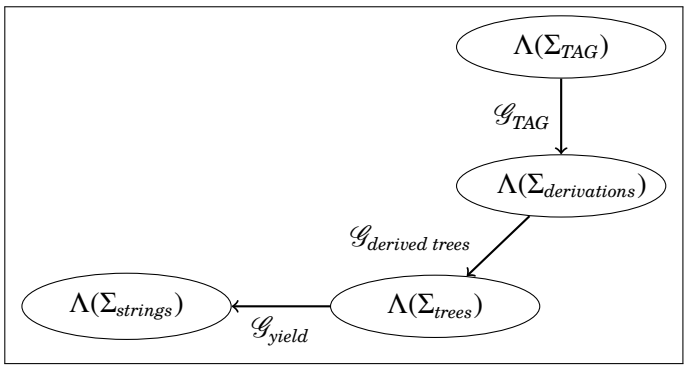


FIGURE 4.2 – Encodage actuel de TAG2ACG

4.6 Scripts ACG

Maintenant que le langage des chaînes est introduit, il est aisé de tester si une phrase appartient à la grammaire. Pour certaines ressources, on dispose d'une suite de tests dans un format conçu pour un autre logiciel. J'ai donc converti manuellement ces fichiers de tests en scripts Acg (voir figure 4.3). Cela permet notamment de vérifier la couverture de la grammaire et qu'il n'y a pas d'oubli dans la conversion.

```
load o minimal.acgo;

list;
select tag_yields;


# Parse
parse Bill + sleep: s;
parse a + cake + sleep: s;
parse Bill + eat + a + cake:s;
parse Bill + eat + a + free + cake :s;
```

FIGURE 4.3 – Exemple de script Acg

4.7 Recherche de nouvelles ressources

Après quelques recherches sur XMG, il s'est avéré que des grammaires plus grosses étaient disponibles dans un format compatible avec TAG2ACG pour le français et l'anglais notamment. Cela a permis de tester le comportement de TAG2ACG sur des fichiers de plus de 80 méga-octets, mais aussi de Acgc et Acg.

4.8 Comparateur de grammaires

Les grammaires font plusieurs millions de lignes. Il est impossible de les comparer de manière manuelle. On m'a donc demandé de construire un programme qui permet de le faire automatiquement. Celui-ci effectue des opérations ensemblistes basiques : intersection et différence sur deux grammaires et renvoie un rapport en Markdown.

Néanmoins, après une recherche ultérieure plus approfondie, j'ai trouvé qu'un outil existait déjà et qui est toujours maintenu. Il est donc préférable d'utiliser celui-ci [3].

4.9 Limites

4.9.1 Ressources

Dans certains cas, les lemmes sont associés à des familles inexistantes dans les ressources. Ce qui fait que la conversion est partielle. La suite de tests de la ressource n'est donc pas validée totalement.

Il reste à déterminer si l'absence de catégorie syntaxique dans certains nœuds signifie que les données sont incomplètes ou si cela peut être interprété comme toutes les catégories syntaxiques possibles.

Il serait sans doute utile d'avoir à disposition un outil qui effectue une normalisation des données.

Il existe un format XMG pour les réalisations morphologiques des lemmes. Malheureusement, je n'ai pas été en mesure de trouver des ressources suffisantes pour être exploitables. Le programme n'est donc pas capable de les utiliser.

4.9.2 Librairie Acgtk

Pendant la durée de mon stage, une librairie a été construite à partir du code source de Acgtk par Monsieur Pogodalla. Le programme devrait l'intégrer pour la construction des signatures et lexiques ACG et éviter le "duplicata" de code.

4.9.3 Structures de traits

Le programme n'est, à l'heure actuelle pas capable de faire, par exemple, la différence entre un nom et un nom propre. Les structures de traits ne sont, pour le moment pas considérées. L'ajout de cette fonctionnalité entraîne une modification structurelle assez forte du programme. Un premier travail sur la réalisation d'un *parser* complet pour les fichiers de grammaire et lexique a été réalisé. Il n'est toutefois pas incorporé dans la version actuelle et n'est pas terminé. De plus, l'utilisation des structures de traits dans les ACG est un travail actuel de recherche.

3. `http://dokufarm.phil.hhu.de/xmg/doku.php?` (dans la rubrique *tools*)

5 Bilan

Ce stage a été une expérience très enrichissante. Il m'a permis de découvrir un domaine totalement inconnu pour moi la recherche académique à travers différents aspects. D'abord, la nature du stage demande l'appréhension d'un contenu théorique assez important, ce qui m'a permis de me confronter à la littérature scientifique. Ensuite, j'ai eu la possibilité d'assister à un ensemble de conférences, de séminaires, colloquia et de groupes de travail sur la médiation scientifique. Enfin, cela m'a donné l'occasion d'être en contact direct avec les acteurs du domaine.

Cela confirme mon envie de m'orienter vers cette branche, notamment en traitement automatique des langues qui est un domaine riche tant sur le plan des approches que de l'intersection de différentes disciplines et donc par extension de poursuivre en master de traitement automatique des langues à Nancy.

D'un point de vue purement informatique, il m'a permis de découvrir l'écosystème Ocaml, de consolider mes connaissances en outils GNU/Linux et en format de données XML mais aussi d'approcher des fondements théoriques de l'informatique comme les paradigmes de programmation, le typage et un modèle de représentation du calcul.

J'ai également pu en apprendre plus sur les niveaux linguistiques du traitement automatique des langues notamment la syntaxe et la sémantique. J'ai découvert une des manières de construire des ressources linguistiques et l'importance de sa documentation.

Pour conclure, cela m'a permis de mettre en perspective et prendre du recul concernant les compétences développées au sein de la licence MIASHS. J'ai eu la possibilité de mettre en application et d'utiliser ce que j'avais vu en traitement automatique des langues. Ce qui a été étudié en logique et en mathématiques m'a permis d'appréhender le lambda-calcul et les formalismes grammaticaux. Enfin, j'ai pu mettre en œuvre les compétences informatique acquises pendant ma formation tout au long de ce stage.

Bibliographie

Abeillé, Anne (1993). “Les nouvelles syntaxes : grammaires d’unification et analyse du français”. In : *Collection Linguistique. Armand Colin,* DOI : 10.1017/S0959269500002787.

Chomsky, Noam (1958). “Three Models for the Description of Language”. In : *Journal of Symbolic Logic* 23.1, p. 71–72.

Church, Alonzo (1936). “An Unsolvable Problem of Elementary Number Theory”. In : *American Journal of Mathematics* 58.2, p. 345–363. DOI : 10.2307/2371045.

Crabbé, Benoît et al. (2013). “XMG : eXtensible MetaGrammar”. In : *Computational Linguistics* 39.3, p. 591–629. URL : https://hal.archives-ouvertes.fr/hal-00768224.

De Groote, Philippe (2001). “Towards Abstract Categorial Grammars”. In : *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. ACL ’01. Toulouse, France : Association for Computational Linguistics, p. 252–259. DOI : 10.3115/1073012.1073045.

Girard, Jean-Yves (1987). “Linear logic”. In : *Theoretical Computer Science* 50.1, p. 1–101. ISSN : 0304-3975. DOI : 10.1016/0304-3975(87)90045-4.

Hickey, Jason, Anil Madhavapeddy et Yaron Minsky (2014). *Real World OCaml*. ISBN : 144932391. URL : http://www.worldcat.org/isbn/144932391.

Joshi, Aravind K (1991). “Unification-Based Tree Adjoining Grammars Unification-Based Tree Adjoining Grammars”. In : March. URL : https://repository.upenn.edu/cis_reports/762/.

Joshi, Aravind K., Leon S. Levy et Masako Takahashi (1975). “Tree adjunct grammars”. In : *Journal of Computer and System Sciences* 10.1, p. 136–163. DOI : 10.116/S0022-0000(75)80019-5.

Montague, Richard (1970). “Universal grammar”. In : *Theoria* 36.3, p. 373–398. DOI : 10.1111/j.1755-2567.1970.tb00434.x.

Pogodalla, Sylvain (2015). “Functional Approach to Tree-Adjoining Grammars and Semantic Interpretation : an Abstract Categorial Grammar Account”. working paper or preprint.

– (2016). “ACGTK : un outil de développement et de test pour les grammaires catégorielles abstraites”. In : URL : https://hal.inria.fr/hal-01328702.

– (2017). “A syntax-semantics interface for Tree-Adjoining Grammars through Abstract Categorial Grammars”. In : *Journal of Language Modelling*. DOI : 10.15398/jlm.v5i3.193.

Schabes, Yves et Aravind K. Joshi (1988). “An Earley-type Parsing Algorithm for Tree Adjoining Grammars”. In : *Proceedings of the 26th Annual Meeting on Association for Computational Linguistics*. ACL ’88. Buffalo, New York : Association for Computational Linguistics, p. 258–269. DOI : 10.3115/982023.982055.

Annexes

```
<!-- lexicon -->
<!ELEMENT lexicon (lemmas?)>

<!-- entries -->
<!ELEMENT lemmas (lemma*)>

<!-- lemma entry -->
<!ELEMENT lemma (equations?, coanchors?, interface?, filter?, semantics?)>
<!ATTLIST lemma  name   CDATA   #REQUIRED
                 family CDATA   #REQUIRED
                 cat    CDATA   #IMPLIED
                 acc    CDATA   #IMPLIED>

<!-- equations -->
<!ELEMENT equations (equation*)>

<!-- anchoring equation -->
<!ELEMENT equation (fs?)>
<!ATTLIST equation  node_id  CDATA  #REQUIRED
          type CDATA #REQUIRED>
<!-- coanchors -->
<!ELEMENT coanchors (coanchor*)>

<!-- coanchoring -->
<!ELEMENT coanchor (fs?)>
<!ATTLIST coanchor node_id  CDATA  #REQUIRED
          type     CDATA  #REQUIRED>

<!-- interface -->
<!ELEMENT interface (fs?)>

<!-- filters to select trees -->
<!ELEMENT filter (fs?)>

<!-- syntax/semantic interface in the lemma lexicon -->
<!ELEMENT semantics (literal*)>

<!-- literal -->
<!ELEMENT literal (args)>
<!ATTLIST literal label  CDATA #REQUIRED
        predicate CDATA #REQUIRED>

<!-- list of arguments for the instantiation -->
<!ELEMENT args (sym*)>

<!-- features -->
<!ELEMENT fs (f*)>

<!ELEMENT f (sym | vAlt | fs)>
<!ATTLIST f name CDATA #REQUIRED>

<!ELEMENT vAlt (sym+)>

<!ELEMENT sym EMPTY>
<!ATTLIST sym value   CDATA #IMPLIED
         varname CDATA #IMPLIED>
```

FIGURE 5.1 – Définition formelle du XML des lexiques

```
<!ELEMENT grammar (entry* | subgrammar*)>

<!ELEMENT subgrammar (entry*)>
<!ATTLIST subgrammar id CDATA #REQUIRED>

<!ELEMENT entry (family,trace,tree,semantics,interface)>
<!ATTLIST entry name ID #REQUIRED>

<!ELEMENT family (#PCDATA)>

<!ELEMENT trace (class*)>
<!ELEMENT class (#PCDATA)>

<!-- syntactic descriptions (the templates) -->
<!ELEMENT tree (node)>
<!ATTLIST tree id CDATA #REQUIRED>

<!ELEMENT node (narg? , node*)>
<!ATTLIST node type ( nadj | std | subst | lex | anchor | coanchor | foot ) #REQUIRED>
<!ATTLIST node name CDATA #IMPLIED>
<!ELEMENT narg (fs)>

<!ELEMENT fs (f*)>
<!ATTLIST fs coref CDATA #IMPLIED>

<!ELEMENT f (sym | vAlt | fs)>
<!ATTLIST f name CDATA #REQUIRED>

<!ELEMENT vAlt (sym*)>
<!ATTLIST vAlt coref CDATA "">

<!ELEMENT sym EMPTY>
<!ATTLIST sym
     value   CDATA #IMPLIED
     varname CDATA #IMPLIED>

<!-- semantic representations -->
<!ELEMENT semantics (literal | sym | semdominance)*>

<!ELEMENT literal (label? , predicate, arg*)>
<!ATTLIST literal negated CDATA "no">

<!ELEMENT label (sym)>
<!ELEMENT predicate (sym)>
<!ELEMENT arg (sym | fs)>

<!ELEMENT semdominance (arg+)>
<!ATTLIST semdominance op CDATA "ge">

<!ELEMENT interface (fs?)>
```

FIGURE 5.2 – Définition formelle du XML des grammaires

Rapport de stage

Grammaires d'arbres adjoints à large couverture et grammaires catégorielles abstraites

Maxime GUILLAUME

Année 2017–2018

UFD63 Stage de fin de licence réalisé dans le Laboratoire Lorrain de Recherche en Informatique et ses Applications

Maxime GUILLAUME
5, Boulevard du Recteur Senn
54000, Nancy
+33 6 73 56 01 22
maxime.guillaume2@etu.univ-lorraine.fr

UFR Mathématiques et Informatique
Bâtiment de l'ESPE , 54 Bis Bd de Scarpone,
54000, Nancy
+33 3 72 74 16 18

Laboratoire Lorrain de Recherche en Informatique et ses Applications
615, Rue du Jardin-Botanique
54506, Vandœuvre-lès-Nancy
+33 3 83 59 20 00

Maîtres de stage : Maxime Amblard et Sylvain Pogodalla

Encadrant universitaire : Sylvain Castagnos