UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR

ADAM JOLY

DU DOCUMENT TEXTUEL À SA CARTE SÉMANTIQUE FONCTIONNELLE

AOÛT 2009

CE PROJET A ÉTÉ ÉVALUÉ
PAR UN JURY COMPOSÉ DE :

M. Ismail Biskri, directeur de projet
Département de mathématiques et informatique à l'Université du Québec à Trois-Rivières

M. François Meunier, professeur
Département de mathématiques et informatique à l'Université du Québec à Trois-Rivières

M. Mhamed Mesfioui, professeur
Département de mathématiques et informatique à l'Université du Québec à Trois-Rivières

SOMMAIRE

En traitement automatique du langage naturel utilisant l'approche des grammaires catégorielles, une stratégie d'analyse incrémentale de gauche à droite pour éliminer la pseudo-ambiguïté peut mener à la formation de faux constituants. Lorsque cela se produit, des opérations de décomposition de ces faux constituants deviennent nécessaires afin que l'analyse puisse se poursuivre. Toutefois, peu de travaux ont abordé ce problème. L'une des rares publications dans laquelle une solution, en l'occurrence la décomposition, a été présentée reste celle de Steedman (2000). La décomposition est basée sur le principe de la neutralité paramétrique qu'on peut résumer comme suit : deux types catégoriels associés entre eux par une règle combinatoire (qui ne prend en compte que les types catégoriels) permettent de déterminer le troisième type. Ce principe rencontre cependant des limites importantes dans le cas de certaines formes elliptiques de la coordination pour lesquelles il peut même s'avérer être inopérant. Dans ce travail, nous proposons une autre méthode de décomposition basée principalement sur la « mémoire » intrinsèque contenue dans les combinateurs de la logique combinatoire et véhiculée lors de leur introduction dans les expressions combinatoires. Ce sont ces variations structurelles des opérandes des combinateurs **B** et $\mathbf{C}_{\star}$ lors de leur β-réduction que nous avons étudié. Nous en avons dégagé un algorithme qui calcule automatiquement comment restructurer l'expression combinatoire de la coordination afin de permettre la poursuite de l'analyse catégorielle. Une série d'exemples illustrant dans les moindres détails le fonctionnement du processus s'en suit. Aussi, une implémentation de l'algorithme a été effectuée. Elle a entre autres permis de tester davantage la robustesse de l'algorithme avec un corpus d'expressions combinatoires à complexités variables, pour lesquelles la bonne solution a toujours été trouvée. Enfin, nous souhaitons que la méthode puisse éventuellement inclure d'autres combinateurs, ce qui ne devrait pas causer problème, puisque chaque combinateur est traité indépendamment dans l'algorithme. Par ailleurs, une des prochaines étapes consisteraient à greffer ce module à un outil d'analyse catégorielle. Nous pensons aussi que notre méthode pourrait servir à prouver que deux phrases sont en fait des paraphrases.

ABSTRACT

In natural language processing using the categorial grammars, a strategy of incremental analysis from left to right to eliminate the pseudo-ambiguity often leads to the construction of spurious constituents. Whenever this happens, operations of decomposition of these false constituents become necessary so the analysis can continue. However, few works addressed this problem. One of the rare publications in which a solution, in occurrence the decomposition, has been presented is from Steedman (2000). The decomposition is based on the principle of parametric neutrality, which can be summed up as following: two categorial types linked together by a combinatory rule (which only consider categorial types) allow to infer the third type. This principle though meets important limits in the case of some elliptic forms of coordination for which it can even be inoperative. Within this work, we propose another method of decomposition mainly based on the intrinsic "memory" contained into the combinators of the combinatory logic and transported at the time of their introduction into the combinatory expressions. These are those structural variations of the operands of combinators **B** and $\mathbf{C}_*$ after a β-reduction that we studied. What emerged from this is an algorithm that calculates automatically how to restructure the combinatory expression of the coordination so the categorial analysis can continue. A series of detailed examples showing the mechanisms of the process follows. An implementation of the algorithm was also done. Among other things, it gave us the possibility to test the robustness of the algorithm further, with a corpus of combinatory expressions of different complexities, for which a solution was found every time. At last, we wish to eventually add more combinators to the method, which should not be a major problem, considering that each combinator is handled separately into the algorithm. In addition, one of the next steps would consist of using this module into a categorial analysis tool. We also thing that our approach could be useful to prove that two sentences are in fact paraphrases.

REMERCIEMENTS

J'aimerais dans un premier temps remercier profondément le Professeur Ismail Biskri pour m'avoir fait l'honneur d'être mon directeur de travaux, pour avoir été un véritable mentor depuis le baccalauréat, pour sa disponibilité, pour ses encouragements et pour son soutien.

Je désire également remercier le Professeur Boucif Amar Bensaber pour sa générosité, son soutien et les opportunités d'expériences de travail.

Des remerciements aussi à tous mes collègues du laboratoire LAMIA, aux chargés de projet informatique et technologique Robert Ouellet, Guy Therrien et Daniel St-Yves, ainsi qu'à Chantal Guimond et Manon Goulet du département de mathématiques et informatique.

Je remercie finalement ma famille, soit mon père, ma mère, mes quatre frères, ainsi que ma conjointe, pour leur soutien moral et tout l'amour qu'il me donne chaque jour.

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

CHAPITRE 1

INTRODUCTION

Le modèle de la Grammaire Catégorielle Combinatoire Applicative considère les unités de la langue comme étant des opérateurs ou des opérandes qui sont formalisées en des expressions de la logique combinatoire (Curry, 1958 ; Shaumyan, 1998). Les modèles linguistiques de la Grammaire Applicative Universelle (Shaumyan, 1998) et de son extension la Grammaire Applicative et Cognitive (Desclès, 1990, 1996) soutiennent qu'une analyse du langage doit postuler trois niveaux de représentation, dont l'objectif principal est d'étudier les propriétés formelles des systèmes linguistiques et d'en analyser les structures logiques : (i) *le niveau des structures morphosyntaxiques,* qui correspond à la représentation observable du langage; (ii) *le niveau des structures prédicatives,* qui permet d'exprimer l'interprétation sémantique fonctionnelle; et enfin (iii) *le niveau cognitif*, où la signification des prédicats linguistiques est donnée.

Une analyse dans ce modèle permet la vérification de la bonne connexion syntaxique des énoncés et la vérification explicite de l'expression morphosyntaxique à sa représentation prédicative à l'aide de combinateurs de la logique combinatoire auxquels sont associées des règles d'introduction et d'élimination (β-réduction). Nous présentons ci-contre les règles β-réduction pour les combinateurs **Φ**, **B** et $\mathbf{C}_*$, pour lesquelles u_1, u_2, u_3 et u_4 représentent des expressions applicatives typées qui agissent soient comme des opérateurs ou des opérandes:

$(\mathbf{\Phi}\ u_1\ u_2\ u_3)\ u_4 \rightarrow u_1\ (u_2\ u_4)\ (u_3\ u_4)$

$((\mathbf{B}\ u_1\ u_2)\ u_3) \rightarrow (u_1\ (u_2\ u_3))$

$((\mathbf{C}_*\ u_1)\ u_2) \rightarrow (u_2\ u_1)$

Les règles de la Grammaire Catégorielle Combinatoire Applicative utilisent des unités linguistiques avec des types orientés[1] dont la conséquence est la construction d'une expression applicative typée à laquelle peut potentiellement s'ajouter un combinateur. Le changement de type d'une unité *u* introduit le combinateur $\mathbf{C}_*$, alors que la composition de deux unités concaténées introduit le combinateur **B**.

Règles d'application :

$[X/Y : u_1]$ - $[Y : u_2]$		$[Y : u_1]$ - $[X\backslash Y : u_2]$
---------->	;	<----------
[X : (u1 u2)]		[X : (u2 u1)]

Règles de changement de type :

[X: u]		[X : u]
---------->**T**	;	<**T**----------
$[Y/(Y\backslash X) : (\mathbf{C}_* u)]$		$[Y\backslash(Y/X) : (\mathbf{C}_* u)]$

Règles de composition fonctionnelle :

$[X/Y : u_1]$ - $[Y/Z : u_2]$		$[Y\backslash Z : u_1]$ - $[X\backslash Y : u_2]$
----------><**B**	;	<**B**----------
$[X/Z : (\mathbf{B}\ u_1\ u_2)]$		$[X\backslash Z : (\mathbf{B}\ u_2\ u_1)]$

L'élaboration d'analyseurs catégoriels pose plusieurs défis importants. Parmi eux, nous retrouvons entre autre le problème de la pseudo-ambiguïté, qui se produit lorsque plusieurs arbres syntaxiques correspondent à une seule interprétation sémantique pour

[1] Les types orientés sont composés à partir de types de base N (pour syntagme nominale) et S (pour phrase) et de constructeurs avant (/) et arrière (\) qui détermine le sens d'application de la fonction.

un même énoncé. Une solution est d'effectuer une analyse incrémentale de gauche vers la droite et ainsi ne conserver qu'un seul arbre de dérivation. Toutefois, cette méthode peut mener à la formation de faux constituants. Considérons l'exemple *Charles mange son repas lentement* :

(1)	[N : Charles] - [(S\N)/N : mange] - [N : son-repas] - [(S\N)\(S\N) : lentement]	
(2)	[S/(S\N) : (**C**$_*$ Charles)] - [(S\N)/N : mange] - [N : son-repas] - [(S\N)\(S\N) : lentement]	>T
(3)	[S/N : (**B** (**C**$_*$ Charles) mange)] - [N : son-repas] - [(S\N)\(S\N) : lentement]	>B
(4)	[S : ((**B** (**C**$_*$ Charles) mange) son-repas)] - [(S\N)\(S\N) : lentement]	>

À l'étape (4), l'expression *Charles mange son repas* de type catégoriel S (phrase) est un faux constituant, puisqu'elle est validée en tant qu'énoncé syntaxiquement bien construit sans avoir considéré *lentement*.

Afin de résoudre ce problème, Steedman (2000) propose la décomposition du faux constituant selon le principe de la neutralité paramétrique en prenant comme indices les seuls types catégoriels. Nous pouvons définir le principe de la neutralité paramétrique comme suit : *deux types catégoriels associés entre eux par une règle catégorielle (qui ne prend en compte que les types catégoriels) permettent de déterminer le troisième type*. En d'autres termes, cela suppose la règle catégorielle combinatoire A - B —> C pour laquelle dans la mesure où nous connaissons les types catégoriels (i) A et B, nous pouvons déduire C; (ii) B et C, nous pouvons déduire A; et (iii) A et C, nous pouvons déduire B.

À l'exemple précédent, nous devrions donc effectuer une décomposition à l'étape 5. Nous avons le type catégoriel S de l'expression *Charles mange son repas* et nous avons besoin d'obtenir le type S\N pour qu'il puisse être combiné au type (S\N)\(S\N) de *lentement*. En vertu de la neutralité paramétrique, nous posons x – $S\backslash N$ —> S et nous pouvons alors déduire le troisième terme (x) qui est S/(S\N).

(1)	[N : Charles] - [(S\N)/N : mange] - [N : son-repas] - [(S\N)\(S\N) : lentement]	
(2)	[S/(S\N) : (**C**$_*$ Charles)] - [(S\N)/N : mange] - [N : son-repas] - [(S\N)\(S\N) : lentement]	>T
(3)	[S/N : (**B** (**C**$_*$ Charles) mange)] - [N : son-repas] - [(S\N)\(S\N) : lentement]	>B
(4)	[S : ((**B** (**C**$_*$ Charles) mange) son-repas)] - [(S\N)\(S\N) : lentement]	>
(5)	[S : ((**C**$_*$ Charles) (mange son-repas))] - [(S\N)\(S\N) : lentement]	
(6)	[S/(S\N) : (**C**$_*$ Charles)] - [S\N : (mange son-repas] - [(S\N)\(S\N) : lentement]	
(7)	[S/(S\N) : (**B** (**C**$_*$ Charles) mange)] - [S\N : (lentement (mange son-repas))]	<
(8)	[S : ((**C**$_*$ Charles) (lentement (mange son-repas)))]	>

Toutefois, la décomposition est confrontée à des limites importantes : (i) l'identification du type catégoriel nécessaire n'est associé à aucune stratégie clairement établie ; (ii) la méthode est inopérante en présence de certaines formes elliptiques de la coordination. Par exemple, si au lieu du cas précédent nous avons plutôt *Charles mange son repas lentement et son dessert rapidement*, l'analyse nous donne encore à l'étape 7 le type S, avant de rencontrer la conjonction de coordination *et*. L'analyse du second membre de la coordination se fera ainsi :

(8)	... - [N : son-dessert] - [(S\N)\(S\N) : rapidement]	
(9)	... - [(S\N)\((S\N)/N) : (**C**$_*$ son- dessert)] - [(S\N)\(S\N) : rapidement]	<T
(10)	... - [(S\N)\((S\N)/N) : (**B** rapidement (**C**$_*$ son- dessert))]	<B

Suite à l'étape 10, nous disposons de deux types catégoriels, soit S le type obtenu et (S\N)\((S\N)/N) le type nécessaire. Selon le principe énoncé plus haut, la décomposition de l'expression *Charles mange son repas lentement* pour extraire le premier membre de la coordination donne le type catégoriel S/((S\N)\((S\N)/N)). Or, ce type ne correspond à aucune interprétation sémantique.

Par conséquent, nous devrons nous tourner vers une autre stratégie. Nous voudrons plutôt exploiter le fait que les combinateurs de la logique combinatoire dans les expressions combinatoires véhiculent un contenu sémantique intrinsèque qui joue le rôle d'une mémoire tout le long de l'analyse, permettant ainsi de systématiser une réorganisation structurelle des constituants. Cette *réorganisation structurelle* est essentiellement différente de la décomposition que nous avons discutée précédemment, car elle utilise comme indices non seulement les types catégoriels, mais également l'interprétation sémantique fonctionnelle. Considérons à nouveau l'analyse de la même expression afin de présenter la restructuration :

(1)	[N : Charles] - [(S\N)/N : mange] - [N : son-repas] - [(S\N)\(S\N) : lentement] - [(X\X)/X : et] - [N : son-dessert] - [(S\N)\(S\N) : rapidement]	
	...	
(10)	[S : ((**C**. Charles) (lentement (mange son-repas)))] - [(X\X)/X : et] - [(S\N)\((S\N)/N) : (**B** rapidement (**C**. son- dessert))]	<B
(11)	[S/(S\N) : (**C**. Charles)] - [S\N : (lentement (mange son-repas))] - [(X\X)/X : et] - [(S\N)\((S\N)/N) : (**B** rapidement (**C**. son-dessert))]	>dec
(12)	**|S/(S\N) : (C. Charles)| - |S\N : ((B lentement (C. son-repas)) mange)| - |(X\X)/X : et| - |(S\N)\((S\N)/N) : (B rapidement (C. son-dessert))|**	
(13)	[S/(S\N) : (**C**. Charles)] - [(S\N)/N : mange] - [(S\N)/((S\N)/N) : (**B** lentement (**C**. son-repas))] - [(X\X)/X : et] - [(S\N)\((S\N)/N) : (**B** rapidement (**C**. son-dessert))]	<dec
(14)	[S/(S\N) : (**C**. Charles)] - [(S\N)/N : mange] - [(S\N)/((S\N)/N) : (**B** lentement (**C**. son-repas))] - [((S\N)\((S\N)/N))\ ((S\N)\((S\N)/N)) : (et (**B** rapidement (**C**. son-dessert)))]	>
(15)	[S/(S\N) : (**C**. Charles)] - [(S\N)/N : mange] - [(S\N)\((S\N)/N) : ((et (**B** lentement (**C**. son-repas))) (**B** rapidement (**C**. son-dessert)))]	<
(16)	[S/(S\N) : (**C**. Charles)] - [S\N : (((et (**B** lentement (**C**. son-repas))) (**B** rapidement (**C**. son-dessert))) mange)]	<
(17)	[S : ((**C**. Charles) (((et (**B** lentement (**C**. son-repas))) (**B** rapidement (**C**. son-dessert))) mange))]	>
(18)	((**C**. Charles) (((et (**B** lentement (**C**. son-repas))) (**B** rapidement (**C**. son-dessert))) mange))	

(19)	(((((et (**B** lentement (**C*** son-repas))) (**B** rapidement (**C*** son-dessert))) mange) Charles)	(**C***)
...		
(25)	((∧ (lentement (mange son-repas)) (rapidement (mange son-dessert))) Charles)	(**C***)

Les étapes 1 à 17 vérifient si la phrase est syntaxiquement bien construite, tandis que les étapes 18 à 25 font parties du niveau prédicatif et ont pour objectif de construire l'interprétation sémantique fonctionnelle par l'utilisation des règles de β-réduction des combinateurs[2]. À l'étape 10, nous disposons du faux constituant *((**C***Charles) (lentement (mange son-repas))* et du second membre de la coordination *(**B** lentement (**C*** son-dessert))*. Deux étapes sont nécessaires à l'extraction du premier membre de la coordination du faux constituant.

La première étape demande à tester si le type de l'opérande (*rapidement* (*mange son-repas*)) peut être combiné au type de l'unité linguistique résiduelle (**B** *lentement* (**C*** *son-dessert*)). Si c'est le cas alors nous procédons à la décomposition, sinon nous réduisons un combinateur. Ce processus s'exécutera jusqu'à un aboutissement ou lorsqu'il n'y aura plus de combinateur. L'expression applicative obtenue à l'étape 11 (*rapidement* (*mange son-repas*)) contient le premier membre de la coordination.

La seconde étape consiste à déterminer une structure combinatoire équivalente à (*rapidement* (*mange son-repas*)), après quoi une décomposition lui sera appliquée pour isoler le premier membre de la coordination. S'il demeure possible de déterminer arbitrairement cette équivalence, soit *((**B** rapidement (**C*** son-repas)) mange)*, il n'existe cependant aucun procédé permettant de restructurer automatiquement le constituant.

[2] Le symbole ∧ à l'étape 25 est un connecteur logique entre *(lentement (mange son-repas))* et *(rapidement (mange son-dessert))*.

L'objectif principal de notre travail consiste donc à concevoir une méthode de réorganisation structurelle automatique des expressions combinatoires. Un second objectif est d'en réaliser l'implémentation.

Le présent mémoire sera divisé en six chapitres, chacun d'entre eux portant sur une portion de notre travail.

Ainsi, en guise d'introduction, le premier chapitre fut consacré à définir brièvement la problématique et la solution envisagée.

Au chapitre 2, nous présenterons en détails la logique combinatoire fondamentalement décrite par Curry (1958). Chacun des combinateurs y sera défini. Nous verrons aussi les notions de combinateurs complexes, avec puissance et à distance. Enfin, nous aborderons un terme très important pour la suite des choses, c'est-à-dire les formes normales et le théorème de Church-Rosser.

Le chapitre 3 contiendra pour sa part un état de l'art sur les grammaires catégorielles. Nous partirons des fondements des Grammaires Catégorielles, en passant par la présentation du calcul de Lambdek, jusqu'à la Grammaire Catégorielle Combinatoire Applicative, qui est l'approche que nous privilégierons et utiliserons par la suite. La dernière section du chapitre sera consacrée à la présentation de la problématique des faux constituants sur laquelle porte le travail.

Le quatrième chapitre constitue le coeur du mémoire. Nous y proposerons une solution pour restructurer les constituants de façon à ce que l'analyse puisse se poursuivre. Bien entendu, les détails de notre démarche ainsi qu'un algorithme seront donnés. Quelques exemples illustrés seront exposés et permettront de bien en saisir le fonctionnement. Enfin, le chapitre sera clos sur le sujet de la complexité algorithmique de l'approche et sur une discussion.

Le chapitre 5 aura pour objectif de présenter l'implémentation d'un prototype fondé sur l'algorithme que nous aurons présenté au chapitre précédent.

Finalement, le chapitre 6 conclura notre travail. Nous mettrons en relief les points importants à en dégager. Nous en profiterons également pour présenter des avenues de recherche futures en lien avec notre approche.

CHAPITRE 2

LOGIQUE COMBINATOIRE

La logique combinatoire tire ses origines des travaux de Schonfinkel qui en 1924 définit la notion de combinateur, puis de ceux de Curry et Feys, un peu plus tard, en 1958. Cette notion fut introduite dans le but d'apporter une solution logique à certains paradoxes, comme celui de Russell[3], mais aussi avec l'objectif de supprimer les besoins en variables en mathématiques.

Aussi, la logique combinatoire tisse des liens étroits avec le lambda-calcul de Church (1941). Ces modèles constituent d'ailleurs les fondations de l'analyse des propriétés sémantiques langages de programmation de haut-niveau.

2.1 Les combinateurs

Nous pouvons définir les combinateurs comme étant des opérateurs abstraits qui, à partir d'autres opérateurs, contribueront à construire des opérateurs d'une plus grande complexité. Les combinateurs agissent comme des fonctions sur des arguments, dans une structure opérateur – opérandes. Ces actions sont représentées par des règles dites de β-réduction[4] qui définissent des équivalences entre une expression logique contenant un combinateur et une expression sans combinateur.

---

[3] Le paradoxe de Russell peut être définit comme suit : est-ce que l'ensemble de tous les ensembles qui ne se contiennent pas eux-mêmes comme élément se contient-il lui-même? Cela nous mène nécessairement vers une contradiction. On fait également référence à cette impasse sous le nom de « paradoxe du barbier », en hommage à la métaphore qu'a fait Russell quelques années plus tard pour l'expliquer : si un barbier rase exactement tous les hommes du village qui ne se rasent pas eux-mêmes, doit-il se raser lui-même?

[4] Au sens de Curry.

Desclés (1990) affirme que les combinateurs ne se limitent pas à jouer un rôle purement syntaxique, comme le soutenait Shaumyan, mais qu'ils vont au-delà de cela en introduisant également une sémantique intrinsèque lors des opérations de β-réduction qui représentent d'une certaine façon la signification des combinateurs.

Les prochains paragraphes présenteront un à un l'ensemble des combinateurs de la logique combinatoire.

2.1.1 Le combinateur d'identité (I)

Le combinateur **I** est la fonction d'identité. Il se joint à un opérateur f pour former l'opérateur complexe **I** f. Son action est formellement définie selon la β-réduction qui suit :

I f →f

Il est représenté par la λ-expression λ x → x.

2.1.2 Le combinateur de composition (B)

Le combinateur **B** s'associe à deux opérateurs f et g afin de construire l'opérateur complexe **B** f g tel que pour un argument x on obtienne la règle de **β**-réduction suivante :

B f g x ε f (g x)

Sa représentation en λ-expression est λ x y z → x (y z).

2.1.3 Le combinateur de substitution (S)

Le combinateur de substitution[5] agit selon le principe suivant :

Considérons deux opérateurs f et g, f étant binaire et g unaire, associés par le combinateur **S** afin de former l'opérateur complexe **S** f g. Cet opérateur agit sur un opérande x en fonction de la β-réduction suivante :

S f g x → f x (g x)

Son équivalence en terme de λ-expression est λ x y z → x z (x z).

2.1.4 Le combinateur de coordination (Φ)

Le combinateur de coordination[6], appelé ainsi par son rôle dans l'étude de la coordination, associe trois opérateurs, f g et h, pour former l'opérateur complexe **Φ** f g h. L'action de ce dernier sur un opérande x est donnée par :

Φ f g h x → f (g x) (h x)

En λ-calcul, il est exprimé comme ceci : λ x y z u → x (y u) (z u).

2.1.5 Le combinateur de distribution (Ψ)

Le combinateur **Ψ** compose avec les opérateurs f et g afin de construire l'opérateur complexe **Ψ** f g. Cet opérateur agit sur les opérandes x et y en fonction de la β-réduction suivante :

Ψ f g x y →f (g x) (g y)

---

[5] Dans les travaux de Curry (1958), ce combinateur portait le nom « de combinateur de distribution ».

[6] À l'origine, ce combinateur était lui aussi appelé « combinateur de distribution ».

La représentation du combinateur **Ψ** sous la forme d'une λ-expression va comme suit : λ x y u v → x (y u) (y v).

2.1.6 Le combinateur de changement de type (C_*)

Le combinateur C_* s'associe à un opérateur x pour former l'opérateur complexe $\mathbf{C}_*$ x dont l'action sur un opérande y est d'inter-changer les statuts opérateur-opérande. Ainsi, après réduction, x deviendra l'opérande de y:

$\mathbf{C}_*$ x y → y x

La λ-expression de ce combinateur est λ x y → y x.

2.1.7 Le combinateur de permutation (C)

Le combinateur C utilise un opérateur f afin de construire l'opérateur complexe C f tel que si f agit sur les opérandes x et y, C f agira sur ces opérandes dans l'ordre inverse, soit y et x, de la façon suivante :

C f y x →f x y

La λ-expression du combinateur de permutation est λ x y z → x z y.

Ce combinateur permet la description de verbes symétriques comme *croiser* ou *rencontrer*. Prenons l'exemple de la proposition *Charles croise Mathieu*, représentée par l'expression combinatoire *croise Charles Mathieu*, et posons *croise* = ***C*** *croise*. Si nous effectuons un remplacement dans l'expression combinatoire, nous obtenons ***C*** *croise Charles Mathieu*. L'application de la réduction nous donnera par conséquent *croise Mathieu Charles*, c'est-à-dire *Charles croise Mathieu*, qui est la réflexion de *Mathieu croise Charles*.

2.1.8 Le combinateur de duplication (W)

Considérons un opérateur f et un opérande x. Le combinateur de duplication **W** s'associe à f pour construire le combinateur complexe **W** f et ainsi agir sur x de la façon suivante :

W f x →f x x

La λ-expression qui correspond à cette action est λ x y → x y y.

Comme nous pouvons le constater, le résultat de l'action est de nouveau appliqué à x. Cela est particulièrement utile dans le traitement des prédicats réfléchis, comme par exemple *se-laver* ou *se-convaincre*. Si nous prenons ce dernier et que nous l'appliquons à Charles, il construira l'expression combinatoire *se-convainc Charles*. En posant *se-convainc* = **W** *convaincre*, nous pouvons ainsi substituer *se-convaincre* par **W** *convaincre* dans l'expression, ce qui donne **W** *convainc Charles*. La réduction donne alors *convainc Charles Charles* ou, autrement dit, *Charles convainc Charles*. Cela démontre l'équivalence sémantique qui existe entre les propositions *Charles se convainc* et *Charles convainc Charles*.

2.1.9 Le combinateur d'adjonction d'un opérande fictif (K)

Le combinateur **K** construit un opérateur complexe avec un opérateur f afin d'éliminer un opérande x que nous appelons « opérande fictif ». Cette action est décrite comme suit :

K f x → f

Le combinateur K s'exprime sous la représentation d'une λ-expression par λ x y → x.

Nous nous servons de ce combinateur lors du traitement de constructions impersonnelles, c'est-à-dire des constructions impliquant des verbes impersonnels tels *falloir*, *pleuvoir*, *geler* et *neiger*. Par exemple, dans la phrase *il neige*, *neige* constitue un opérateur unaire ayant *il* pour opérande fictif. En linguistique, *neige* est considéré comme un prédicat unaire qui est le dérivé du prédicat zéro-aire *il-y-a-neige*. Maintenant, si nous posons l'expression combinatoire *neige il* et que nous substituons l'équivalence *neige* = **K** *il-y-a-neige* dans l'expression, cela donne **K** *il-y-a-neige il*. La réduction de cette nouvelle expression mènera à *il-y-a-neige* qui a par conséquent la même signification que *il neige*.

2.1.10 Les combinateurs complexes

L'association de plusieurs combinateurs a pour résultat de construire des combinateurs complexes. Nous pourrions avoir, par exemple, le combinateur complexe **B B C**$_*$ x y z. Son action est déterminée par l'application successive des combinateurs élémentaires qui le composent, du combinateur de gauche vers celui de droite. L'ordre de réduction des combinateurs est donc **B**, **C** et enfin **C**$_*$. Ainsi :

(i) **B B C**$_*$ x y z

(ii) **B** (**C**$_*$ x) x y

(iii) (**C**$_*$ x) (y z)

(iv) y z x

Il existe deux autres cas de combinateurs complexes qui seront décrits ci-dessous.

2.1.10.1 Les combinateurs avec une puissance

Il est possible d'attribuer une puissance à un combinateur. Cela aura pour effet de réitérer *n* fois l'action du combinateur en vertu de la définition qui suit :

Si χ est un combinateur alors χ^n itère *n* fois l'action de χ tel que $\chi^1 = \chi$ et $\chi^n = \mathbf{B}\ \chi\ \chi^{n-1}$.

Par exemple, si nous avons l'expression combinatoire complexe $\mathbf{B}^3$ a b c d e, l'action est donnée par la succession d'équivalence ci-contre:

(i) $\mathbf{B}^3$ a b c d e
(ii) **B B** $\mathbf{B}^2$ a b c d e
(iii) **B B** (**B B B**) a b c d e
(iv) **B** (**B B B** a) b c d e
(v) **B B B** a (b c) d e
(vi) **B** (**B** a) (b c) d e
(vii) **B** a (b c d) e
(viii) a (b c d e)

2.1.10.2 Les combinateurs à distance

Le second type particulier de combinateurs complexes est celui des combinateurs à distance. Ils tirent cette appellation du fait qu'ils agissent à distance dans l'expression selon le principe suivant :

Si χ est un combinateur alors χ_n diffère son action de *n* pas tel que $\chi_1 = \chi$ et $\chi_n = \mathbf{B}^n\ \chi$.

Par exemple, considérons l'expression combinatoire $\mathbf{C}_2$ a b c d e. L'action du combinateur complexe se traduit par sa réduction, dont voici les étapes intermédiaires :

(i) $\mathbf{C}_2$ a b c d e
(ii) $\mathbf{B}^2$ **C** a b c d e
(iii) **B B B C** a b c d e
(iv) **B** (**B C**) a b c d e
(v) **B C** (a b) c d e

(vi) **C** (a b c) d e

(vii) a b c e d

2.2 Théorèmes d'équivalences sur les combinateurs

Curry (1958) fait la démonstration que tous les combinateurs, hormis le combinateur C_*, peuvent être exprimés en utilisant uniquement les combinateurs **S** et **K**. Ainsi, nous retrouvons les équivalences suivantes[7] :

I = S K K

B = S (K S) K

W = S S (K (S K K))

C = S (B B S) (K K)

Φ = B (B S) B

Ψ = Φ (Φ (Φ B)) B (K K)

2.3 La forme normale

Une expression est considérée comme étant sous forme normale une fois qu'elle a été réduite de tous ses combinateurs. Plus formellement, si nous avons la forme normale E' et l'expression combinatoire E, et que la réduction de E mène à E', nous dirons de E' qu'elle est la forme normale de E.

[7] Les démonstrations de ces équivalences peuvent être trouvées dans (Desclés, 1990).

Par exemple, considérons l'expression combinatoire suivante : **B B C*** a b c d. Les étapes de réduction de l'expression sont :

(i) **B C*** a b (c d)

(ii) **C*** a (b (c d))

(iii) (b (c d)) a

Alors, (b (c d)) a est la forme normale de **B B C*** a b c d.

Cependant, toutes les expressions combinatoires ne possèdent pas nécessairement une forme normale comme, par exemple, l'expression **W W W** qui, lorsque nous réduisons le combinateur **W**, son argument (un autre combinatoire **W**) est dupliqué. La réduction se poursuivrait ainsi *ad vitam aeternam*, sans jamais permettre l'atteinte de la forme normale.

2.3.1 Le théorème de Church-Rosser

Ce théorème s'avère être d'une importance capitale en logique combinatoire. Il pose la question de l'unicité de la forme normale, à savoir si une expression combinatoire peut être réduite à plusieurs formes normales ou bien à une seule. Le théorème de Church-Rosser affirme que si une expression combinatoire X peut être réduite en deux expressions distinctes Y_1 et Y_2, alors il existe une expression combinatoire Z tel que Y_1 et Y_2 se réduisent en Z. Cela signifie que les expressions de la logique combinatoire possèdent la propriété de Church-Rosser en ce qui concerne les relations transitives, qui peut être illustré selon le schéma suivant :

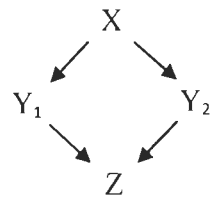


Figure 1 Propriété de Church-Rosser

De ce théorème découlent les corollaires qui suivent :

(i) Une expression combinatoire a au plus une seule forme normale;

(ii) Si X = Y alors il existe un Z tel que X → Z et Y → Z;

(iii) Si X = Y et Y est une forme normale alors X → Y;

(iv) Si X = Y alors soit X et Y ont soit la même forme normale, soit ils n'en ont aucune;

(v) Si la forme normale de X est différente de celle de Y alors X ≠ Y.

CHAPITRE 3

GRAMMAIRES CATÉGORIELLES

Dans le cadre de ce chapitre, nous présenterons un état de l'art des Grammaires catégorielles. En partant des origines philosophiques et logiques du modèle, nous décrirons ensuite les modèles d'Ajdukiweicz (1935), Bar-Hillel (1953), Lambek (1958, 1961) et Steedman (1982, 1987, 1989). Puis, nous ferons un survol d'autres modèles récents basés sur la Grammaire Catégorielle avant de s'attaquer au modèle de la Grammaire Catégorielle Combinatoire Applicative développé par Desclés et Biskri (1996, 2006). C'est dans cette dernière partie que nous exposerons la problématique des faux constituants, le sujet de nos travaux.

3.1 Les origines philosophiques et logiques

Les origines philosophiques des Grammaires Catégorielles remontent jusqu'aux travaux de recherche du philosophe Husserl (1913), où il reprend une opposition traditionnelle depuis les Grecs en distinguant les expressions catégorématiques des expressions syncatégorématiques. Husserl argumente que tout symbole grammatical n'a pas une signification indépendante par rapport aux éléments sur lesquels il porte. Ainsi, il définit les expressions catégorématiques comme étant les expressions pourvues d'une signification comme les syntagmes nominaux ou les énoncés, tandis que les expressions syncatégorématiques sont celles dont la signification n'est complète que lorsqu'elles sont considérées conjointement avec d'autres parties du discours dont la signification est également partielle.

Par la suite, le logicien polonais Lesniewski (1922) a élaboré le concept des catégories sémantiques, en s'inspirant de la tradition des catégories aristotéliciennes, des parties du discours de la grammaire traditionnelle et des catégories de signification de Husserl. Il a considéré comme types d'expressions les noms, des expressions linguistiques servant à

désigner des entités objectales ou des classes de ces entités, ainsi que les propositions, des expressions linguistiques construites par une énonciation visant à représenter un « état de chose » comme un objet intentionnel. Il considère toutes les autres expressions du langage comme étant des syncatégorèmes. Nous pouvons d'hors et déjà voir les expressions syncatégorématiques comme des opérateurs servant à former des expressions catégorématiques.

Cette idée de classification constituera les fondements des modèles qui seront développés un peu plus tard par Ajdukiewicz, Bar-Hillel et Lambek et qui formeront ce que nous désignerons sous l'appellation des Grammaires Catégorielles. Les trois prochaines sections décriront chacun de ces modèles.

3.2 Le modèle d'Ajdukiewicz

Reprenant le système de classification de Lesniewski, le philosophe Kazimierz Ajdukiewicz (1935) propose de diviser en deux classes l'ensemble des catégories : les catégories de base et les catégories foncteurs. Les catégories de base incluent les catégories syntagmes nominaux, notées « N », et les catégories phrases, notées « S »[8], tandis que les catégories foncteurs sont récursivement construites à partir des catégories de base et d'un symbole d'application, permettant ainsi de générer une infinité de catégories.

L'ensemble des catégories pouvant appartenir à ce modèle sont contraintes à ces règles :

(i) Les catégories de base du modèle constituent des catégories de ce modèle;

(ii) Considérant que X et Y sont des catégories du modèle alors $\frac{X}{Y}$ est aussi une catégorie de ce modèle, nommée « catégorie foncteur », pour laquelle la

[8] « N » provient de l'anglais « noun » qui signifie « nom » et « S » de « sentence » qui signifie « phrase ».

barre de fraction symbolise l'application, le dénominateur le type de l'argument et le numérateur le type du résultat de l'application du foncteur de type $\frac{X}{Y}$ à l'argument Y.

La stratégie d'Ajdukiewicz sera d'attribuer des types catégoriels aux mots et aux expressions en fonction des rapports qu'ils entretiennent entre eux dans la phrase. Les deux règles de réduction unidirectionnelles suivantes permettront ensuite de vérifier si la suite de mots a une bonne connexion syntaxique :

(i) $\frac{X}{Y}$ Y → X

(ii) Y $\frac{X}{Y}$ → X

Par exemple, considérons la phrase *Charles saute* :

Charles	saute
N	$\frac{S}{N}$
S	

L'utilisation de la règle de réduction (ii) donne le type S, puisque :

N $\frac{S}{N}$ → S

Selon Ajdukiewicz, si par l'application de ces règles de réduction nous parvenons à obtenir une expression ayant l'une ou l'autre des catégories de base, cela signifie que l'expression est syntaxiquement valide.

Par ailleurs, il établit une distinction entre l'opérateur principal et les autres opérateurs d'une expression. Nous nous permettrons ici de citer Ajdukiewicz lui-même :

> « In every signifiant composite the relations of functors to their arguments have to be such that the entire expression may be divided into constituents, of which one is a functor (possibly itself a composite expression) and the others are its arguments. This functor we call the main functor of the expression. »

En d'autres mots, les relations des foncteurs envers leurs arguments sont telles que l'expression complète peut être divisées en constituants, dont un est un foncteur et les autres ses arguments. Ce foncteur est celui qu'on appelle le foncteur principal et il peut être lui-même une expression complexe.

Par conséquent, la validation syntaxique d'une expression ne dépendra pas uniquement du fait que la réduction des types mène à un des types de base, mais également de la possibilité qu'elle puisse être divisée en un foncteur principal suivi de ses arguments.

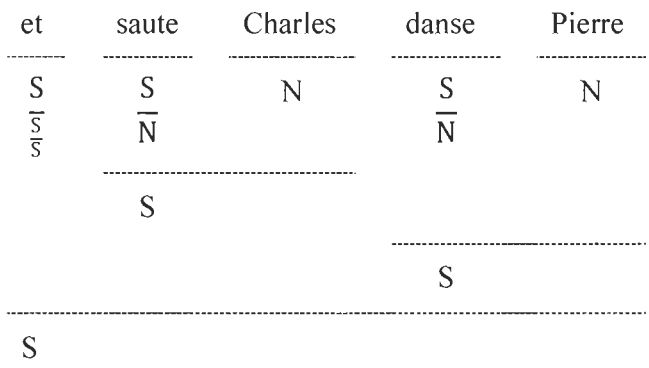
Ainsi, considérons maintenant la phrase *Charles saute et Pierre danse* :

Charles	saute	et	Pierre	danse
N	$\frac{S}{N}$	$\frac{S}{\frac{S}{S}}$	N	$\frac{S}{N}$

Dans cette phrase, le foncteur *saute* et son argument *Charles* et le foncteur *danse* et son argument *Pierre* sont les arguments du foncteur principal *et*.

Afin de pouvoir faire la vérification des conditions visant à valider la bonne forme syntaxique de l'expression, nous devons modifier l'ordre des arguments de manière à avoir un foncteur suivi à sa droite par ses arguments. Cette nouvelle configuration nous

permettra d'utiliser les règles de réduction que nous avons présentées auparavant. Ainsi :



Puisque le type S est finalement obtenu et que l'expression *Charles saute et Pierre danse* peut être divisée en des foncteurs et leurs arguments, la syntaxe est considérée valide.

3.3 Le modèle de Bar-Hillel

En 1952, le linguiste et mathématicien Yehoshua Bar-Hillel élabore un modèle catégoriel qui partage avec celui d'Ajdukiewicz plusieurs caractéristiques, dont le fait de n'utiliser que des règles de réduction. Toutefois, le modèle de Bar-Hillel se distingue par la bidirectionnalité de ces règles. Ainsi, les réductions peuvent s'effectuer dans les deux sens, selon l'ordre dans lequel l'opérateur attend ses arguments.

Le modèle considère les catégories suivantes :

(i) Les catégories de base, soient S et N;

(ii) Les catégories complexes X/Y et X\Y, où X et Y représentent des catégories de base ou non. X/Y est la catégorie d'un opérateur dont l'argument est un opérande de type Y positionné à droite. Inversement, X\Y est la catégorie

d'un opérateur dont l'argument est un opérande de type X positionné à gauche.

Il spécifie les deux règles de réduction (respectivement de droite et de gauche) suivantes :

(i) X/Y Y → X

(ii) Y Y\X → X

Considérons la phrase *Charles taquine Pierre* :

Charles	taquine	Pierre	
N	(N\S)/N	N	
	N\S		(règle de droite)
S			(règle de gauche)

L'application successive des règles de réduction de droite et de gauche nous donne le type S, ce qui a pour conséquence de valider la structure syntaxique de la phrase.

3.4 Le Calcul de Lambek

Le mathématicien Joachim Lambek publia des articles en 1958 et en 1961 dans lesquels il présenta un certain nombre de règles de calcul sur les types nommé « Calcul de Lambek ».

Ce système inclut les éléments suivants :

(i) Les types primitifs S et N;

(ii) Les opérations constructeurs de types / et \ qui se lisent respectivement « sur » et « sous »;

(iii) Les types X/Y et X\Y, représentant des types d'opérateurs ayant un opérande de type Y situés respectivement à droite ou à gauche donnant pour résultat un argument de type X[9];

(iv) Le symbole de concaténation « - ». La concaténation de deux expressions E_1 et E_2 s'écrit E_1-E_2. Si E_1 est de type X et E_2 de type Y, alors le type de l'expression E_1-E_2 est X-Y;

(v) La relation de réduction « —> » pour laquelle X —> Y signifie que le type X est réduit au type Y. La relation X <—> Y indique une bidirectionnalité dans la relation, c'est-à-dire X —> Y et Y —> X.

Le calcul de Lambek est un système de calcul sur les types caractérisé par les trois axiomes suivants :

(1) X —> X (Réflexivité)

(2) (X-Y)-Z —> X-(Y-Z) (Associativité de la concaténation)

(3) X-(Y-Z) —> (X-Y)-Z (Associativité de la concaténation)

De ces axiomes, nous pouvons inférer les cinq règles suivantes :

(4) Si X —> Y et Y —> Z alors X —> Z (Transitivité)

[9] Cette notation est différente de celles données originairement par Lambek et Bar-Hillel. Pour une transition plus aisée vers le modèle de la Grammaire Catégorielle Combinatoire qui suivra, nous préférons immédiatement passer à la notation de Steedman, dans laquelle l'opérateur se trouve toujours à gauche et le résultat à droite.

(5) Si X-Y —> Z alors X —> Z/Y (Curryfication à droite)

(6) Si X-Y —> Z alors Y —> Z\X (Curryfication à gauche)

(7) Si X —> Z/Y alors X-Y —> Z (Décurryfication à droite)

(8) Si Y —> Z\X alors X-Y —> Z (Décurryfication à gauche)

Ces axiomes et ces règles permettent la déduction des théorèmes qui suivent[10] :

(T1) X —> (X-Y)/Y

(T2) (Z/Y)-Y —> Z (Application)

(T3) Y —> Z\(Z/Y) (Changement de type)

(T4) (Z/Y)-(Y/X) —> Z/X (Composition fonctionnelle de types)

(T5) Z/Y —> (Z/X)/(Y/X) (Division des types)

(T6) (Y\X)/Z <—> (Y/Z)\X (Permutation)

(T7) (X/Y)/Z <—> X/(Z-Y)

(T8) Si X —> X' et Y —> Y' alors X-Y —> X'-Y'

(T9) Si X —> X' et Y —> Y' alors X/Y' —> X'/Y

[10] Nous invitons le lecteur à consulter l'article de Lambek (1961) pour la démonstration des théorèmes.

Ces neufs théorèmes expriment de nouvelles réductions sur les types qui sont spécifiques au Calcul de Lambek. Les théorèmes T2 à T4 introduisent les concepts très importants de l'application, du changement de type et de la composition fonctionnelle.

3.5 La Grammaire Catégorielle Combinatoire

La Grammaire Catégorielle Combinatoire de Mark Steedman (1987, 1989) constitue une généralisation des Grammaires Catégorielles de Ajdukiewicz, de Bar-Hillel et de Lambek. Elle emprunte aux Grammaires Catégorielles le concept de la catégorisation des unités du langage en opérandes ou en opérateurs agissant sur des opérandes et à la logique combinatoire l'utilisation des combinateurs afin de se doter d'opérations sur des types autres que l'application fonctionnelle.

Ces opérations incluent le changement de type (opération unaire) et la composition fonctionnelle (opération binaire), basés sur des théorèmes du Calcul de Lambek, puis la substitution fonctionnelle (opération binaire) proposée dans les travaux de Szabolcsi (1987).

Grâces à elles, nous serons en mesure de pouvoir considérer des constituants linguistiques dans la phrase autres que les verbes et les syntagmes verbaux. Comme nous le verrons, elles permettront également une analyse incrémentale des énoncés. La nature de cette approche incrémentale est issue des travaux de Haddock (1987), dans lesquels ce dernier affirme que la compréhension d'une phrase est incrémentale puisque chaque terme successif contribue à l'accumulation graduelle du sens.

Le système proposé par Steedman autorise sans s'y limiter les catégories de base suivantes :

(i) La phrase (S);

(ii) Le syntagme nominal (NP);

(iii) Le syntagme verbal (VP);

(iv) Le groupe prépositionnel (PP).

Les catégories complexes sont construites récursivement à l'aide de catégories de base et d'opérateurs construction droite (/) et gauche (\).

Une catégorie est donnée aux éléments du langage, comme les verbes, les pronoms ou les adverbes, et leurs confère un statut d'opérateur (foncteur). À titre d'exemple, le type syntaxique du verbe *respecter* est *(S\NP)/NP*, *NP* étant le type des arguments du foncteur *respecter*. Effectivement, le verbe *respecter* agit sur un syntagme nominal en position objet afin de créer un syntagme verbal qui lui-même s'appliquera à un syntagme nominal en position sujet pour former la phrase (Steedman, 1989).

Dans ce modèle, les catégories sont considérées comme étant des objets non seulement syntaxiques, mais aussi sémantiques. C'est d'ailleurs pourquoi elles sont représentées par une structure de données informatique unique dans le cadre d'une implémentation d'un analyseur basé sur l'unification (Pareschi et Steedman, 1987).

Considérant cela, la catégorie du verbe *respecter* sera notée :

(S:respecter' np2 np1\NP:np1)/NP:np2

Dans cette notation, les lettres majuscules représentent les types syntaxiques, les lettres minuscules des variables sémantiques et les symboles « ' » les constantes sémantiques.

Considérant qu'elles sont à la fois syntaxiques et sémantiques, voici les règles combinatoires proposées par Steedman :

Règles d'application :

[X/Y : λ x (f x)] - [Y : a]	[Y : a] - [X\Y : λ x (f x)]
----------> ;	----------<
[X : (f a)]	[X : (f a)]

Règles de changement de type :

[X: x]	[X : x]
---------->**T** ;	----------<**T**
[Y/(Y\X) : λ f (f x)]	[Y\(Y/X) : λ f (f x)]
[X: x]	[X : x]
---------->**Tx** ;	----------<**Tx**
[Y/(Y/X) : λ f (f x)]	[Y\(Y\X) : λ f (f x)]

Règles de composition fonctionnelle :

[X/Y : λ y (f y)] - [Y/Z : λ x (g x)]	[Y\Z : λ x (g x)] - [X\Y : λ y (f y)]
---------->**B** ;	----------<**B**
[X/Z : λ x (f (g x))]	[X\Z : λ x (f (g x))]
[X/Y : λ y (f y)] - [Y\Z : λ x (g x)]	[Y/Z : λ x (g x)] - [X\Y : λ y (f y)]
---------->**Bx** ;	----------<**Bx**
[X\Z : λ x (f (g x))]	[X/Z : λ x (f (g x))]

Règles de substitution fonctionnelle :

[(X/Y)/Z : λ x λ y (f x y)] - [Y/Z : λ x (g x)] [Y\Z : λ x (g x)] - [(X\Y)\Z : λ x λ y (f x y)]

----------->**S** ;-----------<**S**

[X/Z : λ x (f x (g x))] [X\Z : λ x (f x (g x))]

[(X/Y)\Z : λ x λ y (f x y)] - [Y\Z : λ x (g x)] [Y/Z : λ x (g x)] - [(X\Y)/Z : λ x λ y (f x y)]

----------->**Sx** ; ----------<**Sx**

[X\Z : λ x (f x (g x))] [X/Z : λ x (f x (g x))]

Pour chacune des règles présentées, les types syntaxiques sont associés à des interprétations sémantiques qui nous permettent d'exprimer l'aspect fonctionnel des énoncés. La construction de l'interprétation sémantique utilise le lambda-calcul et l'unification, comme nous le verrons en reprenons l'exemple de la phrase *Charles taquine Pierre* :

Charles	taquine	Pierre	
N : Charles'	(S : taquine' np2 np1\NP : np1)/NP : np2	NP : Pierre'	
S : pred Charles'/(S : pred Charles'\NP : Charles')			(>**T**)
S : taquine' np2 Charles'/NP : np2			(>**B**)
S : taquine' Pierre' Charles			(>)

À la première étape, nous utilisons immédiatement la règle de changement de type >**T** sur l'argument *Charles* de type N, dans le but d'obtenir le type S/(S\NP). Ce type permet alors une composition fonctionnelle par la règle >**B** avec le type de l'argument *taquine*, (S\NP)/NP. Le type qui en résulte, S/NP, est finalement unifié avec le type de l'argument *Pierre* pour obtenir le type S qui confirme la validation syntaxique de la phrase.

3.5.1 Analyse incrémentale

Le problème le plus important issu de l'analyse syntaxique avec les Grammaires Catégorielle est celui de la pseudo-ambiguïté. Ce problème se pose lorsque plusieurs analyses pour un même énoncé peuvent être considérées alors qu'elles réfèrent à une interprétation sémantique unique.

Par exemple, la phrase utilisée précédemment, *Charles taquine Pierre*, pourrait être analysé autrement :

Charles	taquine	Pierre	
NP	(S\NP)/NP	NP	
	S\NP		(>)
S			(<)

Dans cette deuxième analyse, nous appliquons d'abord la catégorie du verbe à celle de son objet pour ensuite appliquer la catégorie résultante avec celle du sujet.

Voici une troisième analyse possible :

Charles	taquine	Pierre	
NP	(S\NP)/NP	NP	
S/(S\NP)			(>**T**)
	S\NP		(>)
S			(>)

Cette solution consiste à d'abord appliquer une opération de changement de type, mais plutôt que d'appliquer une composition fonctionnelle, nous appliquons immédiatement

(S\NP)/NP avec NP pour obtenir la catégorie S\NP pour le prédicat complexe. Dès lors, une autre application avant suffit pour obtenir le type S.

Afin de résoudre ce problème, Haddock (1987), Pereschi (1987) et Steedman (1987) soumettent l'idée d'une analyse syntaxique incrémentale de gauche vers la droite afin de ne privilégier qu'une seule et unique analyse pour laquelle les règles applicatives et combinatoires sont utilisées aussitôt que possible.

Cependant, cette stratégie introduit un autre problème qui est celui du non déterminisme, dû à la présence de modifieurs arrière dans le langage. Ces modifieurs arrières sont des opérateurs qui agissent sur une partie ou la totalité d'une structure déjà construite.

Dans certains cas, de tels modifieurs peuvent tout de même permettre une analyse incrémentale de gauche à droite. Par exemple, dans l'énoncé *Charles taquinera Pierre demain* pour laquelle *demain* est le modifieur arrière de *Charles taquine Pierre* (par conséquent, il se voit attribuer le type S\S), nous avons l'analyse suivante :

	Charles	taquinera	Pierre	demain	
(1)	NP	(S\NP)/NP	NP	S\S	
(2)	S/(S\NP)				(>**T**)
(3)	S/NP				(>**B**)
(4)	S				(>)
(5)	S				(<)

À la dernière étape, une application arrière permet d'obtenir le type S pour l'expression.

Toutefois, une stratégie strictement incrémentale ne parvient pas à résoudre une situation comme la phrase *Charles taquine Pierre gentiment*, où le type de l'adverbe *gentiment*

est (S\NP)\(S\NP), puisqu'il est un modifieur de l'opérande *taquine Pierre* de type S\NP. Voyons plutôt :

	Charles	taquine	Pierre	gentiment	
(1)	NP	(S\NP)/NP	NP	(S\NP)\(S\NP)	
(2)	S/(S\NP)				(>**T**)
(3)	S/NP				(>**B**)
(4)	S				(>)

À l'étape 4, l'analyse produit un *faux constituant*[11] : *(taquine' Pierre' Charles')*. En effet, ce faux constituant est de type S et n'est pas combinable avec *gentiment* qui lui est de type (S\NP)\(S\NP). Or, c'est l'énoncé en entier qui doit être de type S afin d'être jugé comme étant syntaxiquement valide et il faut donc nécessairement pouvoir considérer l'adverbe *gentiment*.

3.5.2 Neutralité paramétrique et décomposition

L'ensemble des règles combinatoires définies par la Grammaire Catégorielle Combinatoire possèdent une propriété nommée « *neutralité paramétrique* » (Pareschi, 1987; Steedman, 1987).

Son principe va comme suit : *deux catégories associées entre-elles par une règle combinatoire déterminent la troisième catégorie*. En d'autres mots, lorsque nous utilisons des règles combinatoires de forme A – B —> C, nous avons en entrée deux catégories fonctionnelles (A et B) qui sont composées pour former une troisième catégorie en sortie (C). Selon ce principe, il suffit de connaître deux catégories parmi

[11] *Spurious constituent*.

ces trois afin de pouvoir déduire la dernière. Ainsi, considérant la règle combinatoire A – B —> C :

(i) Si nous connaissons A et B alors nous pouvons déduire C;

(ii) Si nous connaissons A et C alors nous pouvons déduire B;

(iii) Si nous connaissons B et C alors nous pouvons déduire A.

Steedman se servira de ce principe afin de résoudre des cas de faux constituants comme celui que nous avons présenté précédemment.

Nous avions alors construit pour l'énoncé *Charles taquine Pierre gentiment* le faux constituant *(taquine' Pierre' Charles')* de type S et l'argument *gentiment* de type (S\NP)\(S\NP). L'analyseur procédera à une décomposition du faux constituant par rapport à la règle d'application avant, c'est-à-dire X/Y – Y —> X, pour laquelle :

(i) X = S : *taquine' Pierre' Charles'*

(ii) X/Y = *(S:pred Charles')/(S:pred Charles'\NP:Charles')*

Par unification, nous pouvons ainsi déduire :

(iii) Y = S : *taquine' Pierre' Charles'\NP : Charles'*

Cela engendre l'analyse complète suivante :

	Charles	taquine	Pierre	gentiment
(1)	NP	(S\NP)/NP	NP	(S\NP)\(S\NP)

(2)	S/(S\NP)		(>**T**)
(3)	S/NP		(>**B**)
(4)	S		(>)
(5)	S/(S\NP)	S\NP	(**déc.**)
(6)		S\NP	(<)
(7)	S		(>)

Comme nous pouvons le constater, l'opération de décomposition permet au type S : *taquine' Pierre' Charles'\NP : Charles'* de pouvoir se combiner au type du modifieur *gentiment* et S : *taquine' Pierre' Charles'\NP : Charles'* fait ensuite de même avec le résultat pour enfin obtenir le type S pour l'énoncé dans sa totalité.

3.6 Autres modèles récents

Avant de poursuivre, nous désirons à travers cette section faire mention de certains autres modèles catégoriels sur lesquels nous ne nous attarderons pas, mais dont nous tenons tout de même à signaler l'existence.

Parmi eux, soulignons tout d'abord la Grammaire Catégorielle Combinatoires Multimodale de Baldridge et Kruijff (2003). Ce modèle se veut être une hybridation de la Grammaire Catégorielle Combinatoire de Steedman et la Logique des Types Catégoriels[12] (Morrill, 1994; Moortgat, 1997). Il permet entre autres des restrictions sur l'opérabilité des règles catégorielles pour éliminer des cas d'ambiguïté.

Par ailleurs, le modèle des Grammaires Catégorielles Abstraites (De Groote, Podogalla 2004) décrit le niveau syntaxique et la relation entre le niveau syntaxique et le niveau

[12] *Categorial Type Logics*

sémantique. Il s'appuie sur un langage abstrait (les structures de dérivation) et un langage objet (les structures dérivées).

Enfin, le modèle de la Grammaire Catégorielle Combinatoire d'Héritage des Types[13] (Beavers, 2004) exploite les performances des Grammaires Catégorielles tout en ayant une base grammaticale syntagmatique issue de la Grammaire de Structures de Phrases Conduites par Tête[14].

3.7 La Grammaire Catégorielle Combinatoire Applicative

Le modèle de la Grammaire Catégorielle Combinatoire Applicative développé par Biskri et Desclés (1996) fait parti du modèle général de la Grammaire Applicative et Cognitive (Desclès, 1990), une extension de la Grammaire Applicative Universelle (Shaumyan, 1998). Selon ces modèles, une analyse du langage doit postuler trois niveaux de représentation, dont l'objectif principal est d'étudier les propriétés formelles des systèmes linguistiques et d'en analyser les structures logiques. Ces trois niveaux sont les suivants :

(i) *Le niveau des structures morphosyntaxiques*, où sont exprimées les caractéristiques particulières de la langue, comme par exemple l'ordre des mots dans les énoncés et les cas morphologiques. Il s'agit de la représentation observable du langage;

(ii) *Le niveau des structures prédicatives*, où sont exprimés les invariants grammaticaux ainsi que les structures prédicatives sous-jacentes aux énoncés phénotypiques. Ce niveau permet d'exprimer l'interprétation sémantique

[13] *Type-Inheritance Combinatory Categorial Grammar*.

[14] *Head-Driven Structures Grammar*.

fonctionnelle des énoncés dans laquelle chaque unité linguistique est un opérateur suivi de ses opérandes ;

(iii) *Le niveau cognitif*, où la signification des prédicats linguistiques est donnée.

Le modèle de la Grammaire Catégorielle Combinatoire Applicative élargit celui de la Grammaire Catégorielle Combinatoire par une association canonique entre les règles catégorielles combinatoires de Steedman et les combinateurs de la logique combinatoire de Curry. Ainsi, l'utilisation des règles catégorielles combinatoires introduisent les combinateurs dans la chaîne syntagmatique, ce qui permet de passer d'une structure concaténée vers une structure applicative.

À l'instar d'autres modèles présentées auparavant, les règles ont pour objectif de vérifier la bonne connexion syntaxique des phrases par l'obtention du type S. De plus, cette vérification construira une expression applicative typée avec combinateurs qui, une fois réduits, permettent d'obtenir l'interprétation sémantique fonctionnelle de l'énoncé.

Les règles de la Grammaire Catégorielle Combinatoire Applicative sont les suivantes :

Règles d'application et de décomposition:

$[X/Y : u_1]$ - $[Y : u_2]$		$[Y : u_1]$ - $[X\backslash Y : u_2]$
---------->	;	<----------
$[X : (u_1\ u_2)]$		$[X : (u_2\ u_1)]$

Règles de changement de type :

$[X: u]$		$[X : u]$
---------->**T**	;	<----------**T**
$[Y/(Y\backslash X) : (\mathbf{C}_*\ u)]$		$[Y\backslash(Y/X) : (\mathbf{C}_*\ u)]$

[X: u] [X : u]

----------->**Tx** ; -----------<**Tx**

[Y/(Y/X) : (**C*** u)] [Y\(Y\X) : (**C*** u)]

Règles de composition fonctionnelle :

[X/Y : u_1] - [Y/Z : u_2] [Y\Z : u_1] - [X\Y : u_2]

----------->**B** ; -----------<**B**

[X/Z : (**B** u_1 u_2)] [X\Z : (**B** u_2 u_1)]

[X/Y : u_1] - [Y\Z : u_2] [Y/Z : u_1] - [X\Y : u_2]

----------->**Bx** ; -----------<**Bx**

[X\Z : (**B** u_1 u_2)] [X/Z : (**B** u_2 u_1)]

Règles de substitution fonctionnelle :

[(X/Y)/Z : u_1] - [Y/Z : u_2] [Y\Z : u_1] - [(X\Y)\Z : u_2]

----------->**S** ; -----------<**S**

[X/Z : (**S** u_1 u_2)] [X\Z : (**S** u_2 u_1)]

[(X/Y)\Z : u_1] - [Y\Z : u_2] [Y/Z : u_1] - [(X\Y)/Z : u_2]

----------->**Sx** ; -----------<**Sx**

[X\Z : (**S** u_1 u_2)] [X/Z : (**S** u_2 u_1)]

La prémisse de chaque règle est une concaténation d'unités linguistiques typées. La conséquence est une expression applicative typée avec possiblement l'introduction d'un combinateur : le changement de type introduit le combinateur **C*** et la composition de deux unités concaténées introduit le combinateur **B** ou **S**.

L'analyse au moyen des règles utilise une approche quasi-incrémentale de gauche à droite. Considérons l'exemple *Charles taquine Pierre* :

(1)	[N : Charles] - [(S\N)/N : taquine] - [N : Pierre]	
(2)	[S/(S\N) : (**C**$_*$ Charles)] - [(S\N)/N : taquine] - [N : Pierre]	>**T**
(3)	[S/N : (**B** (**C**$_*$ Charles) taquine)] - [N : Pierre]	>**B**
(4)	[S : ((**B** (**C**$_*$ Charles) taquine) Pierre)]	>
(5)	((**B** (**C**$_*$ Charles) taquine) Pierre)	
(6)	((**C**$_*$ Charles) (taquine Pierre))	(**B**)
(7)	((taquine Pierre) Charles)	(**C**$_*$)

À l'étape 2, un changement de type >**T** a été appliqué sur l'opérande *Charles* afin de construire l'opérateur *(**C**$_*$ Charles)* de type S/(S\N). Ce dernier peut alors se composer avec l'opérateur *taquine* de type (S\N)/N en utilisant la règle de composition fonctionnelle >**B**, ce qui forme l'opérateur complexe *(**B** (**C**$_*$ Charles) taquine)* de type S/N qui s'applique à l'opérande *Pierre* afin d'obtenir l'expression applicative *((**B** (**C**$_*$ Charles) taquine) Pierre)* de type S (étape 4).

Les étapes 5 à 7 illustrent les opérations de β-réduction menant à la forme normale *((taquine Pierre) Charles)* qui est l'interprétation sémantique fonctionnelle de l'énoncé *Charles taquine Pierre*.

3.7.1 La coordination

La Grammaire Catégorielle Combinatoire Applicative fournit également des méthodes pour traiter les cas de coordination.

Elles sont basées sur la considération que deux unités linguistiques peuvent être coordonnées pour donner une unité linguistique de type X si et seulement si chaque

unité est de type X (Steedman, 1989; Barry, Pickering, 1990). Elles sont également fondées sur deux hypothèses (Desclés, Biskri, 1996) :

(i) La catégorie construite qui suit immédiatement la conjonction *et* détermine le type de la coordination. Cela a pour effet de suspendre temporairement l'analyse quasi-incrémentale lorsque nous rencontrerons la conjonction *et* afin de construire immédiatement le second membre de la coordination;

(ii) Quand nous avons une coordination de type X définie par l'hypothèse 1, le premier membre de la coordination est la catégorie de type X qui précède immédiatement la conjonction.

Considérant ces hypothèses, nous donnerons à une conjonction de coordination telle que *et* le type syntaxique (X\X)/X, où X est un type abstrait qui empruntera le type du membre de la coordination qui le succède.

Analysons par exemple l'énoncé *Charles apprécie et Pierre déteste Patrick* :

(1)	[N : Charles] - [(S\N)/N : apprécie] - [(X\X)/X : et] - [N : Pierre] - [(S\N)/N : déteste] - [N : Patrick]	
(2)	[S/(S\N) : (**C**$_*$ Charles)] - [(S\N)/N : apprécie] - [(X\X)/X: et] - [N : Pierre] - [(S\N)/N : déteste] - [N : Patrick]	>**T**
(3)	[S/N : (**B** (**C**$_*$ Charles) apprécie)] - [(X\X)/X: et] - [N : Pierre] - [(S\N)/N : déteste] - [N : Patrick]	>**B**
(4)	[S/N : (**B** (**C**$_*$ Charles) apprécie)] - [(X\X)/X: et] - [S/(S\N) : (**C**$_*$ Pierre)] - [(S\N)/N : déteste] - [N : Patrick]	>**T**
(5)	[S/N : (**B** (**C**$_*$ Charles) apprécie)] - [(X\X)/X: et] - [S/N : (**B** (**C**$_*$ Pierre) déteste)] - [N : Patrick]	>**B**
(6)	[S/N : (**B** (**C**$_*$ Charles) apprécie)] - [(S/N)\(S/N) : (et (**B** (**C**$_*$ Pierre) déteste))] - [N : Patrick]	>
(7)	[S/N : ((et (**B** (**C**$_*$ Pierre) déteste)) (**B** (**C**$_*$ Charles) apprécie))] - [N : Patrick]	<

(8)	[S : (((et (**B** (**C**. Pierre) déteste)) (**B** (**C**. Charles) apprécie)) Patrick)]	>
(9)	(((et (**B** (**C**. Pierre) déteste)) (**B** (**C**. Charles) apprécie)) Patrick)	
(10)	(((**Φ**∧ (**B** (**C**. Pierre) déteste)) (**B** (**C**. Charles) apprécie)) Patrick)	(et = **Φ**∧)
(11)	(∧ ((**B** (**C**. Charles) apprécie) Patrick) ((**B** (**C**. Pierre) déteste) Patrick))	(**Φ**)
(12)	(∧ ((**C**. Charles) (apprécie Patrick)) ((**B** (**C**. Pierre) déteste) Patrick))	(**B**)
(13)	(∧ ((apprécie Patrick) Charles) ((**B** (**C**. Pierre) déteste) Patrick))	(**C**.)
(14)	(∧ ((apprécie Patrick) Charles) ((**C**. Pierre) (déteste Patrick))	(**B**)
(15)	(∧ ((apprécie Patrick) Charles) ((déteste Patrick) Pierre))	(**C**.)

L'étape 3 construit le premier membre de la coordination, *(**B** (**C**. Charles) apprécie)*, de type S/N. Nous rencontrons alors la conjonction *et*, ce qui fait en sorte que nous devons immédiatement construire le second membre. Nous l'obtenons à l'étape 5 avec *(**B** (**C**. Pierre) déteste)*, également de type S/N. La correspondance entre les types syntaxiques de chacun des membres de la coordination permet d'appliquer la conjonction avec le membre de droite. Le type de *et* est ((S/N)\(S/N))/(S/N), puisque le type abstrait X est celui du membre avec lequel il s'applique, c'est-à-dire (S/N). Une application arrière joint le membre de gauche à la conjonction. Enfin, une application avant nous donne le type S pour l'expression *(((et (**B** (**C**. Pierre) déteste)) (**B** (**C**. Charles) apprécie)) Patrick)*.

Les étapes 9 à 15 illustrent la réduction des combinateurs jusqu'à l'obtention de l'interprétation sémantique fonctionnelle. À l'étape 10, la conjonction *et* est remplacée par le combinateur de composition distributive **Φ** suivi du symbole ∧ en guise de connecteur logique entre les deux membres de la coordination. L'expression de β-réduction du combinateur **Φ** à l'étape 11 est donnée par **Φ** u_1 u_2 u_3 u_4 ➔ u_1 (u_2 u_4) (u_3 u_4). La réduction de tous les combinateurs nous donne finalement la forme normale *(∧ ((apprécie Patrick) Charles) ((déteste Patrick) Pierre))*.

3.7.2 Les métarègles

Les règles de changement de type constituent l'aspect des grammaires catégorielles le plus controversé, puisque leur utilisation abusive peut mener à un problème d'explosion combinatoire. Toutefois il a été démontré qu'elles sont essentielles à l'analyse syntaxique de certains cas de coordination avec ellipses (Dowty, 2000 ; Steedman, 2000 ; Biskri, Desclés, 2006).

Afin d'éviter ce problème, le formalisme de la Grammaire Catégorielle Combinatoire Applicative inclut un certain nombre de métarègles qui permettent de contrôler les changements de type pour ainsi éviter l'explosion combinatoire (Desclés, Biskri, 1996).

À la façon d'un déclencheur, ces métarègles ont pour rôle de déterminer si une règle de changement de type doit être utilisée et, le cas échéant, choisir la règle spécifique à appliquer. La majorité d'entre elles construisent des types syntaxiques qui pourront ensuite être combinés par l'utilisation de règles d'application et de composition.

Sans faire l'étalage de l'ensemble des métarègles, nous nous contenterons de détailler deux d'entre elles, c'est-à-dire celles qui seront utilisées dans les exemples qui suivront dans ce mémoire. Le lecteur pourra retrouver d'autres métarègles dans (Desclès, Biskri, 1996).

Métarègle 1 : Soient u_1 et u_2 dans l'expression concaténée u_1-u_2. Si u_1 est de type N et u_2 de type (Y\N)/Z alors la règle **>T** est appliquée à u_1. Ainsi :

$$[N : u_1 \rightarrow Y/(Y\backslash N) : (C_* u_1)]$$

Prenons la phrase *Patrick boit du jus* :

(1) [N : Patrick] - [(S\N)/N : boit] - [N/N : du] - [N : jus]

(2)	[S/(S\N) : (**C**$_*$ Patrick)] - [(S\N)/N : boit] - [N/N : du] - [N : jus]	>**T**, **M1**
(3)	[S/N : (**B** (**C**$_*$ Patrick) boit)] - [N/N : du] - [N : jus]	>**B**
(4)	[S/N : (**B** (**B** (**C**$_*$ Patrick) boit) du)] - [N : jus]	>**B**
(5)	[S : ((**B** (**B** (**C**$_*$ Patrick) boit) du) jus)]	>

À l'étape 1, nous avons l'argument *Patrick* de type N suivi de l'argument *boit* de type (S\N)/N, qui correspond au schéma-type (Y\N)/Z de la métarègle 1, où Y = S. Un changement de type avant donnera alors l'opérateur ***(C* Patrick)*** de type S/(S\N) qui pourra se composer avec l'argument *boit* et ainsi de suite, jusqu'à l'obtention du type S. C'est cette même métarègle qui a été utilisée plus tôt pour l'exemple *Charles taquine Pierre*.

Métarègle 7 : Si u_1 est de type N et précédé de *et* et u_2 de type Y\X, alors la règle <**T** est appliquée à u_1.

$$[N : u_1 \rightarrow X\backslash(X/N) : (\mathbf{C}_* \; u_1)]$$

Examinons partiellement la phrase *Patrick regarde Charles sévèrement et Pierre bêtement* seulement dans le but de montrer l'utilisation de la règle qui permettra dans une prochaine section l'analyse complète d'un exemple similaire.

(1)	[N : Patrick] - [(S\N)/N : regarde] - [(S\N)\(S\N) : sévèrement] - [N : Charles] - [(X\X)/X: et] - [N : Pierre] - [(S\N)\(S\N) : bêtement]	
…		
(10)	… - [(X\X)/X: et] - [N : Pierre] - [(S\N)\(S\N) : bêtement]	
(11)	… - [(X\X)/X: et] - [(S\N)\((S\N)/N) : (**C**$_*$ Pierre)] - [(S\N)\(S\N) : bêtement]	>**T**, **M7**
(12)	… - [(X\X)/X: et] - [(S\N)\((S\N)/N) : (**B** bêtement (**C**$_*$ Pierre))]	<**B**
…		

À l'étape 10, nous retrouvons les conditions de déclenchement de la métarègle 7 avec l'argument *Pierre* de type N et *bêtement* de type (S\N)\(S\N). Le type X correspond à (S\N). Par conséquent, nous obtenons l'expression *(**C** *Pierre)* de type (S\N)\((S\N)/N), qui se composera ensuite avec *bêtement* à l'étape 12 pour former *(**B** bêtement (**C** *Pierre))*.

3.7.3 Réorganisation structurelle

La Grammaire Catégorielle Combinatoire Applicative offre également un certain nombre de mécanismes visant à permettre la réorganisation structurelle des expressions, dont celui de la décomposition telle que proposé par Steedman.

Si la décomposition s'avère très utile dans bien des cas, elle est toutefois confrontée à d'importantes limites. Tout d'abord, elle n'offre aucune stratégie clairement établie pour identifier le type syntaxique à octroyer aux éléments décomposés. De surcroît, la méthode est inopérante face à certaines formes elliptiques de la coordination.

Prenons la phrase *Patrick boit du lait souvent et du jus rapidement* :

(1)	[N : Patrick] - [(S\N)/N : boit] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	
(2)	[S/(S\N) : (**C*** Patrick)] - [(S\N)/N : boit] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	**>T, M1**
...		
(5)	[S : ((**B** (**B** (**C*** Patrick) boit) du) lait)] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	>

À l'étape 5, nous obtenons *((**B** (**B** (**C*** Patrick) boit) du) lait)* de type S. Il s'agit d'un faux constituant, puisque tout l'énoncé n'a pas encore été considéré. Comme nous l'avons vu, la stratégie à utiliser dans ce cas est celle proposée par Steedman, qui consiste en une décomposition du faux constituant en une nouvelle expression

concaténée, selon le principe de la neutralité paramétrique. Dans la Grammaire Catégorielle Combinatoire Applicative, cette décomposition se traduit en ces deux règles :

$[X : (u_1\ u_2)]$	$[X : (u_1\ u_2)]$
----------->**dec** ;	----------<**dec**
$[X/Y : u_1]$ - $[Y : u_2]$	$[Y : u_2]$ - $[X\backslash Y : u_1]$

Notons que les règles de décomposition >**dec** et <**dec** sont respectivement les règles inverses de l'application avant et arrière.

À l'étape 5, le faux constituant n'est pas décomposable puisqu'il n'a pas une structure *(u_1 u_2)*. L'analyseur tentera donc de réduire les combinateurs un à un jusqu'à ce qu'il puisse être décomposé ou jusqu'à ce que tous les combinateurs aient été réduits. Dans notre cas, une première réduction permet d'obtenir l'expression *((**B** (**C*** Patrick) boit) (du lait))*.

Par la suite, une décomposition avant produit d'une part l'expression *(**B** (**C*** Patrick) boit)* de type S/(S\N) et l'expression *(du lait)* de type S\N, qui par une règle d'application arrière pourra se combiner à *souvent*, de type (S\N)\(S\N). Ainsi, en reprenant de l'étape 5 :

(5)	[S : ((**B** (**C*** Patrick) boit) du) lait)] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	
(6)	[S: ((**B** (**C*** Patrick) boit) (du lait))] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	(**B**)
(7)	[S/(S\N) : (**B** (**C*** Patrick) boit)] - [S\N : (du lait)] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	>**dec**
(8)	[S/(S\N) : (**B** (**C*** Patrick) boit)] - [S\N : (souvent (du lait))] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	<

(9)	[S : ((**B** (**C**$_*$ Patrick) boit) (souvent (du lait)))] - [(X\X)/X : et] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : rapidement]	>

La réduction du combinateur est décrite à l'étape 6 et la décomposition à l'étape 7. L'étape 9 permet d'obtenir le type S pour la première partie de l'énoncé.

Puisque nous rencontrons la conjonction *et*, nous devons immédiatement construire le second membre de la coordination :

(10)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	>
(11)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [N: (du lait)] - [(S\N)\(S\N) : rarement]	>
(12)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**C**$_*$ (du lait))] - [(S\N)\(S\N) : rarement]	<**T, M7**
(13)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**B** rarement (**C**$_*$ (du lait)))]	<**B**

À l'étape 13, nous obtenons le second membre de la coordination, soit l'expression *(**B** rarement (**C**$_*$ (du lait))*. Cependant, le type des expressions avant et après la conjonction *et* ne sont pas les mêmes et ils ne peuvent donc pas s'appliquer. Nous devons donc à nouveau opérer une décomposition pour tenter d'extraire le premier membre de la coordination. Nous obtiendrions alors le type catégoriel S/((S\N)\((S\N)/N)), qui ne correspond à aucune interprétation sémantique et ne nous permet pas de poursuivre l'analyse. La décomposition ne nous est plus d'aucun recours.

À ce problème, la Grammaire Catégorielle Combinatoire Applicative propose une autre avenue qui est celle d'une *réorganisation structurelle* basée sur le principe que les combinateurs de la logique combinatoire dans les expressions combinatoires véhiculent un contenu sémantique intrinsèque qui joue le rôle d'une mémoire tout le long de l'analyse. C'est ce qui permettra de systématiser une réorganisation structurelle des

constituants. Cette réorganisation structurelle se distingue du processus de décomposition par sa considération de l'interprétation sémantique fonctionnelle.

Le processus de réorganisation nécessite deux étapes pour extraire le premier membre de la coordination d'un faux constituant.

Une première étape demande à tester si le type de l'opérande peut être combiné au type de l'unité linguistique résiduelle. Si c'est le cas alors nous procédons à la décomposition, sinon nous réduisons un combinateur. Le processus se répète jusqu'à ce que la décomposition soit possible ou lorsqu'il n'y aura plus de combinateur. Dans l'exemple précédent, nous obtiendrions l'expression (*souvent* (*boit* (*du jus*))) de type S\N. Cette expression contient le premier membre de la coordination.

La dernière étape consiste à construire pour ce membre une structure combinatoire équivalente à celle du second membre de la coordination. Par la suite, nous lui appliquerons une décomposition pour isoler le premier membre de la coordination qui aura en conséquence un type catégoriel identique à celui du deuxième membre de la coordination. La conjonction *et* pourra dès lors se composer avec ses deux membres.

Dans la première version de la Grammaire Catégorielle Combinatoire Applicative, des règles d'équivalence logiques[15] sont suggérées afin de permettre la restructuration du membre. Par exemple :

(i) $(u_1\ (u_2\ u_3)) \Longleftrightarrow ((\mathbf{B}\ u_1\ u_2)\ u_3)$

(ii) $((u_1\ u_2)\ u_3) \Longleftrightarrow ((\mathbf{B}\ (\mathbf{C}_*\ u_3)\ u_1)\ u_2)$

(iii) $(u_1\ (u_2\ u_3)) \Longleftrightarrow ((\mathbf{B}\ u_1\ (\mathbf{C}_*\ u_3))\ u_2)$

(iv) $((u_1\ u_2)\ u_3) \Longleftrightarrow ((\mathbf{B}\ (\mathbf{C}_*\ u_3)\ (\mathbf{C}_*\ u_2))\ u_1)$

[15] Ces équivalences sont directement la conséquence d'opérations de β-réduction.

Toutefois, cette méthode éprouve des limites. En effet, elle ne nous assure pas qu'une équivalence pourra être trouvée : des cas autres que ceux prévus dans le dictionnaire des équivalences pourraient se présenter. De plus, elle ne repose que sur une stratégie de comparaison et de substitution.

Nous avons donc besoin d'enrichir le modèle de la Grammaire Catégorielle Combinatoire Applicative d'une méthode permettant une restructuration **automatique** d'un constituant.

CHAPITRE 4

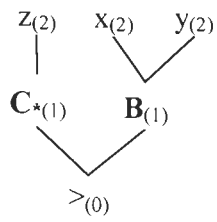
INTRODUCTION DES COMBINATEURS

Nous proposerons dans cette section une solution qui permettra de restructurer automatiquement le membre d'une coordination de manière à ce que la structure de l'expression combinatoire de ce dernier soit équivalente à celle du second membre.

Concrètement, cette solution consistera à étudier le principe que la logique combinatoire est sans variable et que ce sont les combinateurs qui jouent le rôle de « mémoire ». Cette mémoire est celle de l'action du combinateur, intrinsèquement véhiculée lors de l'opération de β-réduction. En effet, il suffit de réduire un combinateur de son expression pour retrouver l'état de la structure précédant son introduction.

4.1 Représentation des expressions combinatoires

Dorénavant, nous présenterons les expressions combinatoires au moyen d'une structure en arbre, afin de faciliter la lecture. Nous indiquerons en indice et entre parenthèses le niveau de profondeur des noeuds dans l'arbre, le niveau 0 correspondant à la racine, qui sera par ailleurs toujours une expression applicative (>). Par exemple, « $\mathbf{C}_{*(2)}$ » signifie que le combinateur $\mathbf{C}_*$ se trouve à deux niveaux de profondeur supplémentaires par rapport à la racine. Par exemple, considérons l'expression combinatoire suivante : ((**C*** z) (**B** x y)). L'arbre correspondant serait dans ce cas celui-ci :



Cette structure sera également très importante au moment de présenter l'algorithme d'introduction de combinateurs, puisque l'expression y sera traitée sous cette forme. Aussi, nous utiliserons couramment les abréviations « G » pour « gauche » et « D » pour « droite » pour signifier le premier et le deuxième argument d'un opérateur (qui sera donc l'enfant de gauche ou de droite dans l'arbre). Nous nous en servirons également pour indiquer les chemins à emprunter dans l'arbre pour se rendre à des noeuds en particulier, dans la plupart du temps conjointement avec le niveau de profondeur du noeud dans l'arbre, où le nombre de directions à emprunter correspondra conséquemment à la profondeur. En indice, on inscrira d'abord le niveau, « : », ainsi que la liste ordonnée des directions pour se rendre jusqu'au noeud. Par exemple, une expression applicative « x », situé à gauche puis à droite de la racine serait noté $x_{(2 :GD)}$.

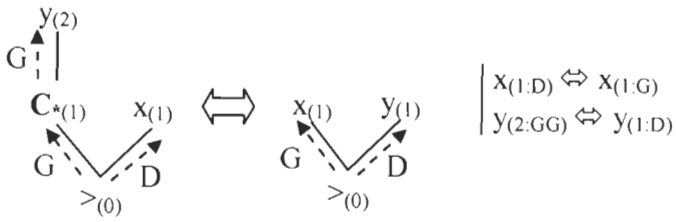
4.2 Étude de la β-réduction des combinateurs

Nous allons maintenant observer ce qui se produit au niveau des expressions applicatives lorsque nous effectuons une β-réduction pour les combinateurs **B** et **C***.

Nous nous limiterons dans le cadre de nos recherches à l'exploitation de ces deux combinateurs, bien qu'il en existe davantage, mais ceux-ci s'avéreront suffisant pour démontrer la stratégie et, comme nous le soulignerons ultérieurement, rien ne nous empêchera d'utiliser la méthode proposée pour l'appliquer aux autres combinateurs.

4.2.1 Le combinateur de changement de type (C*)

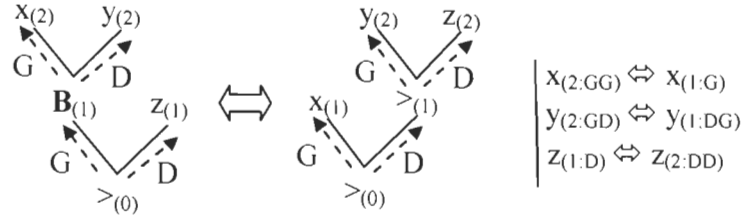
L'expression formelle « littéraire » qui décrit la β-réduction de ce combinateur est : ((**C*** y) x) ➔ (x y). Sous forme d'arborescence, nous obtiendrons par conséquent ceci:



On peut noter dans le schéma précédent une variation non seulement en termes de niveaux, mais aussi de directions. Plus spécifiquement, après la β-réduction, l'unité « x » passe de droite à gauche. Puis, afin d'atteindre l'unité « y », nous devons aller à gauche à deux reprises, mais une fois que le combinateur **C*** est réduit, le noeud perd un niveau et est directement à la droite de l'application racine.

4.2.2 Le combinateur de composition fonctionnelle (B)

Dans le cas du combinateur **B** la β-réduction se fait comme suit : ((**B** x y) z) ➔ (x (y z)). En utilisant un arbre, cela donne :



La β-réduction du combinateur **B** a un impact sur tous les arguments. Initialement, « x » est à gauche de **B**, qui est lui-même à gauche de la racine. Après la réduction, un niveau est déduit et le chemin qui était « gauche, gauche » devient seulement « gauche ». Dans le cas de « y », son niveau ne varie pas, mais ses deux directions changent : « gauche, droite » devient « droite, gauche ». Finalement, « z » augmente d'un niveau et nous devons ajouter une direction « droite » supplémentaire.

4.3 Interprétation

Ces transformations que subissent le niveau et le chemin des arguments sont liées à la fonction d'un combinateur et jouent un rôle capital : elles constituent la clé pour déterminer la position à laquelle un combinateur doit être introduit. Le second aspect à considérer est que dans tous les arbres de β-réduction de combinateurs, l'opération d'application avant à la racine de l'arbre ne varie jamais et représente un point de référence que nous appellerons un « délimiteur fonctionnel », puisqu'il délimite la fonction exercé par le combinateur sur ses arguments.

Conséquemment, l'introduction d'un combinateur passera par l'identification du noeud dans l'arbre que représente ce délimiteur fonctionnel. Afin de l'identifier, nous devrons retrouver le chemin original menant au combinateur avant sa β-réduction (ou de façon synonyme, après son introduction). Alors, puisque les combinateurs $\mathbf{C}*$ et **B** se trouvent à gauche du délimiteur fonctionnel, on retranchera la dernière direction, et on obtiendra l'endroit précis dans l'arbre représentant le second membre de la coordination où on doit effectuer l'introduction. Notons que ce ne sont pas tous les combinateurs qui se retrouvent directement à la gauche de leur délimiteur fonctionne. À titre d'exemple, le combinateur de distributivité $\boldsymbol{\varphi}$ qui est « deux fois à gauche » de son délimiteur.

Aussi, puisque l'élimination des combinateurs dans une expression combinatoire se fait de droite à gauche (d'un point de vue littéraire), le parcours de l'expression combinatoire pour trouver les chemins d'introduction se fera en sens opposé, soit de gauche vers la droite. Dans une représentation en arbre binaire comme nous le considérons, cela revient donc à parcourir l'arbre de la racine vers la feuille, de gauche vers la droite. À chaque fois que nous rencontrerons un combinateur (**B** ou $\mathbf{C}*$), nous devrons rechercher les noeuds qui représentent chacun de ces arguments et modifier les chemins de ces noeuds conséquemment à ce qui a été présenté plus haut, afin d'obtenir la position de ces arguments avant l'introduction du combinateur. Évidemment, ces opérations ne visent pas à changer pas le chemin du combinateur « en cours » (le

combinateur n'existant pas avant son introduction, il ne peut avoir un chemin qu'après son introduction), mais seulement ceux de ses arguments, dont peuvent ou non faire partie d'autres combinateurs qui ont été introduits dans l'expression avant ce dernier. Ainsi, au moment où nous atteindrons un combinateur, son chemin sera nécessairement celui après son introduction, ce qui sera suffisant pour déterminer le délimiteur fonctionnel et l'introduire au bon endroit. Par ailleurs, ces affirmations sous-entendent qu'au départ, nous connaissons déjà la position où sera introduit le premier combinateur de l'expression combinatoire (le dernier à être introduit), et que le dernier combinateur ne peut modifier le chemin d'aucun combinateur, puisqu'il est le premier à être introduit.

4.4 Algorithme

L'algorithme suivant permet de calculer les chemins vers les combinateurs d'une expression combinatoire donnée après leur introduction. L'expression combinatoire est organisée en une structure d'arbre binaire où chaque noeud possède une mémoire contenant le chemin initial jusqu'à ce dernier à partir de la racine (sous la forme d'une liste, par exemple), et que la seule entrée de la méthode est un noeud (en commençant par la racine).

méthode calculerChemin
    si le noeud courant est un combinateur B alors
        remplacer le chemin « GG » par « G » pour les noeuds du premier argument
        remplacer le chemin « GD » par « DG » pour les noeuds du deuxième argument
        trouver les noeuds du troisième argument et changer le chemin « D » par « DD »
    sinon si le noeud courant est un combinateur C alors*
        remplacer le chemin « GG » par « D » pour les noeuds du premier argument
        trouver les noeuds du deuxième argument et changer le chemin « D » par « G »
    fin si
    pour chaque noeud enfant, de gauche vers la droite
        appeler calculerChemin pour ce noeud

fin pour
fin de la méthode

Le procédé s'avère être plutôt simple : il s'agit d'un algorithme récursif qui appelle à tour de rôle chacun des noeuds de l'arbre, de la racine jusqu'aux feuilles, du premier noeud enfant au dernier. Pour chaque noeud, on vérifie s'il s'agit d'un combinateur. Si ce n'est pas le cas, on passe au noeud suivant. Dans un cas positif, on recherche les arguments du combinateur en question et on modifie les chemins selon le combinateur en présence. La recherche des noeuds représentant chaque argument demande à parcourir l'arbre à la recherche de noeuds avec des chemins qui contiennent le chemin jusqu'au délimiteur fonctionnel du combinateur courant, plus le chemin jusqu'à l'argument. Par exemple, si le chemin vers **C*** est (GDG) et le délimiteur fonctionnel est à (GD), puisque le deuxième argument doit être à droite du délimiteur fonctionnel, alors les noeuds du second argument sont ceux qui ont un chemin qui débute par (GDD). Ainsi, des chemins valides pourraient être (GDD), (GDDG), (GDDD), et ainsi de suite. Cependant, on n'aura pas à utiliser cette méthode pour chacun des arguments d'un combinateur, étant donné que le premier et le second argument de **B** et le premier argument de **C*** sont leurs noeuds enfants : la propagation des directions à être modifiées est facilement effectuée en agissant directement sur ces noeuds, sans rechercher un chemin précis. Par contre, le troisième argument de **B** et le second argument de **C*** ne sont pas toujours nécessairement à la même position tels que présentés dans leurs schémas respectifs, à cause des différentes combinaisons possibles entre combinateurs dans l'expression combinatoire (entre autres lorsque plusieurs compositions fonctionnelles se succèdent) et il nous faudra donc utiliser la méthode proposée afin de retracer leurs positions. Toutefois, nous n'aurons pas à fouiller l'arbre en entier pour trouver cet argument – et il sera même très important de ne pas le faire. Tout d'abord, on n'effectuera la recherche que dans les noeuds de l'arbre que nous n'avons pas encore parcouru avec la méthode *calculerChemin*. La raison à cela est que l'introduction d'un combinateur ne peut avoir altéré le chemin d'introduction d'un autre combinateur qui le

précède dans l'expression et qui n'existe pas encore à ce moment-là (les combinateurs étant introduits de la droite vers la gauche). Nous considérons donc qu'un combinateur peut modifier seulement les chemins des éléments qui le suivent dans l'expression combinatoire. Même si un sous-arbre situé avant le combinateur courant correspondrait à l'argument recherché, nous ne modifierions par les chemins. Nous n'avons pas non plus à fouiller le ou les noeuds enfants du combinateur, car ils correspondent déjà à un ou des arguments. Par ailleurs, il faut effectuer la recherche pour l'ensemble des noeuds candidats, car il est possible que l'argument soit dispersé dans différents sous-arbres. Ceci dit, lorsque nous trouvons qu'un noeud appartient à l'argument et qu'il n'est pas une feuille (une unité linguistique), nous sommes assurés que tout le sous-arbre en fait également parti. À l'inverse, si le chemin d'un noeud ne correspond pas à celui que nous recherchons, l'ensemble de ses noeuds enfants non plus. Nous observerons tous ces aspects plus en détails à l'intérieur des exemples qui seront bientôt présentés.

Une fois que tous les noeuds de l'arbre ont été passés en revue et, par conséquent, que les chemins vers chacun des combinateurs après leur introduction ont été calculés, nous pouvons procéder à l'opération d'introduction des combinateurs dans l'expression applicative sans combinateurs (en forme normale) à restructurer. Rappelons-nous que ce que nous souhaitons dans le cadre d'une analyse de la Grammaire Catégorielle Combinatoire Applicative est de réorganiser la structure du faux constituant contenant le premier membre de la coordination afin qu'elle soit similaire à celle du second membre.

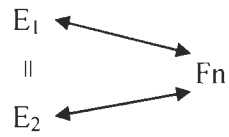
Les combinateurs seront introduits dans l'ordre inverse de leur réduction, soit de droite vers la gauche du point de vue de l'expression combinatoire, en tenant compte des chemins et en ignorant la dernière direction « gauche » pour les raisons expliquées auparavant.

L'aboutissement du processus nous mènera à l'une des deux situations suivantes :

(i) Soit tous les combinateurs auront pu être introduits aux positions calculées et le premier membre aura alors été correctement restructuré. Dans un tel cas, l'analyse pourra se poursuivre;

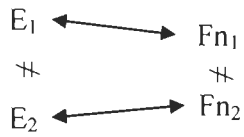
(ii) Soit nous aurons une situation où le combinateur ne peut être introduit à l'endroit calculé, car la position n'existe pas ou ne permet pas l'exécution d'une opération de β-réduction pour ce combinateur. Dans ce cas, l'analyse ne pourra pas se compléter et cela signifiera que la phrase n'est probablement pas syntaxiquement bien construite.

Ces conclusions puisent leurs sources dans le théorème de Church-Rosser que nous avons présenté plus tôt. Revenons-y en considérant deux expressions combinatoires différentes, E_1 étant que premier membre de la coordination et E_2 le second. Le théorème stipule que si E_1 possède une forme normale, celle-ci est unique, et par conséquent si la réduction de E_1 mène à une forme normale Fn et que la réduction de E_2 mène également à Fn alors par confluence E_1 et E_2 sont deux expressions combinatoires équivalentes.



En d'autres mots, notre méthode calcule les étapes d'introduction de combinateurs qui permettent de construire E_2 (le second membre de la coordination) à partir de Fn. Si par la réduction de E_1 nous obtenons Fn alors nécessairement l'introduction des combinateurs dans cette forme normale construira un membre ayant une structure équivalente au second. Nous aurons alors la situation que nous avons décrite en (i).

À l'opposé, si E_1 et E_2 ne possèdent pas la même forme normale alors d'aucune façon nous arriverons à reconstruire E_1 afin que sa structure soit équivalente à celle de E_2.



Cela correspond à la situation (ii). Il se produira dans le cas où la structure syntaxique de la phrase à analyser est invalide. La conséquence sera que l'application des règles de la Grammaire Catégorielle Combinatoire Applicative dans l'expression applicative typée mènera à la construction d'une expression combinatoire erronée pour le premier ou le second membre de la coordination (ou les deux).

4.5 Application de l'algorithme

Voyons maintenant concrètement l'exécution de l'algorithme à travers des exemples croissants en complexité qui permettront d'illustrer plus clairement les propos tenus précédemment. Nous supposerons qu'à chaque fois, nous avons une expression combinatoire pour laquelle nous recherchons les chemins d'introduction des combinateurs que nous insérerons, par la suite, dans la forme normale de l'expression.

Notons que dans le cadre des exemples qui suivront, il est tout à fait volontaire de notre part de réintroduire les combinateurs dans la forme normale de la même expression. Ceci aura pour but de démontrer que si les chemins calculés sont les bons et que si le premier membre a été bien construit, l'introduction sera possible.

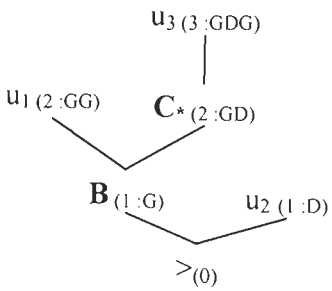
Un premier schéma représentera l'arbre binaire de l'expression combinatoire. Par la suite, afin d'alléger le nombre de schémas représentant l'exécution de l'algorithme, nous ne représenterons que les étapes où un combinateur est rencontré. Ce dernier sera encerclé dans le schéma. Par ailleurs, les directions soulignées à l'intérieur des chemins seront celles qui auront été modifiées par l'exécution de l'algorithme pour le combinateur courant, pour chacun des arguments. Finalement, en considérant les chemins trouvés et l'ordre d'introduction, les combinateurs seront introduits un par un à

partir de la forme normale. Des schémas supplémentaires illustreront le processus, à l'intérieur desquels le délimiteur fonctionnel sera encerclé et, s'il y a lieu, des flèches pointillées accompagnées d'une direction indiqueront le chemin qui a été suivi pour atteindre ce délimiteur, en concordance avec le chemin calculé des combinateurs. Le fait de parvenir à l'expression combinatoire initiale indiquera que l'algorithme a bien fonctionné. Aussi, dans tous les exemples de cette section, nous utiliserons la notation u_i afin de représenter les unités linguistiques de l'expression combinatoire.

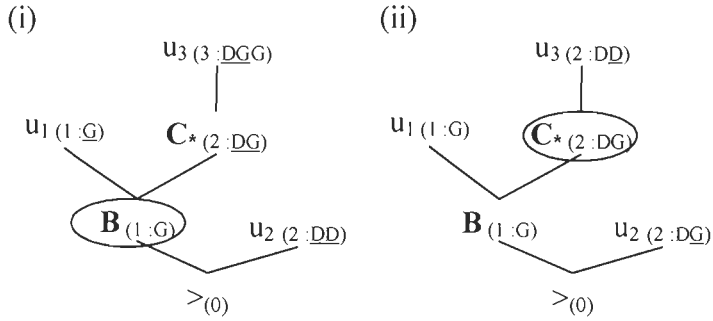
Nous verrons trois exemples. Le premier consistera en un cas de base qui permettra de bien comprendre les notions fondamentales de l'algorithme. L'exemple 2 illustrera un cas particulier où le dernier argument d'un combinateur se trouve avant celui-ci dans l'expression. Le dernier exemple visera à démontrer que l'algorithme fonctionne même avec des expressions combinatoires beaucoup plus complexes. Plus particulièrement, il présentera une situation où le dernier argument est dispersé dans plusieurs branches de l'arbre binaire.

4.5.1 Exemple 1 : ((B u_1 (C_* u_3)) u_2)

Nous débuterons avec un exemple simple : une expression combinatoire avec deux combinateurs. L'arbre binaire représentant l'expression est le suivant :



Voici maintenant les différentes étapes de l'exécution de l'algorithme pour chacun des combinateurs :



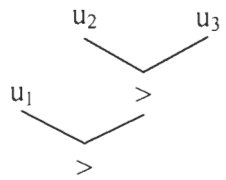
L'arbre étant parcouru de la racine vers les feuilles, de gauche à droite, le premier combinateur à être rencontré est le combinateur **B** à gauche de la racine, comme illustré à l'étape (i). Son premier argument est son noeud enfant de gauche, u_1, dont le chemin passera de « G » à « GG ». Son deuxième argument est (**C*** u_3), pour lesquelles les deux premières directions « GD » sont inversées afin de devenir « DG ». Enfin, en ce qui concerne le troisième argument, le délimiteur fonctionnel du combinateur étant la racine, il faut rechercher les noeuds dont le chemin débute par « D ». Cela correspond à u_2 et « D » deviendra « DD ».

À l'étape (ii), nous atteignons le combinateur **C***. Son noeud enfant, u_3, correspond à son premier argument. Ce noeud perd alors un niveau, soit la dernière direction. « DGG » devient donc « DG ». Pour ce qui est du dernier argument, nous pouvons remarquer que nous ne sommes pas en présence d'une structure comme celle présentée pour le combinateur **C*** au tout début du chapitre, dû à la présence de la composition fonctionnelle dans l'expression. Peu importe, la démarche est la même : le chemin vers **C*** est « DG », ce qui signifie que son délimiteur fonctionnel est situé à droite de la racine, et que le deuxième argument englobe tous les noeuds dont le chemin débute par « DD », ce qui est le cas du noeud u_2, pour lequel « DD » devient « DG ». Puisqu'il n'y

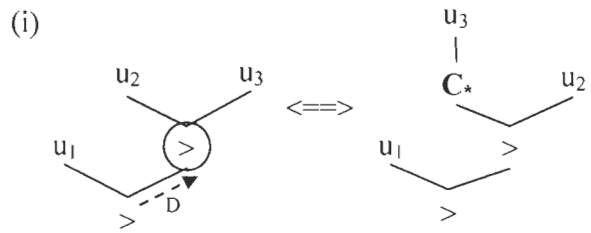
a pas d'autres combinateurs, aucunes autres modifications ne seront effectuées et l'algorithme récursif se terminera après avoir parcouru les noeuds restant.

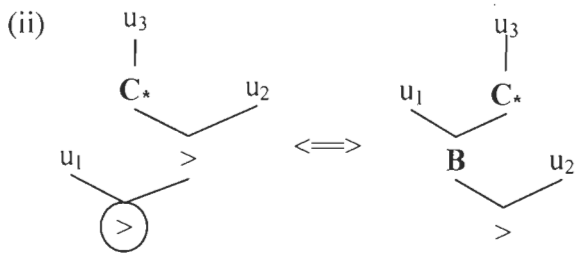
Maintenant que nous connaissons la position des combinateurs après leurs introductions, il suffit de les insérer un à un à partir de la forme normale, dans l'ordre inverse qu'ils apparaissent dans l'expression combinatoire. L'ordre d'introduction sera donc : $\mathbf{C}_{*(2:DG)}$, $\mathbf{B}_{(1:G)}$. Comme nous devrons atteindre le délimiteur fonctionnel pour effectuer l'introduction, rappelons que nous devrons ignorer la dernière direction « gauche », tant pour le combinateur $\mathbf{C}_*$ que **B**.

La forme normale de l'expression combinatoire ((**B** u_1 (**C**$_*$ u_3)) u_2) est (u_1 (u_2 u_3)) et construit l'arbre binaire qui suit :



Les deux combinateurs s'introduisent tels qu'illustrés par les schémas ci-contre :



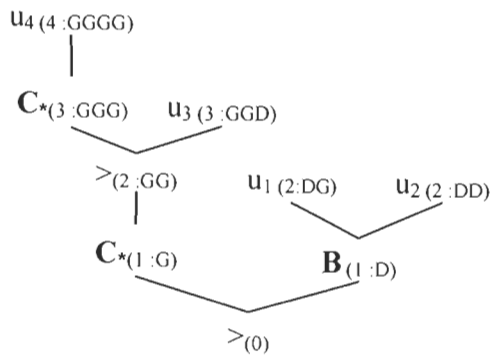


En ignorant la dernière direction du chemin du combinateur $\mathbf{C}_{\star}$, il ne reste que « D », ce qui signifie que son délimiteur fonctionnel est à droite de la racine et l'introduction se fera à partir de ce noeud de référence, comme illustré en (i). Enfin, à l'étape (ii), le chemin du combinateur B n'a qu'une seule direction et elle est ignorée. La racine est donc le délimiteur fonctionnel et est l'endroit à partir duquel est introduit le combinateur.

L'expression combinatoire obtenue est $((\mathbf{B}\ u_1\ (\mathbf{C}_{\star}\ u_3))\ u_2)$, ce qui correspond exactement à l'expression de départ.

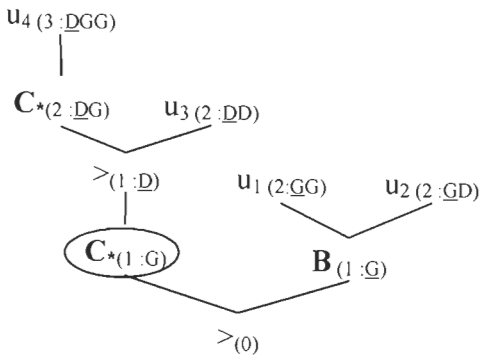
4.5.2 Exemple 2 : $((\mathbf{C}_{\star}\ ((\mathbf{C}_{\star}\ u_4)\ u_3)\ (\mathbf{B}\ u_1\ u_2))$

Poursuivons maintenant avec un exemple légèrement plus complexe qui, du même coup, illustrera une des particularités dont il faut tenir compte lors de la recherche du dernier argument des combinateurs **B** et $\mathbf{C}_{\star}$. L'expression combinatoire peut être traduite par l'arbre binaire ci-dessous :

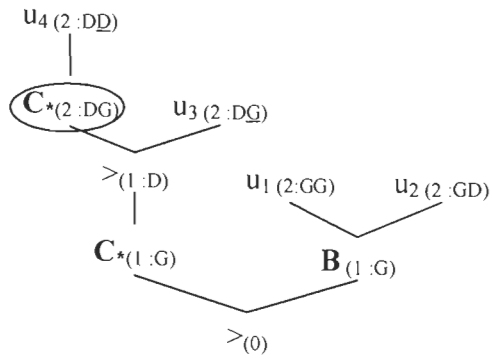


Tout comme ce fut le cas pour l'exemple précédent, nous devons exécuter l'algorithme pour chacun des noeuds de l'arbre, de la racine vers les feuilles, de gauche vers la droite, ce qui donne les étapes intermédiaires suivantes :

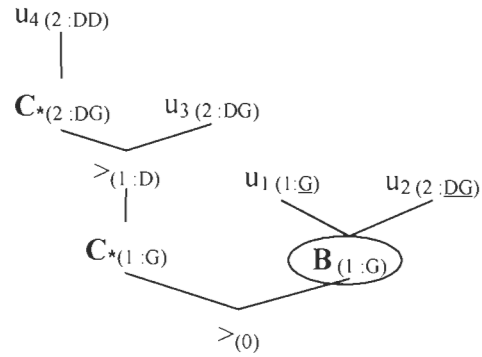
(i)



(ii)



(iii)

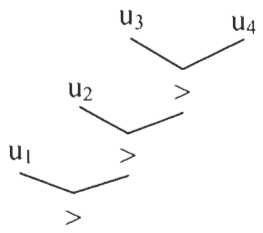


À l'étape (i), nous atteignons le premier combinateur **C**⋆. Le premier argument comporte tous les noeuds inférieurs au combinateur, c'est-à-dire ((**C**⋆ u_4) u_3). Ces noeuds perdent une direction, « GG » devenant « G ». Le délimiteur fonctionnel de **C**⋆ étant la racine, on recherche pour le deuxième argument tous les noeuds dont le chemin débute par « D », ce qui correspond aux noeuds à droite de la racine, dont la première direction des chemins passent de « D » à « G ».

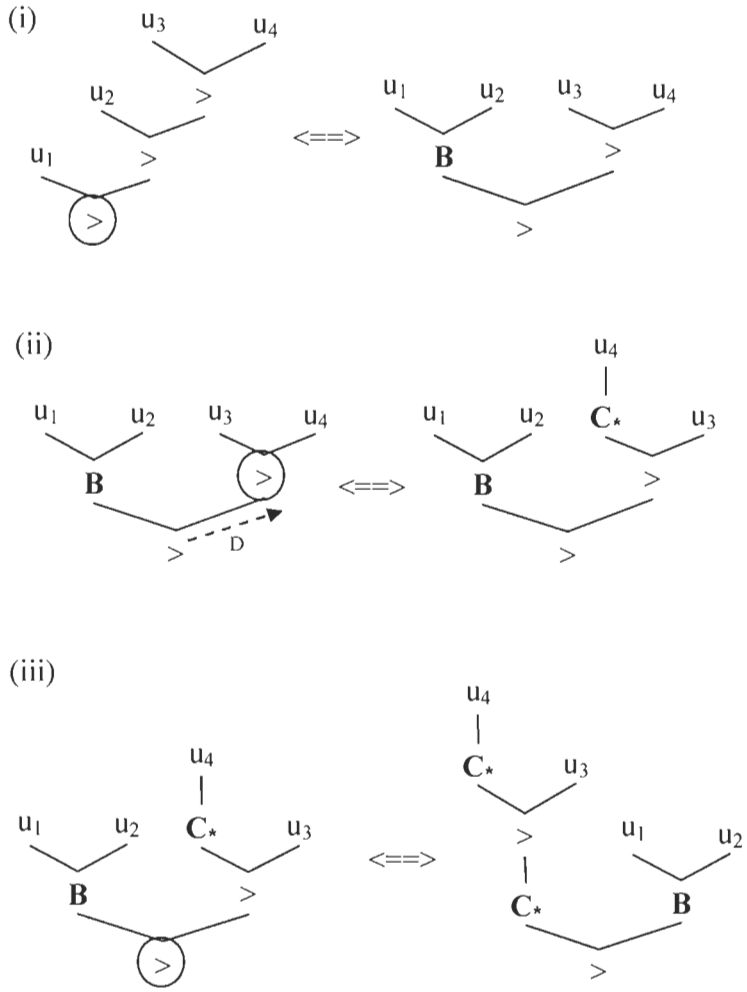
L'étape (ii) nous conduit au second combinateur **C**⋆, pour lequel le premier argument est u_4, qui verra son chemin perdre une fois de plus une direction. Le chemin actuel calculé de C⋆ étant « DG », son délimiteur fonctionnel est à droite de la racine, et le deuxième argument s'avère être u_3, « DG » changeant pour « DD ».

Enfin, à l'étape (iii), nous croisons le dernier combinateur de l'expression. Les deux premiers arguments de **B** sont les noeuds enfants, respectivement u_1 et u_2. u_1 perd une direction gauche, pour passer de « GG » à « G », tandis que les directions de u_2 sont inter-changées, ce qui fait en sorte que « GD » devient « DG ». Il ne reste maintenant plus qu'à trouver le troisième argument de **B**, qui constitue le cas particulier que nous désirions mettre en relief à travers cet exemple. Son délimiteur fonctionnel est la racine, et on devrait rechercher tout noeud dont le chemin débute par « D » et ajouter une direction droite. ((**C**$_*$ u_4) u_3) satisferait à cette condition, mais ces noeuds précèdent le combinateur **B**. Comme nous l'avons mentionné précédemment, un combinateur ne peut modifier le chemin de noeuds déjà visités. Pour cette raison, nous ne modifierons pas les directions de la sous-expression ((**C**$_*$ u_4) u_3). Si nous modifierions tout de même le chemin des noeuds, on modifierait du même coup le chemin déjà calculé du combinateur **C**$_*$, alors que ce dernier n'existerait pas encore au moment d'introduire le combinateur **B**.

Nous avons maintenant terminer de calculer l'ordre d'introduction des combinateurs, qui est $\mathbf{B}_{(1:G)}$, $\mathbf{C}_{*(2:DG)}$, $\mathbf{C}_{*(1:G)}$. Ils seront injectés un à un, à partir d'une expression applicative sans combinateurs équivalente, décrite par (u_1 (u_2 (u_3 u_4))) et représentée par l'arbre ci-contre :



Voici les différentes étapes d'introduction des trois combinateurs :



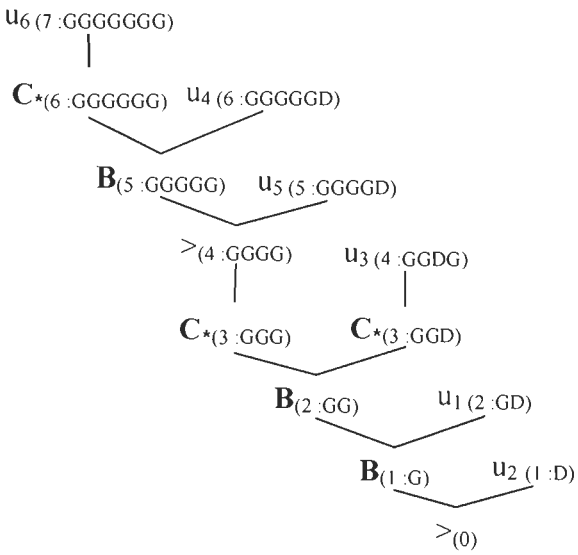
À l'étape (i), nous insérons le combinateur **B**. En se rappelant qu'il nous faut ne pas tenir compte de la dernière direction afin d'identifier le délimiteur fonctionnel qui servira de point de référence pour l'introduction du combinateur, nous trouvons que ce délimiteur est la racine. En (ii), en ignorant la direction gauche, nous devons aller à droite de la racine pour atteindre le délimiteur fonctionnel, d'où l'introduction de **C*** se fera. À l'étape (iii), on introduit le dernier combinateur, **C***, dont le délimiteur fonctionnel s'avère aussi être la racine.

Encore une fois, l'expression combinatoire que nous obtenons possède une structure identique à celle de l'exemple 2, ce qui démontre que l'algorithme a bien fonctionné.

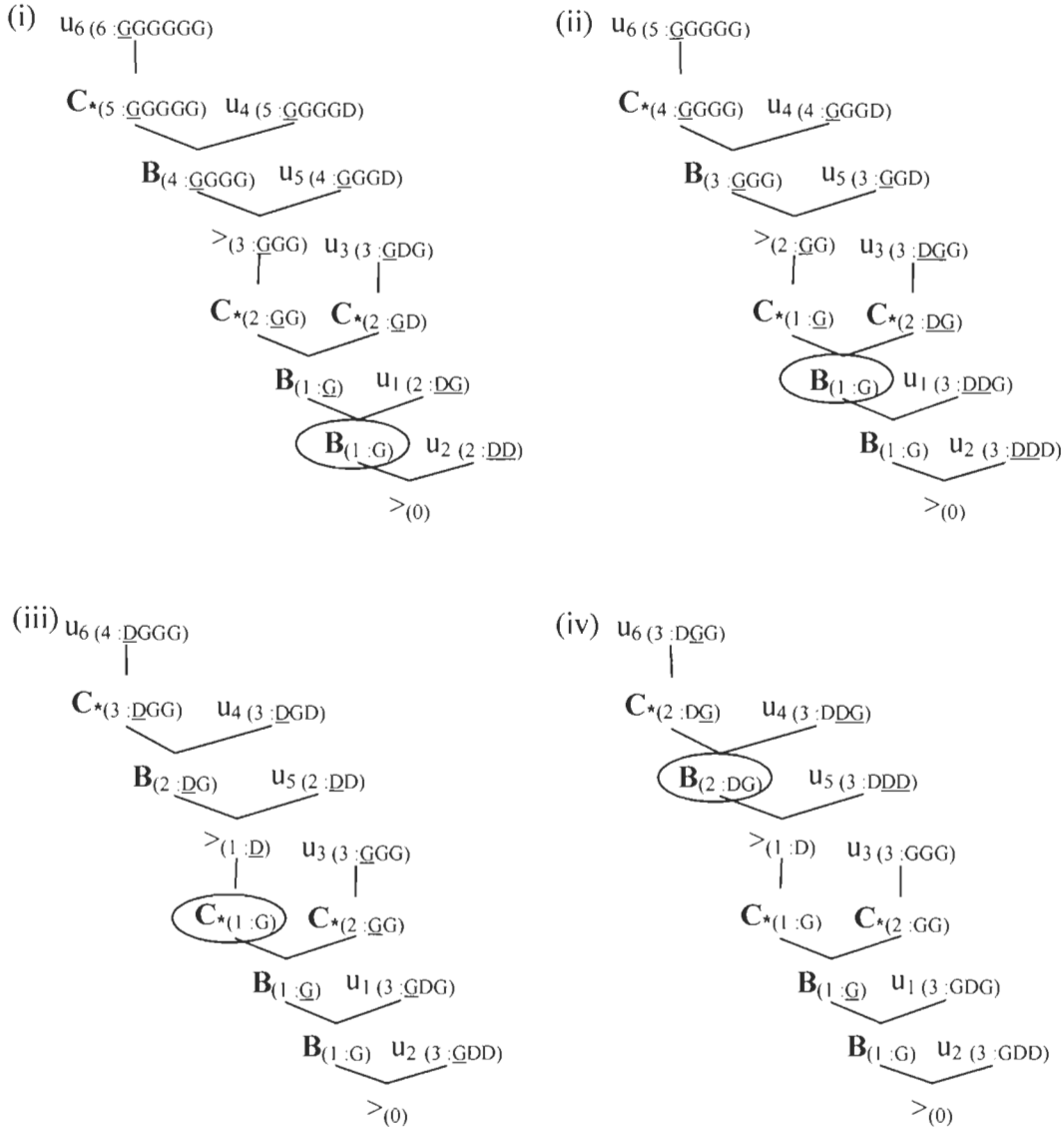
4.5.3 Exemple 3 : ((B ((B (C* ((B (C* u_6) u_4) u_5)) (C* u_3))) u_1) u_2)

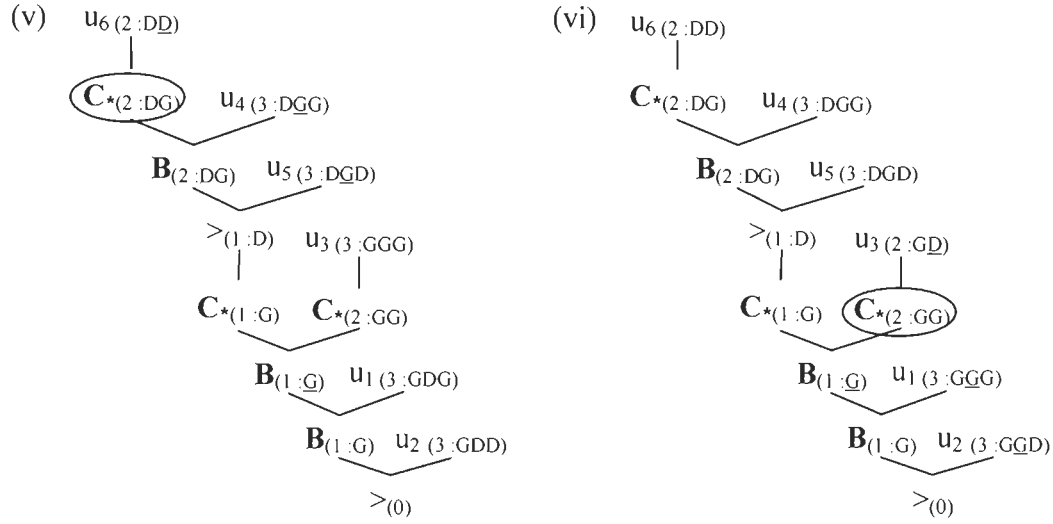
Ce dernier exemple aura pour but de démontrer que l'algorithme permet de calculer les bons chemins d'introduction des combinateurs peu importe la complexité et la structure d'une expression combinatoire.

L'arbre binaire qui représente l'expression combinatoire se construit ainsi :



Les différentes étapes permettant le calcul des chemins pour chaque combinateur sont les suivantes :


(i)
u6 (6 :GGGGGG)
C*(5 :GGGGG) u4 (5 :GGGGD)
B(4 :GGGG) u5 (4 :GGGD)
>(3 :GGG) u3 (3 :GDG)
C*(2 :GG) C*(2 :GD)
B(1 :G) u1 (2 :DG)
B(1 :G) u2 (2 :DD)
>(0)
(ii)
u6 (5 :GGGGG)
C*(4 :GGGG) u4 (4 :GGGD)
B(3 :GGG) u5 (3 :GGD)
>(2 :GG) u3 (3 :DGG)
C*(1 :G) C*(2 :DG)
B(1 :G) u1 (3 :DDG)
B(1 :G) u2 (3 :DDD)
>(0)
(iii)
u6 (4 :DGGG)
C*(3 :DGG) u4 (3 :DGD)
B(2 :DG) u5 (2 :DD)
>(1 :D) u3 (3 :GGG)
C*(1 :G) C*(2 :GG)
B(1 :G) u1 (3 :GDG)
B(1 :G) u2 (3 :GDD)
>(0)
(iv)
u6 (3 :DGG)
C*(2 :DG) u4 (3 :DDG)
B(2 :DG) u5 (3 :DDD)
>(1 :D) u3 (3 :GGG)
C*(1 :G) C*(2 :GG)
B(1 :G) u1 (3 :GDG)
B(1 :G) u2 (3 :GDD)
>(0)



Tout d'abord, à l'étape (i), le premier argument du combinateur **B** est (**B** (**C*** ((**B** (**C*** u_6) u_4) u_5)) (**C*** u_3)). Pour chacun de ces noeuds, les deux premières directions gauches du chemin seront remplacées par une seule. Le deuxième argument est u_1 et « GD » devient « DG ». Le dernier argument correspond à u_2, dont le chemin gagne une direction droite.

À l'étape (ii), nous avons à nouveau un combinateur **B**. Les deux premiers arguments sont (**C*** ((**B** (**C*** u_6) u_4) u_5)) et (**C*** u_3), dont les chemins sont modifiés de la même façon que pour le combinateur précédent. Le troisième argument comprend les noeuds dont le chemin débute par « D », ce qui est le cas de u_1 et u_2, qui sont les enfants de deux noeuds différents.

À l'étape (iii), le premier argument de **C*** est tout ce qui le succède, c'est-à-dire ((**B** (**C*** u_6) u_4) u_5) et « GG » est remplacé par « G ». Son deuxième argument est cependant dispersé à plusieurs endroits dans l'arbre, incluant tous les noeuds dont la première direction est « D », soient les noeuds u_2, u_1 et (**C*** u_3), pour lesquels « D » devient « G ». Dans la recherche de l'argument, au moment où nous atteignons **C***(2 :DG) et que la

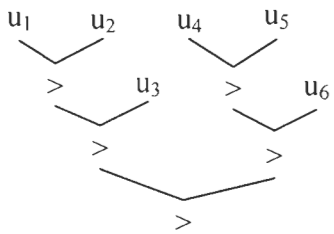
vérification du chemin donne une réponse positive, nous pouvons être confiants que u_3 est également un noeud valide et nous n'avons pas besoin de parcourir ce noeud.

L'étape (iv) nous amène une fois de plus à un combinateur **B**. Le premier argument correspond à ($\mathbf{C}_*$ u_6), tandis que le second argument est u_4. Le chemin calculé jusqu'à **B** étant « DG », le délimiteur fonctionnel est situé à droite de la racine et le troisième argument englobe les noeuds dont le chemin commence par « DD », ce qui donne u_5, qui aura comme nouveau chemin « DDD ».

Le prochain combinateur rencontré, à l'étape (v), est $\mathbf{C}_*$. Le premier argument est bien sûr son noeud enfant, u_6, dont le chemin passera de « DGG » à « DD ». En vertu de la position du délimiteur fonctionnel de $\mathbf{C}_*$, les noeuds du deuxième argument sont ceux dont le chemin débute par « DG », soient les noeuds u_4 et u_5.

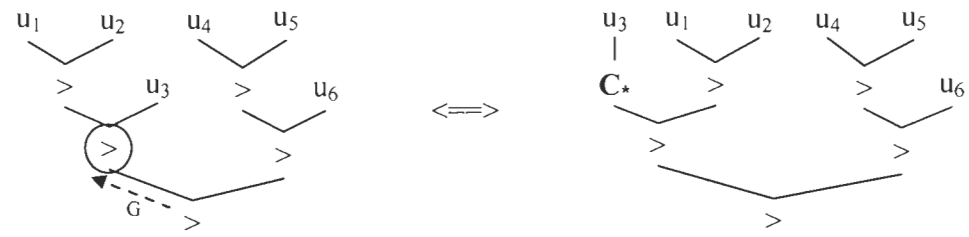
Enfin, à l'étape (vi), le premier argument de C_* est u_3 pour lequel « GGG » changera pour « GD ». Le dernier argument est de nouveau éparpillé dans différentes branches de l'arbre. On doit chercher les noeuds dont les premières directions du chemin sont « GD ». C'est le cas de u_1 et u_2, qui verront leurs chemins être modifiés de « GDG » et « GDD » à « GGG » et « GGD ».

Les calculs étant terminés, il est maintenant temps d'introduire les combinateurs selon les chemins induits. L'ordre d'introduction est $\mathbf{C}_{*(2\ :GG)}$, $\mathbf{C}_{*(2\ :DG)}$, $\mathbf{B}_{(2\ :DG)}$, $\mathbf{C}_{*(1\ :G)}$, $\mathbf{B}_{(1\ :G)}$, $\mathbf{B}_{(1\ :G)}$ et la forme normale de l'expression combinatoire est $(((u_1\ u_2)\ u_3)\ ((u_4\ u_5)\ u_6))$, soit le schéma qui suit :

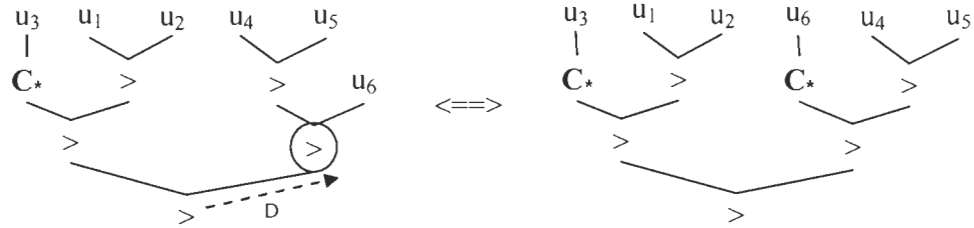


Comme pour les deux exemples précédents, nous retrouverons ci-dessous les schémas illustrant l'introduction de chacun des combinateurs.

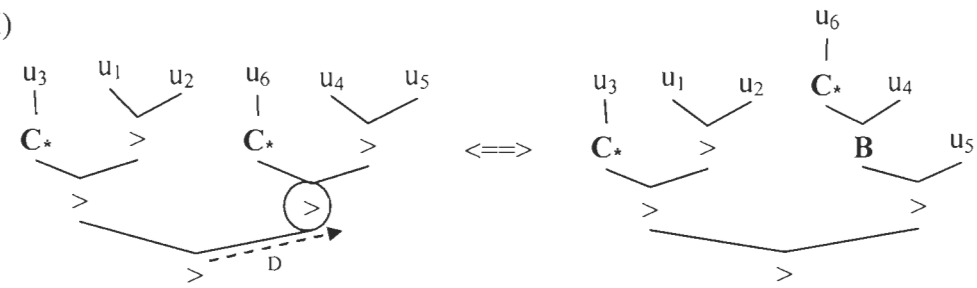
(i)



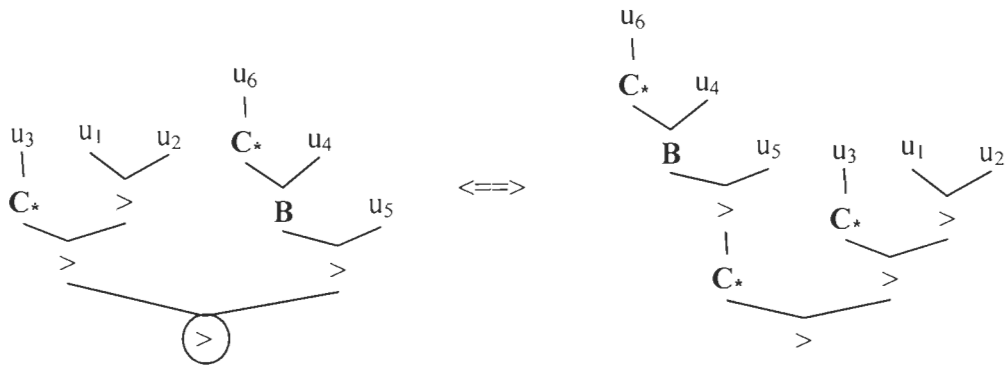
(ii)



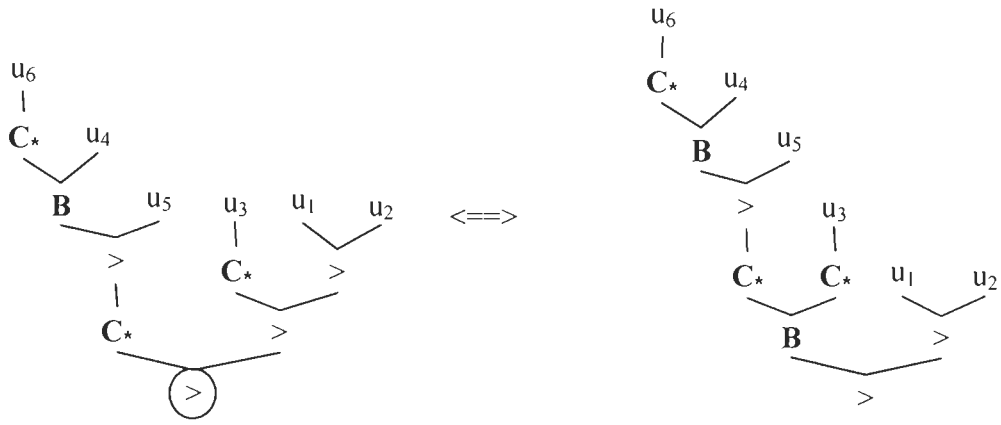
(iii)



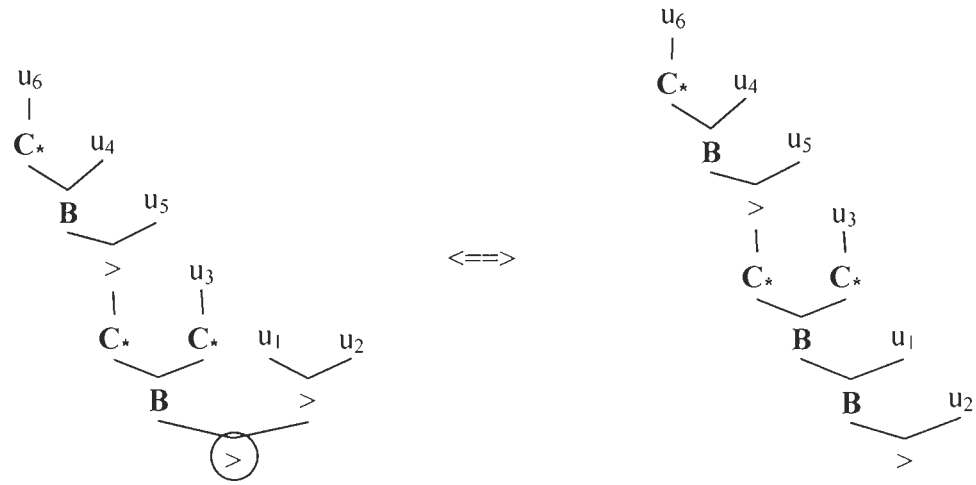
(iv)



(v)



(vi)



À l'étape (i), nous ne tenons pas compte de la dernière direction de $C_{*(2:GG)}$. Nous nous positionnons donc à gauche de la racine – le délimiteur fonctionnel, pour le rappeler – d'où nous procéderons à son introduction. Pour l'étape (ii), nous devons nous diriger à droite de la racine, puisque nous avons $C_{*(2:DG)}$ et que nous ignorons le dernier « G ». L'étape (iii) nous amène à gauche de la racine, tout comme le premier combinateur à avoir été introduit. Finalement, les combinateurs des étapes (iv) à (vi) sont introduits à partir de la racine de l'arbre binaire en restructuration.

Cette dernière étape permet de constater qu'une fois de plus, l'expression combinatoire finale possède la structure souhaitée.

4.6 Application de l'algorithme lors de l'analyse syntaxique

Maintenant que nous avons démontré le fonctionnement de l'algorithme de façon isolée, nous verrons dans cette section deux exemples supplémentaires présentant des cas concrets pour lesquels des phrases contiennent des formes elliptiques de coordination qui entrainent la formation de faux constituants nécessitant une réorganisation structurelle lors de l'analyse de la Grammaire Catégorielle Combinatoire Applicative. Nous décrirons le processus complet de l'analyse, en partant de l'expression applicative pré-typée, jusqu'à l'atteinte de l'interprétation sémantique fonctionnelle.

Pour chaque cas, nous présenterons dans un premier temps les étapes menant à la réorganisation structurelle. Ensuite, nous utiliserons l'algorithme de la même façon que nous l'avons fait dans les exemples de la section précédente. Puis, une fois le membre réorganisé, nous poursuivrons l'analyse jusqu'à la validation syntaxique de l'expression (par l'obtention du type S) et nous la réduirons de ses combinateurs afin d'obtenir la forme normale.

Comme le lecteur pourra le constater, le dernier exemple donné consistera en l'analyse d'une phrase en langue arabe. Nous souhaiterons démontrer que la méthode de réorganisation structurelle ne fait qu'utiliser les règles de β-réduction des combinateurs de la logique combinatoire et est détachée du contexte linguistique sous tendue par l'expression combinatoire. Ainsi, elle fonctionnera tout autant pour toute langue pour laquelle une analyse sous le modèle de la Grammaire Catégorielle Combinatoire Applicative a été démontrée possible[16].

[16] Outre le français, notons principalement l'anglais (Biskri et Desclés, 2006), l'arabe (Biskri et Bensaber, 2008) et le coréen (Kang et Desclés, 2008).

4.6.1 Exemple 1 : Patrick boit du jus souvent et du lait rarement.

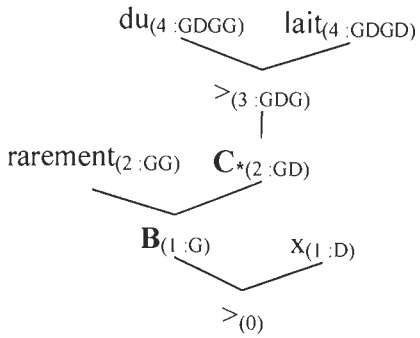
Le premier cas que nous décrirons est en fait celui qui fut présenté à la fin du troisième chapitre. L'analyse va comme suit et la colonne de droite indique la règle de la Grammaire Catégorielle Combinatoire Applicative qui a été appliquée :

(1)	[N : Patrice] - [(S\N)/N : boit] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	
(2)	[S/(S\N) : (**C**$_*$ Patrice)] - [(S\N)/N : boit] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>T, M1**
(3)	[S/N : (**B** (**C**$_*$ Patrice) boit)] - [N/N : du] - [N : jus] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>B**
(4)	[S/N : (**B** (**B** (**C**$_*$ Patrice) boit) du)] - [N : jus] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>B**
(5)	[S: ((**B** (**B** (**C**$_*$ Patrice) boit) du) jus)] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>**
(6)	[S: ((**B** (**C**$_*$ Patrice) boit) (du jus))] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**(B)**
(7)	[S: ((**C**$_*$ Patrice) (boit (du jus)))] - [(S\N)\(S\N) : souvent] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**(B)**
(8)	[S/(S\N): (**C**$_*$ Patrice)] - [S\N : (boit (du jus))] - [(S\N)\(S\N) : souvent] - [(X\X)/X: et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>dec**
(9)	[S/(S\N): (**C**$_*$ Patrice)] - [S\N : (souvent (boit (du jus)))] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**<**
(10)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [N/N : du] - [N : lait] - [(S\N)\(S\N) : rarement]	**>**
(11)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [N: (du lait)] - [(S\N)\(S\N) : rarement]	**>**
(12)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**C**$_*$ (du lait))] - [(S\N)\(S\N) : rarement]	**<T, M7**
(13)	[S: ((**C**$_*$ Patrice) (souvent (boit (du jus))))] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**B** rarement (**C**$_*$ (du lait)))]	**<B**
(14)	[S/(S\N) : (**C**$_*$ Patrice)] - [S\N : (souvent (boit (du jus)))] - [(X\X)/X: et] - [(S\N)\((S\N)/N)): (**B** rarement (**C**$_*$ (du lait)))]	**>dec**

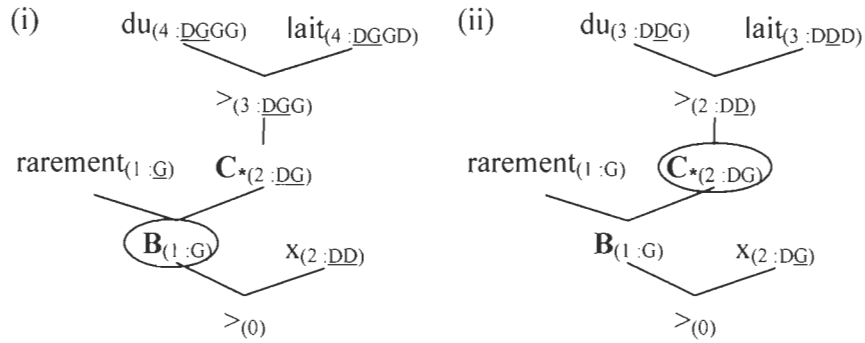
Les étapes 1 à 14 décrivent les opérations de l'analyse catégorielle combinatoire de gauche vers la droite sur l'expression applicative typée jusqu'à l'obtention d'une expression nécessitant une réorganisation structurelle. Les étapes 9 et 14 impliquent une

opération de décomposition qui, nous permettra de reconstruire l'expression en deux composantes qui pourront éventuellement être recombinées ensemble par une règle d'application et dont la deuxième permettra la poursuite de l'analyse. La première décomposition provient du fait que l'étape 5 nous donne le faux constituant *((**B** (**B** (**C*** *Patrick) boit) du) jus)*. Nous ne pouvons pas poursuivre l'analyse et nous devons par conséquent réduire des combinateurs dans l'expression jusqu'à ce qu'il soit décomposable. Suite à la réduction de deux combinateurs **B**, nous obtenons alors *(**C*** *Patrick) (boit (du jus)))*, qui permet une décomposition. À l'étape 13, nous avons encore une fois un faux constituant avec l'expression *((**C*** *Patrice) (souvent (boit (du jus))))* qui est de type S alors que tous les éléments de l'énoncé n'ont pas été considérés. Nous procédons alors à une deuxième décomposition, séparant ainsi *(**C*** *Patrice)* de *(souvent (boit (du jus)))*. Ce dernier contient le premier membre de la coordination. Nous sommes toujours bloqués et par conséquent nous devons recourir à une réorganisation autre que la décomposition, c'est-à-dire notre méthode.

L'expression combinatoire du second membre de la coordination est *(**B** rarement (**C*** *(du lait))*, pour laquelle il manque un argument au combinateur **B** que nous ajouterons et désignerons par x. L'expression devient donc *((**B** rarement (**C*** *(du lait)) x)* et est représentée par l'arbre suivant :



L'exécution de l'algorithme entraîne les étapes ci-dessous :

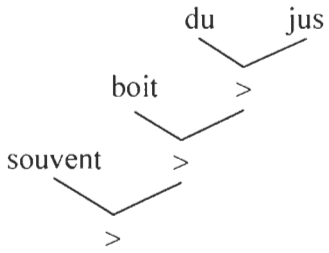


À l'étape (i), nous rencontrons le combinateur **B**. Son premier argument est *rarement* et son chemin passe de « GG » à « G ». Le second argument est la seconde branche, soit *(**C*** (du lait))*. Les directions gauche et droite du début du chemin vers tous ces nœuds seront alors inversées pour devenir droite et gauche. Enfin, on recherchera pour le troisième argument des nœuds dont le chemin débute par « D », ce qui est le cas de *x*, à qui on ajoutera une direction droite.

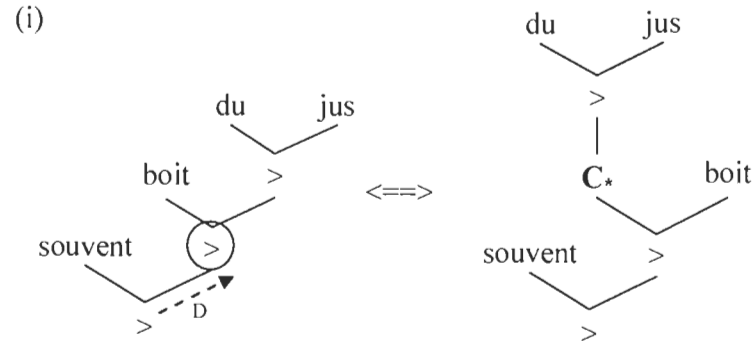
En (ii), nous atteignons le combinateur **C***, pour lequel (du lait) est facilement identifiable en tant que premier argument. Pour ces nœuds, « DGG » deviendra « DD ». Finalement, *x* sera à nouveau le dernier argument, puisqu'on parcourt l'arbre à la recherche de nœuds dont le chemin commence par « DD ». Par conséquent, son chemin deviendra « DG ».

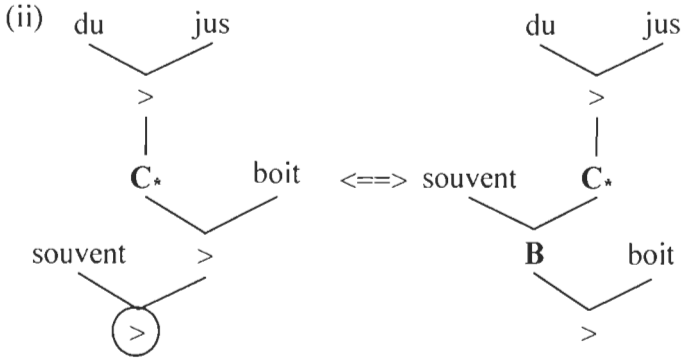
Tous les combinateurs de l'expression ont été parcourus et nous avons calculé la position des combinateurs une fois qu'ils auront été introduits. En récupérant les combinateurs et leurs chemins de droite à gauche dans l'expression combinatoire, l'ordre d'introduction obtenue est **C***(2 :DG), **B**(1 :G). Au risque de le répéter, il nous faudra introduire les combinateurs dans cet ordre dans le premier membre de la coordination, en considérant comme point d'introduction leur combinateur fonctionnel. Pour cela, nous devrons ignorer la dernière direction de leurs chemins.

Le premier membre se structure ainsi :



Les schémas qui suivent illustrent les étapes d'introductions :





À l'étape (i), en ignorant la dernière direction, le délimiteur fonctionnel se trouve à droite de la racine, d'où se fera l'introduction de **C**$_*$. Puis, en (ii), le délimiteur est la

racine, ce qui fait que (souvent ((**C**$_*$ (du jus)) boit)) est l'expression réduite et équivalente de (**B** (souvent ((**C**$_*$ (du jus))) boit).

La réorganisation structurelle est terminée et nous sommes en mesure de poursuivre l'analyse, dont les prochaines étapes se dérouleront ainsi :

(15)	[S/(S\N) : (**C**$_\bullet$ Patrice)] - [S\N : ((**B** souvent (**C**$_\bullet$ (du jus))) boit)] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**B** rarement (**C**$_\bullet$ (du lait)))]	
(16)	[S/(S\N) : (**C**$_\bullet$ Patrice)] - [S\N : boit] - [(S\N)\((S\N)/N) : (**B** souvent (**C**$_\bullet$ (du jus)))] - [(X\X)/X : et] - [(S\N)\((S\N)/N)): (**B** rarement (**C**$_\bullet$ (du lait)))]	<**dec**
(17)	[S/(S\N) : (**C**$_\bullet$ Patrice)] - [S\N : boit] - [(S\N)\((S\N)/N) : (**B** souvent (**C**$_\bullet$ (du jus)))] - [((S\N)\((S\N)/N))\ (S\N)\((S\N)/N))): (et (**B** rarement (**C**$_\bullet$ (du lait))))]	>
(18)	[S/(S\N) : (**C**$_\bullet$ Patrice)] - [S\N : boit] - [(S\N)\((S\N)/N)): ((et (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus))))]	<
(19)	[S/(S\N) : (**C**$_\bullet$ Patrice)] - [(S\N): (((et (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus)))) boit)]	<
(20)	[S: ((**C**$_\bullet$ Patrice) (((et (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus)))) boit))]	>
(21)	((**C**$_\bullet$ Patrice) (((et (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus)))) boit))	
(22)	((((et (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus)))) boit) Patrice)	(**C**$_\bullet$)
(23)	((((**Φ** ∧ (**B** rarement (**C**$_\bullet$ (du lait)))) (**B** souvent (**C**$_\bullet$ (du jus)))) boit) Patrice)	(et = **Φ** ∧)
(24)	((∧ ((**B** rarement (**C**$_\bullet$ (du lait))) boit) ((**B** souvent (**C**$_\bullet$ (du jus))) boit) Patrice)	(**Φ**)
(25)	((∧ (rarement ((**C**$_\bullet$ (du lait)) boit)) ((**B** souvent (**C**$_\bullet$ (du jus))) boit) Patrice)	(**B**)
(26)	((∧ (rarement (boit (du lait))) ((**B** souvent (**C**$_\bullet$ (du jus))) boit) Patrice)	(**C**$_\bullet$)
(27)	((∧ (rarement (boit (du lait))) (souvent ((**C**$_\bullet$ (du jus)) boit)) Patrice)	(**B**)
(28)	**((∧ (rarement (boit (du lait))) (souvent (boit (du jus)))) Patrice)**	(**C**$_\bullet$)

La réorganisation permet une décomposition arrière qui à la fois isole boit et permet les applications avant et arrière avec la conjonction de coordination *et*. Nous aboutissons à l'étape 20 au type S qui nous indique que la phrase est syntaxiquement valide.

Les étapes 21 à 28 construisent l'interprétation fonctionnelle sémantique par des opérations de β-réduction (le combinateur ayant été réduit apparait entre parenthèses

dans la colonne de droite) jusqu'à l'atteinte de la forme normale, dont l'expression est *((^ (rarement (boit (du lait))) (souvent (boit (du jus)))) Patrice)*.

4.6.2 Exemple 2 : Youkhatibou elibnou abahou bi-ihthiramin wa oumahou bi-hananin

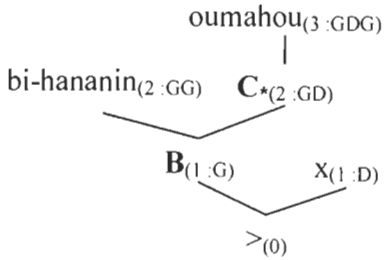
Cette phrase en arabe pourrait se traduire par : « Parle le fils à son père avec respect et à sa mère avec amour ». Encore une fois, l'application des règles sont décrites par les étapes suivantes :

(1)	[(S/N)/N : youkhatibou] - [N :elibnou]-[N : abahou] - [(S/N)\(S/N) : bi-ihthiramin] - [(X\X)/X : wa] - [N :oumahou] - [(S/N)\(S/N) : bi-hananin]	
(2)	[(S/N)/N : youkhatibou] - [S\(S/N): (**C**• elibnou)]-[N : abahou]-[(S/N)\(S/N) : bi-ihthiramin] - [(X\X)/X : wa]-[N : oumahou] - [(S/N)\(S/N) : bi-hananin]	**<T, M1**
(3)	[S/N : (**B** (**C**• elibnou) youkhatibou)] - [N : abahou]-[(S/N)\(S/N) : bi-ihthiramin]-[(X\X)/X : wa]-[N : oumahou] - [(S/N)\(S/N) : bi-hananin]	**<Bx**
(4)	[S : ((**B** (**C**• elibnou) youkhatibou) abahou)] - [(S/N)\(S/N) : bi-ihthiramin] - [(X\X)/X : wa] - [N : oumahou]-[(S/N)\(S/N) : bi-hananin]	**>**
(5)	[S/(S/N) :(**C**• elibnou)] - [S/N:(youkhatibou abahou)] - [(S/N)\(S/N) :bi-ihthiramin] - [(X\X)/X :wa] - [N:oumahou] - [(S/N)\(S/N):bi-hananin]	**>dec**
(6)	[S/(S/N) : (**C**• elibnou)] - [S/N : (bi-ihthiramin (youkhatibou abahou))] - [(X\X)/X : wa] - [N : oumahou] - [(S/N)\(S/N) : bi-hananin]	**<**
(7)	[S : ((**C**• elibnou) (bi-ihthiramin (youkhatibou abahou)))] - [(X\X)/X: wa] - [N : oumahou] - [(S/N)\(S/N) : bi-hananin]	**>**
(8)	[S : ((**C**• elibnou) (bi-ihthiramin (youkhatibou abahou)))] - [(X\X)/X : wa] - [(S/N)\((S/N)/N) : (**C**• oumahou)] - [(S/N)\(S/N) : bi-hananin]	**<T, M7**
(9)	[S : ((**C**• elibnou) (bi-ihthiramin (youkhatibou abahou)))] - [(X\X)/X : wa] - [(S/N)\((S/N)/N) : (**B** bi-hananin (**C**• oumahou))]	**<B**
(10)	[S/(S/N) : (**C**• elibnou)] - [S/N : (bi-ihthiramin (youkhatibou abahou))] - [(X\X)/X : wa] - [(S/N)\((S/N)/N) : (**B** bi-hananin (**C**• oumahou))]	**>dec**

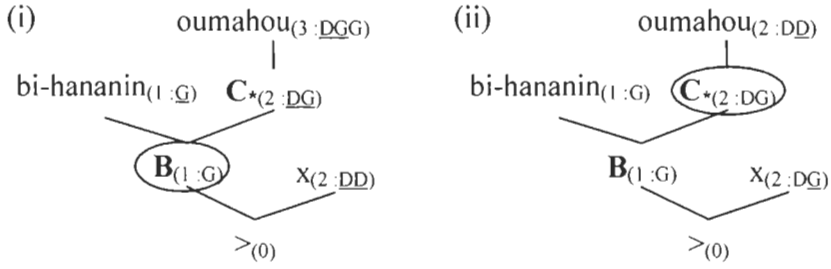
Les étapes 1 à 7 permettent de structurer la première partie de l'expression applicative, moyennant une décomposition avant lors de l'étape 5. Les étapes 8 et 9 forment le second membre incomplet de la coordination. La décomposition avant de l'étape isole le premier membre de la coordination afin de pouvoir procéder à une réorganisation structurelle.

Nous utilisons donc l'algorithme pour calculer les chemins d'introduction des combinateurs du second membre de la coordination, *(**B** bi-hananin (**C*** oumahou)*, qui seront par la suite introduits dans l'expression combinatoire du premier membre de la coordination, soit *(bi-ihthiramin (youkhatibou abahou))*.

L'arbre représentant le second membre prend la forme suivante :



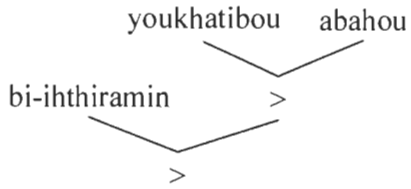
L'algorithme engendre la série de calculs sur les combinateurs tels que présentés ci-contre:



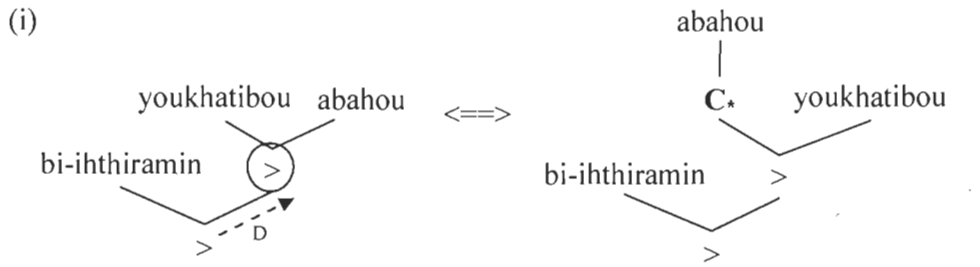
L'étape (i) calcule le chemin des arguments du combinateur **B**. Ses nœuds de gauche et de droite constituent ses deux premiers arguments. Ainsi, *bi-hananin(2 :GG)* devient *bi-hananin(1 :G)* et *(**C***(2 :GD) oumahou(3 :GDG))* deviennent *(**C***(2 :DG) oumahou(3 :DGG))*. Le dernier argument est le nœud restant, *x*, dont le chemin sera majoré d'une direction droite.

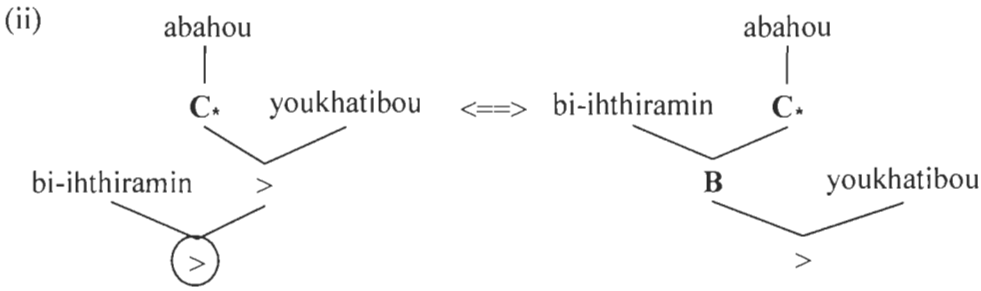
À l'étape (ii) vient le tour du combinateur C*. Le premier argument est évidemment *oumahou*, pour lequel les directions « GG » changent pour « D », puis le deuxième argument est encore une fois *x*, dont le chemin évoluera de « DD » à « DG ».

L'algorithme ayant passé en revue tous les nœuds, nous pouvons procéder à l'introduction des combinateurs dans le premier membre de la coordination, c'est-à-dire *(bi-ihthiramin (youkhatibou abahou))* dont voici la représentation :



L'ordre et les chemins d'introduction sont $\mathbf{C}_{*(2\,:\,DG)}$, $\mathbf{B}_{(1\,:G)}$ et s'illustrent comme ceci:





La première étape introduit le combinateur **C*** à partir de son délimiteur fonctionnel, situé à droite de la racine. Le chemin vers **C*** après son introduction est par conséquent celui qui a été calculé, soit (2 :DG). L'étape (ii) introduit le combinateur **B** directement à la racine de l'arbre.

Le résultat de la réorganisation de l'expression construit le membre *(**B** (bi-ihthiramin ((**C*** abahou)) youkhatibou)*. L'analyse peut donc reprendre son cours, produisant ainsi les étapes suivantes :

(11)	[S/(S/N):(**C**• elibnou)] - [S/N:((**B** bi-ihthiramin (**C**• abahou)) youkhatibou)] - [(X\X)/X : wa] - [(S/N)\((S/N)/N): (**B** bi--hananin (**C**• oumahou))]	
(12)	[S/(S/N):(**C**• elibnou)]-[(S/N)/N:youkhatibou]-[(S/N)\((S/N)/N):(**B** bi-ihthiramin (**C**• abahou))] - [(X\X)/X : wa] - [(S/N)\((S/N)/N): (**B** bi--hananin (**C**• oumahou))]	<**dec**
(13)	[S/(S/N):(**C**• elibnou)]-[(S/N)/N:youkhatibou]-[(S/N)\((S/N)/N):(**B** bi-ihthiramin (**C**• abahou))] - [(X\X)/X : wa] - [(((S/N)\((S/N)/N))\((S/N)\((S/N)/N))): (wa (**B** bi--hananin (**C**• oumahou)))]	>
(14)	[S/(S/N) : (**C**• elibnou)] - [(S/N)/N : youkhatibou] - [(S/N)\((S/N)/N) : ((wa (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou)))]	<
(15)	[S/(S/N) : (**C**• elibnou)] - [(S/N) : (((wa (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou))) youkhatibou)]	<
(16)	[S : ((**C**• elibnou) (((wa (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou))) youkhatibou))]	>
(17)	((**C**• elibnou) (((wa (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou))) youkhatibou))	
(18)	((((wa (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou))) youkhatibou) elibnou)	(**C**•)
(19)	((((**Φ** ^ (**B** bi-hananin (**C**• oumahou))) (**B** bi-ihthiramin (**C**• abahou))) youkhatibou) elibnou)	(wa = **Φ** ^)
(20)	((^ ((**B** bi-hananin (**C**• oumahou)) youkhatibou) ((**B** bi-ihthiramin (**C**• abahou)) youkhatibou))) elibnou)	(**Φ**)
(21)	((^ (bi-hananin ((**C**• oumahou) youkhatibou)) ((**B** bi-ihthiramin (**C**• abahou)) youkhatibou))) elibnou)	(**B**)
(22)	((^ (bi-hananin (youkhatibou oumahou)) ((**B** bi-ihthiramin (**C**• abahou)) youkhatibou))) elibnou)	(**C**•)
(23)	((^ (bi-hananin (youkhatibou oumahou)) ((bi-ihthiramin ((**C**• abahou) youkhatibou))) elibnou)	(**B**)
(24)	((^ (**bi-ihthiramin (youkhatibou abahou)) (bi-hananin (youkhatibou oumahou))) elibnou**)	(**C**•)

Par l'obtention de S, les étapes 11 à 16 mènent à la validation syntaxique de l'expression applicative typée. Finalement, les étapes qui suivent réduisent l'expression

combinatoire jusqu'à l'obtention de l'interprétation fonctionnelle sémantique, qui est *((^ (bi-ihthiramin (youkhatibou abahou)) (bi-hananin (youkhatibou oumahou))) elibnou)*.

4.7 Complexité algorithmique

Si nous avons démontré le bon fonctionnement de l'algorithme, il serait incomplet de ne pas discuter de la complexité algorithmique. D'entrée de jeu, dans le scénario du pire cas, nous aurions un arbre binaire complet (dans lequel toutes les feuilles sont à la même distance de la racine et tous les deux ont deux enfants; cela exclurait la présence d'un combinateur unaire comme **C***), pour lequel la méthode *calculerChemin* serait parcourue $1 + 2^{n-1}!$ fois, *n* représentant le niveau de profondeur de l'arbre, en incluant sa racine. Sans se perdre dans les autres coûts computationnels engendrés, entre autres, par la recherche des chemins à modifier pour les derniers arguments des combinateurs **B** et **C***, nous pouvons d'hors et déjà constater que nous sommes en présence d'une complexité O(n!) dite factorielle, ce qui normalement ne constitue pas une complexité très souhaitable.

Toutefois il ne faut pas perdre de vue que dans le contexte qui nous intéresse, les expressions combinatoires représentées par les arbres binaires auront rarement plus que quelques niveaux de profondeurs. Or, une complexité O(n!) devient problématique lorsque la valeur de n est très élevée, ce qui n'est pas notre cas. En effet, l'expression est celle d'un membre d'une coordination qui, dans sa structure linguistique, sera plutôt limitée dans le nombre d'unités lexicales qu'elle peut comporter. Même en supposant que nous pourrions rencontrer un membre de coordination plutôt long (disons un membre composé de 16 unités linguistiques), il n'y aurait que 31 exécutions de la méthode *calculerChemins* (plus un parcours supplémentaire par combinateur **C*** présent dans l'expression combinatoire). En comparaison, l'exemple 3 de la section 4.5.3 est déjà relativement complexe et seulement 14 exécutions de la méthode sont nécessaires. Certes, nous ne pouvons pas affirmer avec exactitude le nombre d'unités linguistiques que nous retrouvons en moyenne dans le membre d'une coordination, qui dépend d'une

multitude de facteurs, mais nous croyons raisonnable de considérer que l'algorithme ne cause pas un problème majeur en ce qui concerne le coût computationnel.

De plus, malgré la nature également récursive de la recherche des derniers arguments des combinateurs, nous avons vu plus tôt que la zone de fouille s'avère être très réduite, puisque nous n'avons pas à sonder les noeuds enfants des combinateurs, ni les noeuds déjà parcourus. Cela limite donc de façon non négligeable le nombre d'opérations nécessaires à l'exécution de l'algorithme.

4.8 Discussion

Au cours de ce chapitre, nous avons comment nous pouvions exploiter la mémoire véhiculée par les combinateurs **B** et **C*** afin de retracer leurs positions d'introduction dans l'expression. Ces observations nous ont permis d'élaborer une stratégie pour surmonter la problématique des faux constituants dans certains cas de formes elliptiques de coordinations. Ainsi, grâce à cet algorithme, nous pouvons remanier la structure de l'expression combinatoire d'un membre d'une coordination afin que, si possible elle possède une structure similaire à celle du second membre, ce qui permettra la poursuite de l'analyse. Nous croyons que les exemples présentés auront permis d'en rendre compte.

Bien entendu, l'algorithme ne couvre pas l'ensemble des combinateurs existant. Un autre aspect fort intéressant de l'algorithme est que si nous désirons ajouter d'autres combinateurs, nous n'aurons qu'à étudier la structure des expressions combinatoires de la β-réduction du combinateur, comme nous l'avons fait pour **C*** et **B**, sans avoir à considérer sa structure lorsqu'il est aux côtés des autres combinateurs, puisque chaque combinateur est traité indépendamment dans l'algorithme.

Enfin, nous avons voulu démontrer que, malgré la nature récursive de l'algorithme, le coût computationnel de son exécution est toutefois limité par les structures syntaxiques

du constituant à reconstruire. Dans tous les cas, le plus important est que nous avons dorénavant un moyen d'automatiser complètement le processus d'introduction de combinateurs pour la restructuration des expressions combinatoires.

CHAPITRE 5

IMPLÉMENTATION

À l'intérieur de ce chapitre, nous présenterons un prototype qui implémente la méthode décrite au chapitre précédent dans le but d'en prouver empiriquement la robustesse. L'application permet donc à un utilisateur d'entrer un jeu d'expressions combinatoires et de tester le processus de réorganisation structurelle.

5.1 Présentation du prototype

En premier lieu, nous préférerions avertir le lecteur que le prototype n'a pas pour objectif premier de construire un logiciel hautement performant, ni de calculer le coût computationnel de l'algorithme, mais plutôt d'éprouver la méthode par des tests. Aussi, il n'implémente pas le processus d'analyse complet de la Grammaire Catégorielle Combinatoire Applicative : il se limite à tester notre travail.

Cette section présentera les éléments les plus importants de la conception du prototype. Nous ferons aussi une démonstration de l'application en exécution, à la manière d'un guide utilisateur.

5.1.1 Langage de programmation

Le prototype a été implémenté avec le langage de programmation orienté objet *C#*[17]. Nous désirions simplement utiliser un langage actuel qui permet de produire efficacement une application dotée d'interfaces graphiques utilisateurs. D'autres langages auraient évidemment pu être considérés.

[17] La version spécifique qui a été utilisée pour construire le prototype est Microsoft Visual C# 2008. Nous n'entrerons pas dans les détails de ce langage. De nombreux ouvrages spécialisés existent sont disponibles et sauront mieux renseigner le lecteur désirant en apprendre plus sur le sujet.

5.1.2 Structures de données

Nous ne discuterons ici que des structures de données principales qui sont utilisées pour représenter les éléments en lien direct avec la méthode et non celles qui composent les interfaces utilisateurs et produisent les sorties.

Une expression combinatoire est structurée sous la forme d'un arbre composé de n noeuds. Chaque nœud qui le compose est soit une application, soit un combinateur, soit une unité lexicale. Selon le type de noeud, ce dernier contiendra à son tour un nombre fixe de nœuds organisés en tableau : deux sous-nœuds dans le cas d'une application ou du combinateur **B**, un seul dans le cas du combinateur $\mathbf{C}_*$ et aucun s'il s'agit d'une unité lexicale. Dans le langage C#, l'accès à une cellule d'un tableau se fait en spécifiant un indice numérique entre crochets. Ainsi, nous accédons à la première valeur du tableau *t* par la commande *t[0]* et la valeur suivante par *t[1]*.

Dans l'objectif de calculer les chemins d'introduction, un nœud contient un chemin. Ce dernier est implémenté en une concaténation de caractères *0* ou *1*, l'équivalent des directions *gauche* et *droite* de la méthode présentée. Cette tactique permet de se rendre au point d'introduction directement en utilisant les directions *0* et *1* en tant qu'indices dans les tableaux des sous-nœuds.

5.1.3 Modules de l'application

Nous pouvons diviser le prototype en deux modules bien distincts :

(i) Le module de la Grammaire Catégorielle Combinatoire Applicative;

(ii) Le module regroupant les interfaces utilisateurs.

Le premier module contient les structures de données pour les expressions combinatoires et pour les combinateurs, ainsi que l'ensemble des règles de β-réduction (autant pour l'introduction que l'élimination d'un combinateur). De plus, il contient des méthodes de manipulation qui permettent de calculer les chemins d'introduction des combinateurs d'une expression combinatoire (c'est-à-dire l'algorithme présenté au chapitre 4) et d'introduire ces combinateurs dans une autre expression. Par ailleurs, l'arbre de l'expression combinatoire permet de convertir sa structure en une expression combinatoire textuelle. Ce module se compile indépendamment du deuxième module sous la forme d'une bibliothèque de ressources disponibles pour d'autres applications.

Le deuxième module constitue la couche de présentation graphique des données. Il permet à l'utilisateur de créer son jeu de tests en construisant des expressions combinatoires et de tester la méthode de réorganisation structurelle, en utilisant les fonctions et les entités contenues dans le premier module.

5.1.4 Démonstration

Nous présenterons dans cette section quelques captures d'écran du prototype en action. Nous illustrerons les différentes interfaces utilisateurs disponibles et nous en ferons bien entendu la description.

5.1.4.1 Préparation du jeu de tests

Le lancement de l'exécutif affichera à l'écran la fenêtre suivante :

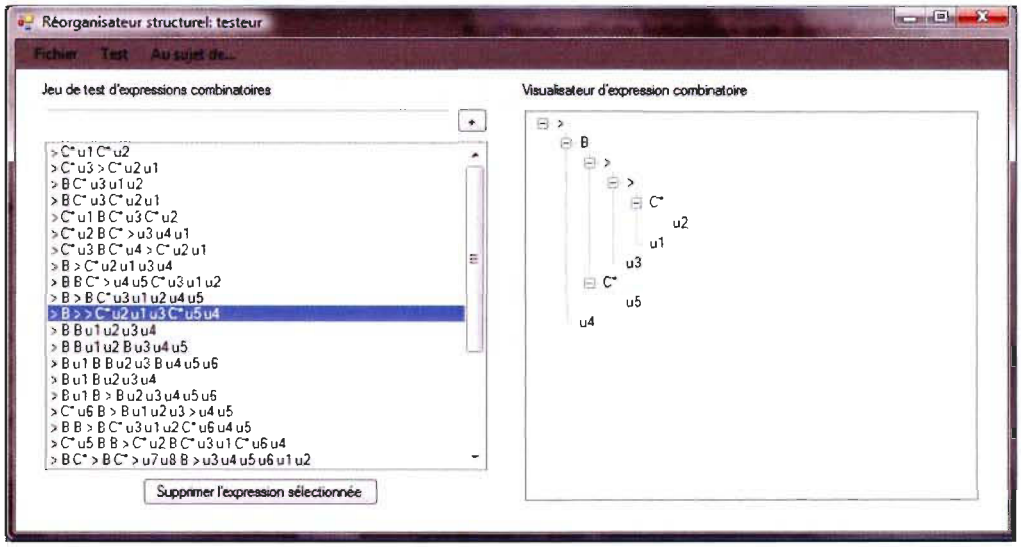


Figure 2 Interface de préparation du jeu de tests

L'utilisateur a la possibilité d'ajouter des expressions combinatoires à un jeu de tests, en écrivant dans la zone de texte au-dessus de la liste, puis en appuyant sur le bouton « + ». Nous vérifions si l'expression combinatoire est valide avant l'ajout. Dans le cas négatif, l'expression ne sera pas considérer et l'utilisateur en sera averti. À l'inverse, il peut supprimer une expression du jeu en la sélectionnant d'abord et en cliquant ensuite sur le bouton « Supprimer l'expression sélectionnée ». Aussi, sélectionner une expression affiche la représentation en arbre de l'expression combinatoire. Enfin, nous offrons la possibilité de sauvegarder ou de charger un jeu de test, à partir du menu « Fichier ».

5.1.4.2 Lancement du test

Une fois le jeu de tests défini, nous sommes fin prêts à démarrer la séquence de tests. Comme illustré ci-contre, le prototype permet au choix de lancer un test unique pour l'expression combinatoire sélectionnée dans la liste ou d'effectuer des tests pour chacune des expressions.

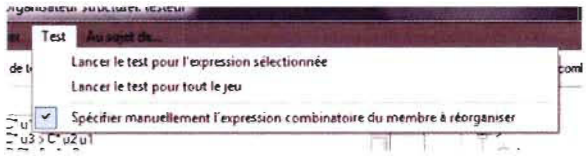


Figure 3 Options des tests

De plus, une option supplémentaire détermine si l'expression combinatoire du membre à réorganiser est spécifiée manuellement ou non. Par défaut, cette option est désactivée et les combinateurs seront alors introduits dans une expression combinatoire qui est la forme normale de celle du test courant. Toutefois, si l'utilisateur coche cette option, il pourra spécifier pour chaque test une expression combinatoire de son choix. Chacun des cas permet de tester un aspect différent de l'algorithme. Nous y reviendrons à la prochaine section. Quoiqu'il en soit, le lancement de test fait apparaître une nouvelle fenêtre telle que présentée ci-dessous :

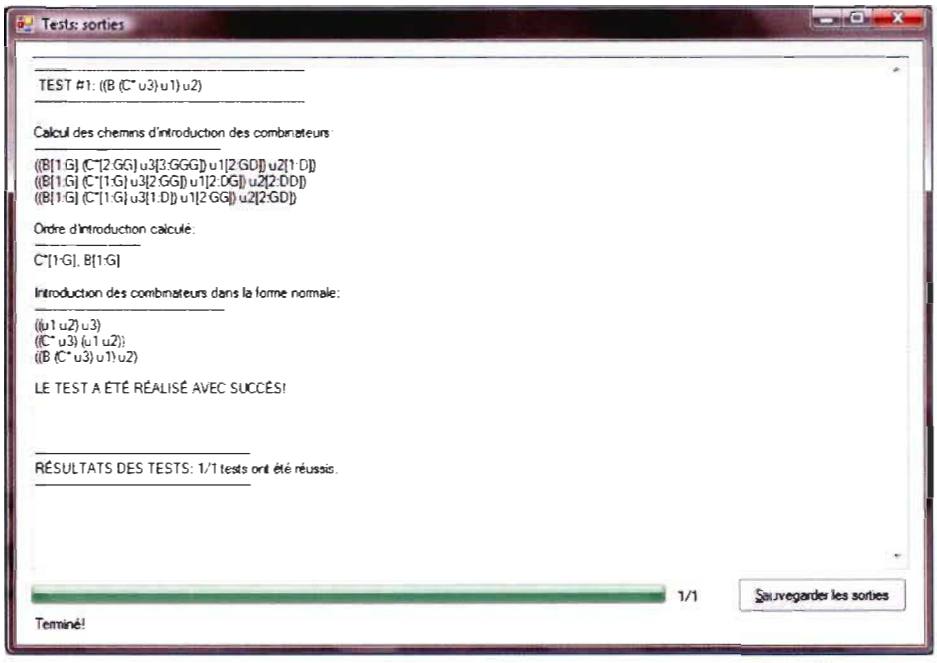


Figure 4 Sortie des tests

Cette interface est chargée d'afficher en sortie les détails de chacune des étapes de chaque test, d'une façon similaire à ce que nous avons présenté lors des exemples d'application de l'algorithme au chapitre précédent. Si l'introduction d'un combinateur dans l'expression s'avère être impossible, le test s'interrompt et affiche une erreur en sortie. Si tous les combinateurs ont pu être introduits sans problème, alors le test est un succès. Une barre de progression donne l'indication de l'avancée des tests. Nous affichons à la toute fin une statistique par rapport au nombre de succès sur la quantité de tests effectués. La fin de la procédure rend disponible la sauvegarde des sorties dans un fichier texte standard.

5.2 Tests

Afin de mettre à l'épreuve vigoureusement l'algorithme, nous avons construit un jeu de 30 expressions combinatoires visant à couvrir l'ensemble des structures types qui peuvent découler de l'introduction de combinateurs **B** et $\mathbf{C}_*$.

Nous soumettons donc au prototype ce jeu afin de strictement vérifier si l'algorithme permet dans tous les cas de calculer les bons chemins d'introduction des combinateurs dans l'expression. C'est pourquoi pour cette exécution les formes normales dans lesquelles sont introduits les combinateurs sont celles des expressions elles-mêmes et non des formes normales déterminées manuellement.

Voici la liste des expressions combinatoires testées :

1. $((\mathbf{C}_* u_1) (\mathbf{C}_* u_2))$
2. $((\mathbf{C}_* u_3) ((\mathbf{C}_* u_2) u_1)$
3. $((\mathbf{B} (C_* u_3) u_1) u_2)$
4. $((\mathbf{B} (C_* u_3) (\mathbf{C}_* u_2)) u_1)$
5. $((\mathbf{C}_* u_1) (\mathbf{B} (\mathbf{C}_* u_3) (\mathbf{C}_* u_2)))$
6. $((\mathbf{C}_* u_2) (\mathbf{B} (\mathbf{C}_* (u_3 u_4)) u_1))$

7. ((**C*** u_3) (**B** (**C*** u_4) ((**C*** u_2) u_1)))
8. ((**B** ((**C*** u_2) u_1) u_3) u_4)
9. ((**B** (**B** (**C*** (u_4 u_5)) (**C*** u_3)) u_1) u_2)
10. ((**B** ((**B** (**C*** u_3) u_1) u_2) u_4) u_5)
11. ((**B** (((**C*** u_2) u_1) u_3) (**C*** u_5)) u_4)
12. ((**B** (**B** u_1 u_2) u_3) u_4)
13. ((**B** (**B** u_1 u_2) (**B** u_3 u_4)) u_5)
14. ((**B** u_1 (**B** (**B** u_2 u_3) (**B** u_4 u_5))) u_6)
15. ((**B** u_1 (**B** u_2 u_3)) u_4)
16. ((**B** u_1 (**B** ((**B** u_2 u_3) u_4) u_5)) u_6)
17. ((**C*** u_6) (**B** ((**B** u_1 u_2) u_3) (u_4 u_5)))
18. ((**B** (**B** ((**B** (**C*** u_3) u_1) u_2) (**C*** u_6)) u_4) u_5)
19. ((**C*** u_5) (**B** (**B** ((**C*** u_2) (**B** (**C*** u_3) u_1)) (**C*** u_6)) u_4))
20. ((**B** (**C*** ((**B** (**C*** (u_7 u_8)) (**B** (u_3 u_4) u_5)) u_6)) u_1) u_2)
21. ((**C*** ((**C*** u_6) u_5)) (**B** ((**B** u_1 u_2) u_3) u_4))
22. ((**B** (**B** (**B** u_1 u_2) u_3) (**B** (**C*** u_6) u_4)) u_5)
23. ((**B** u_1 (**B** u_2 (**C*** ((**B** (**C*** u_6) (**C*** u_5))))) u_4) u_3)
24. ((**B** (**C*** ((**B** (**B** (u_4 u_5) u_6) u_7) u_8)) (**B** (**C*** u_3) u_1)) u_2)
25. ((**B** (**B** (**C*** ((**B** (**C*** ((**C*** u_8) (**B** u_6 u_7))) (**C*** u_5)) u_4)) (**C*** u_3)) (**C*** u_2)) u_1)
26. ((**B** (**C*** ((**B** (**B** u_5 u_6) u_7) u_8)) (**B** (**B** u_1 u_2) u_3)) u_4)
27. ((**B** (**B** (**B** (**B** (**B** (**B** u_1 u_2) u_3) u_4) u_5) u_6) (**C*** u_9)) u_7) u_8)
28. ((**B** u_1 (**C*** u_3)) u_2)
29. ((**C*** ((**C*** u_4) u_3)) (**B** u_1 u_2))
30. ((**B** (**B** (**C*** ((**B** (**C*** u_6) u_4) u_5)) (**C*** u_3)) u_1) u_2)

Les trois dernières expressions de la banque de données sont celles que nous avions présentées lors des exemples d'applications de l'algorithme du chapitre 4.

5.2.1 Résultats

Afin de ne pas encombrer à outrance cette section, nous invitons le lecteur a bien vouloir consulter l'annexe 1 où il retrouvera l'intégralité des sorties obtenus pour chacun des tests.

Toutefois, nous listerons et décrirons brièvement tout de même les sorties de trois d'entre elles, soient les expressions combinatoires 2, 4 et 13.

((C* u_3) ((C* u_2) u_1)

Calcul des chemins d'introduction des combinateurs:
((C[1:G] u3[2:GG]) ((C*[2:DG] u2[3:DGG]) u1[2:DD]))*
((C[1:G] u3[1:D]) ((C*[2:GG] u2[3:GGG]) u1[2:GD]))*
((C[1:G] u3[1:D]) ((C*[2:GG] u2[2:GD]) u1[2:GG]))*

Ordre d'introduction calculé:
C[2:GG], C*[1:G]*

Introduction des combinateurs dans la forme normale:
((u1 u2) u3)
(((C u2) u1) u3)*
((C u3) ((C* u2) u1))*

Le test a été réalisé avec succès!

Ce cas illustre une situation plutôt simple ne comportant que des changements de type. L'algorithme parcourt l'expression de gauche à droite et nous rencontrerons d'abord $\mathbf{C}_{*(1:G)}$. L'unité lexicale u_3, le premier argument de **C***, voit son chemin passer de « GG » à « D ». Le reste de l'expression compose le second argument, ce qui signifie qu'avant l'introduction de **C***, cette partie était à gauche de u_3. Le second combinateur a

pour premier argument u_2 et pour dernier argument u_1. Les chemins sont modifiés en conséquence.

Nous pouvons récupérer les chemins calculés pour chaque combinateur et procéder à leurs introductions dans la forme normale. L'expression construite est la même que nous avions au départ.

((B (C_* u_3) (C_* u_2)) u_1)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (C[2:GG] u3[3:GGG]) (C*[2:GD] u2[3:GDG])) u1[1:D])*
((B[1:G] (C[1:G] u3[2:GG]) (C*[2:DG] u2[3:DGG])) u1[2:DD])*
((B[1:G] (C[1:G] u3[1:D]) (C*[2:GG] u2[3:GGG])) u1[2:GD])*
((B[1:G] (C[1:G] u3[1:D]) (C*[2:GG] u2[2:GD])) u1[2:GG])*

Ordre d'introduction calculé:
C[2:GG], C*[1:G], B[1:G]*

Introduction des combinateurs dans la forme normale:
((u1 u2) u3)
(((C u2) u1) u3)*
((C u3) ((C* u2) u1))*
((B (C u3) (C* u2)) u1)*

Le test a été réalisé avec succès!

Cette expression combinatoire comporte trois combinateurs. Le premier combinateur visité est **B**. Son premier argument est *(**C**$_*$ u_3)*, son deuxième *(**C**$_*$ u_2)* et son troisième u_1. À l'étape suivante vient le tour du combinateur **C**$_*$, dont le premier argument est u_3. Pour ce qui est du dernier argument, l'algorithme recherche parmi les arguments à droite

du combinateur ceux dont le chemin débute par « D ». Cela correspond à *(**C*** u_2*)* et u_1. Enfin, les arguments du second combinateur **C*** sont dans l'ordre u_2 et u_1.

Encore une fois, l'introduction des combinateurs avec les chemins calculés nous mène à l'expression combinatoire originale.

((B (B u_1 u_2) (B u_3 u_4)) u_5)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (B[2:GG] u1[3:GGG] u2[3:GGD]) (B[2:GD] u3[3:GDG] u4[3:GDD])) u5[1:D])
((B[1:G] (B[1:G] u1[2:GG] u2[2:GD]) (B[2:DG] u3[3:DGG] u4[3:DGD])) u5[2:DD])
((B[1:G] (B[1:G] u1[1:G] u2[2:DG]) (B[3:DDG] u3[4:DDGG] u4[4:DDGD])) u5[3:DDD])
((B[1:G] (B[1:G] u1[1:G] u2[2:DG]) (B[3:DDG] u3[3:DDG] u4[4:DDDG])) u5[4:DDDD])

Ordre d'introduction calculé:
B[3:DDG], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:
(u1 (u2 (u3 (u4 u5))))
(u1 (u2 ((B u3 u4) u5)))
((B u1 u2) ((B u3 u4) u5))
((B (B u1 u2) (B u3 u4)) u5)

Le test a été réalisé avec succès!

Dans cette expression, nous n'avons que des combinateurs **B**. Les trois arguments du combinateur **B** de gauche sont *(**B** u_1 u_2)*, *(**B** u_3 u_4)*, ainsi que u_5. Ceux du second combinateur sont d'abord les unités lexicales u_1 et u_2, puis *(**B** u_3 u_4)* et u_5 pour le dernier, puisque nous recherchons des chemins débutant par la direction droite. Finalement, le troisième et dernier combinateur **B** modifie les chemins de ses trois arguments qui correspondent respectivement aux unités lexicales u_3, u_4 et u_5.

De nouveau, introduire les combinateurs dans la forme normale aux positions calculées donnera comme résultat l'expression combinatoire du début.

Dans son entier, les résultats des trente tests indiquent que la méthode a fonctionné pour 100% des expressions combinatoires soumises.

5.2.2 Discussion

Les résultats obtenus font la démonstration empirique de la fiabilité de l'algorithme pour calculer les positions où doivent être introduits les combinateurs d'une expression combinatoire E_1 afin de construire une expression équivalente E_2 à partir de sa forme normale.

Bien entendu, les tests d'introduction utilisent volontairement la forme normale même de l'expression combinatoire. Cela nous ramène à la discussion précédente à la section 4.4 sur le théorème de Church-Rosser. Dans un contexte d'analyse, nous ne sommes pas assurés que la phrase est bien construite syntaxiquement avant d'avoir pu atteindre le type S. Il pourrait donc arriver que les membres construits soient syntaxiquement invalide et ne possèdent pas la même forme normale, ce qui aurait pour conséquence de faire échouer le processus de réorganisation structurelle. Par exemple, si nous lançons un test pour l'expression combinatoire 4, soit *((**B** (**C*** u_3) (**C*** u_2)) u_1)*, en demandant de spécifier manuellement la forme normale dans laquelle il faudra introduire les

combinateurs et que nous donnons la forme normale $(u_1\ (u_2\ u_3))$[18] en entrée, nous obtenons le résultat suivant en sortie :

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (C[2:GG] u3[3:GGG]) (C*[2:GD] u2[3:GDG])) u1[1:D])*
((B[1:G] (C[1:G] u3[2:GG]) (C*[2:DG] u2[3:DGG])) u1[2:DD])*
((B[1:G] (C[1:G] u3[1:D]) (C*[2:GG] u2[3:GGG])) u1[2:GD])*
((B[1:G] (C[1:G] u3[1:D]) (C*[2:GG] u2[2:GD])) u1[2:GG])*

Ordre d'introduction calculé:
C[2:GG], C*[1:G], B[1:G]*

Introduction des combinateurs dans la forme normale:
(u1 (u2 u3))
Échec: Introduction impossible d'un combinateur

Le test a échoué!

Comme nous pouvons le constater, nous avons un problème dès l'introduction du premier combinateur. Le délimiteur fonctionnel de $\mathbf{C}_*$ est sensé être le premier argument de l'application « racine » de l'expression. Hors cet argument est l'unité lexicale u_1, et non une structure applicative permettant l'introduction du combinateur. Cela conduit inévitablement à un échec et le processus d'introduction est avorté. Cette situation est normale et doit se produire, puisque la forme normale de l'expression $((\mathbf{B}\ (\mathbf{C}_*\ u_3)\ (\mathbf{C}_*\ u_2))\ u_1)$ est différente de la forme normale $(u_1\ (u_2\ u_3))$.

[18] La forme normale de l'expression combinatoire 4 étant plutôt $((u_1\ u_2)\ u_3)$.

Enfin, tous les tests effectués se sont avérés concluants. En effet, les réorganisations structurelles de tous les bons exemples ont fonctionné, tandis que celles des mauvais exemples ont toutes échouées. Cela correspond exactement à ce que nous attendions.

CHAPITRE 6

CONCLUSION

Nous avons pu voir à travers ce mémoire que les Grammaires Catégorielles permettent par une analyse catégoriel de valider la connexion syntaxique entre les différents éléments d'une phrase.

Plus spécifiquement, avec la Grammaire Catégorielle Combinatoire Applicative, nous avons démontré comment l'association canonique entre les règles catégorielles combinatoires de Steedman et les combinateurs de la logique combinatoire de Curry permet de vérifier la syntaxe de l'énoncé tout en construisant sa structure applicative. Par la réduction des combinateurs de l'expression applicative obtenue, nous arrivons alors à obtenir l'interprétation sémantique fonctionnelle.

Toutefois l'analyse syntaxique soulève plusieurs défis. Un d'entre eux concerne la formation de faux constituants, c'est-à-dire qu'une expression applicative de type S est construite sans que l'ensemble de l'énoncé n'ait été considéré. Comme solution à ce problème, Steedman propose des règles de décomposition, basées sur le principe de la neutralité paramétrique.

Cependant, la décomposition est insuffisante lorsque nous sommes en présence de certains cas elliptiques de la coordination pour lesquels la méthode ne parvient pas à construire des structures équivalentes pour chacun des membres de la coordination et ainsi permettre une concaténation.

Pour résoudre ce problème, nos travaux proposent donc une méthode qui consiste à réorganiser automatiquement le membre d'une coordination afin qu'il possède une structure équivalente à celle du second membre. Cette réorganisation conserve le contenu sémantique de l'expression originale.

L'algorithme que nous avons mis au point permet de calculer comment des combinateurs ont été introduits dans une expression donnée. Grâce à lui, nous pouvons donc calculer les chemins d'introduction des combinateurs du second membre de la coordination et s'en servir pour restructurer le premier membre. Si un des combinateurs ne peut être introduit à la position calculée, c'est que l'énoncé n'est pas syntaxiquement bien construit et par conséquent l'analyse échoue. Cette méthode améliore donc le modèle de la Grammaire Catégorielle Combinatoire Applicative en le dotant d'un procédé automatique plutôt que de règles d'équivalences statiques afin de réaliser une réorganisation structurelle.

Malgré le fait que l'algorithme soit récursif, il engendre un coût computationnel que nous jugeons raisonnable dans le contexte pour lequel nous l'utilisons. En effet, les contraintes syntaxiques de la langue limitent la dimension de l'expression combinatoire du membre d'une coordination.

Nous avons fait la démonstration du processus pour plusieurs exemples. Deux d'entre eux ont présenté l'analyse complète d'un énoncé, dont un en arabe visant à montrer que la méthode fonctionne également avec d'autres langues.

Une implémentation de l'algorithme a été effectuée afin de valider empiriquement notre méthode. Nous avons soumis 30 expressions combinatoires au prototype. Dans tous les cas, les chemins d'introduction des combinateurs calculés ont permis de reconstruire une expression équivalente.

Une des perspectives de nos travaux consistera à ce que l'algorithme puisse considérer d'autres combinateurs, comme le combinateur de substitution fonctionnelle (**S**), par exemple. Comme nous l'avons mentionné, chaque combinateur est traité de façon indépendante dans l'algorithme. Ainsi, il suffira d'analyser l'expression de β-réduction des nouveaux combinateurs et d'établir les règles de calcul qui leurs sont spécifiques.

Un autre objectif à court terme sera de greffer cette méthode à un analyseur pour la Grammaire Catégorielle Combinatoire Applicative.

Par ailleurs, nous croyons que la méthode que nous avons développée pourrait servir à prouver que deux énoncés sont en fait des paraphrases.

Enfin, les travaux de recherche présentés dans le cadre du présent mémoire ont fait l'objet de plusieurs publications à des conférences internationales. Le lecteur trouvera en annexe 2 les articles publiés dans le cadre des 21^{e} et 22^{e} conférences internationales de la *Florida Artificial Intelligence Research Society*. Aussi, une troisième publication sera disponible sous peu dans un ouvrage des Presses Universitaires du Québec. Une bourse du Fond québécois de recherche sur la nature et les technologies m'a été octroyée et me permettra de donner suite à ces travaux au cours de mes études doctorales.

ANNEXE 1

RÉSULTAT DES TESTS

Cette annexe présente les résultats des tests exécutés avec le prototype implémentant l'algorithme de calcul d'introduction de combinateurs qui a été discuté dans ce mémoire. Nous avons fourni à notre prototype un jeu de 30 expressions combinatoires. Voici les sorties qui ont été produites en sorties pour chacune d'entre elles :

1. ((C* u_1) (C* u_2))

Nous avons l'expression (u_1 u_2). Nous désirons trouver une procédure qui nous permet de passer de (u_1 u_2) à l'expression ((**C*** u_1) (**C*** u_2)).

L'algorithme calculera d'abord comment les combinateurs de l'expression ((**C*** u_1) (**C*** u_2)) ont été introduits à partir de sa forme normale.

Calcul des chemins d'introduction des combinateurs:
((C[1:G] u1[2:GG]) (C*[1:D] u2[2:DG]))*
((C[1:G] u1[1:D]) (C*[1:G] u2[2:GG]))*
((C[1:G] u1[1:D]) (C*[1:G] u2[1:D]))*

Les combinateurs sont alors introduits dans (u_1 u_2) en fonction des chemins calculés et dans l'ordre inverse de leurs réductions, c'est-à-dire de droite vers la gauche.

Ordre d'introduction calculé:
C[1:G], C*[1:G]*

Introduction des combinateurs dans la forme normale:
(u1 u2)
((C u2) u1)*
((C u1) (C* u2))*

Nous obtenons finalement la structure souhaitée.

2. ((C* u_3) ((C* u_2) u_1)

Soit l'expression ((u_1 u_2) u_3). Nous nous intéressons à connaître les étapes nous permettant de parvenir à l'expression ((C* u_3) ((C* u_2) u_1).

Pour débuter, nous calculons le chemin pour se rendre jusqu'à chaque combinateur après son introduction dans l'expression ((C* u1) (C* u2)).

Calcul des chemins d'introduction des combinateurs:
((C[1:G] u3[2:GG]) ((C*[2:DG] u2[3:DGG]) u1[2:DD]))*
((C[1:G] u3[1:D]) ((C*[2:GG] u2[3:GGG]) u1[2:GD]))*
((C[1:G] u3[1:D]) ((C*[2:GG] u2[2:GD]) u1[2:GG]))*

Les combinateurs sont ensuite introduits dans l'expression ((u_1 u_2) u_3) selon l'ordre et les chemins calculés :

Ordre d'introduction calculé:
C[2:GG], C*[1:G]*

Introduction des combinateurs dans la forme normale:
((u1 u2) u3)
(((C u2) u1) u3)*
((C u3) ((C* u2) u1))*

L'expression obtenue est la même qu'au départ.

3. ((B (C* u_3) u_1) u_2)

Considérons à nouveau l'expression ((u1 u2) u3). Cette fois-ci, nous désirons savoir s'il est possible d'effectuer une restructuration qui nous permettrait d'obtenir l'expression ((**B** (**C*** u_3) u_1) u_2).

La première étape consiste à utiliser la méthode de calcul des chemins d'introduction des combinateurs. Cela donne la sortie suivante :

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (C[2:GG] u3[3:GGG]) u1[2:GD]) u2[1:D])*
((B[1:G] (C[1:G] u3[2:GG]) u1[2:DG]) u2[2:DD])*
((B[1:G] (C[1:G] u3[1:D]) u1[2:GG]) u2[2:GD])*

À la seconde étape nous devons introduire dans ((u_1 u_2) u_3) les combinateurs dans l'ordre inverse qu'ils seraient réduits dans l'expression, selon les chemins qui ont été calculés. Ainsi :

Ordre d'introduction calculé:
C[1:G], B[1:G]*

Introduction des combinateurs dans la forme normale:
((u1 u2) u3)
((C u3) (u1 u2))*
((B (C u3) u1) u2)*

Nous obtenons une fois de plus une structure équivalente à celle qui est recherchée.

4. ((B (C* u_3) (C* u_2)) u_1)

Cet exemple et les suivants suivent les mêmes procédures que les exemples 1 à 3. Nous n'afficherons donc à partir de ce point que les sorties produites par le prototype.

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (C*[2:GG] u3[3:GGG]) (C*[2:GD] u2[3:GDG])) u1[1:D])
((B[1:G] (C*[1:G] u3[2:GG]) (C*[2:DG] u2[3:DGG])) u1[2:DD])
((B[1:G] (C*[1:G] u3[1:D]) (C*[2:GG] u2[3:GGG])) u1[2:GD])
((B[1:G] (C*[1:G] u3[1:D]) (C*[2:GG] u2[2:GD])) u1[2:GG])

Ordre d'introduction calculé:
C*[2:GG], C*[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:
((u1 u2) u3)
(((C* u2) u1) u3)
((C* u3) ((C* u2) u1))
((B (C* u3) (C* u2)) u1)

5. ((C⋆ u_1) (B (C⋆ u_3) (C⋆ u_2)))

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] u1[2:GG]) (B[1:D] (C*[2:DG] u3[3:DGG]) (C*[2:DD] u2[3:DDG])))

((C*[1:G] u1[1:D]) (B[1:G] (C*[2:GG] u3[3:GGG]) (C*[2:GD] u2[3:GDG])))

((C*[1:G] u1[1:D]) (B[1:G] (C*[1:G] u3[2:GG]) (C*[2:DG] u2[3:DGG])))

((C*[1:G] u1[1:G]) (B[1:G] (C*[1:G] u3[1:D]) (C*[2:GG] u2[3:GGG])))

((C*[1:G] u1[1:G]) (B[1:G] (C*[1:G] u3[1:D]) (C*[2:GG] u2[2:GD])))

Ordre d'introduction calculé:

C*[2:GG], C*[1:G], B[1:G], C*[1:G]

Introduction des combinateurs dans la forme normale:

((u1 u2) u3)

(((C* u2) u1) u3)

((C* u3) ((C* u2) u1))

((B (C* u3) (C* u2)) u1)

((C* u1) (B (C* u3) (C* u2)))

6. $((C_* \; u_2) \; (B \; (C_* \; (u_3 \; u_4)) \; u_1))$

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] u2[2:GG]) (B[1:D] (C*[2:DG] (u3[4:DGGG] u4[4:DGGD])) u1[2:DD]))

((C*[1:G] u2[1:D]) (B[1:G] (C*[2:GG] (u3[4:GGGG] u4[4:GGGD])) u1[2:GD]))

((C*[1:G] u2[1:D]) (B[1:G] (C*[1:G] (u3[3:GGG] u4[3:GGD])) u1[2:DG]))

((C*[1:G] u2[1:G]) (B[1:G] (C*[1:G] (u3[2:DG] u4[2:DD])) u1[2:GG]))

Ordre d'introduction calculé:

C*[1:G], B[1:G], C*[1:G]

Introduction des combinateurs dans la forme normale:

((u1 u2) (u3 u4))

((C* (u3 u4)) (u1 u2))

((B (C* (u3 u4)) u1) u2)

((C* u2) (B (C* (u3 u4)) u1))

7. ((C* u_3) (B (C* u_4) ((C* u_2) u_1)))

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] u3[2:GG]) (B[1:D] (C*[2:DG] u4[3:DGG]) ((C*[3:DDG] u2[4:DDGG]) u1[3:DDD])))

((C*[1:G] u3[1:D]) (B[1:G] (C*[2:GG] u4[3:GGG]) ((C*[3:GDG] u2[4:GDGG]) u1[3:GDD])))

((C*[1:G] u3[1:D]) (B[1:G] (C*[1:G] u4[2:GG]) ((C*[3:DGG] u2[4:DGGG]) u1[3:DGD])))

((C*[1:G] u3[1:G]) (B[1:G] (C*[1:G] u4[1:D]) ((C*[3:GGG] u2[4:GGGG]) u1[3:GGD])))

((C*[1:G] u3[1:G]) (B[1:G] (C*[1:G] u4[1:D]) ((C*[3:GGG] u2[3:GGD]) u1[3:GGG])))

Ordre d'introduction calculé:

C*[3:GGG], C*[1:G], B[1:G], C*[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) u4)

((((C* u2) u1) u3) u4)

((C* u4) (((C* u2) u1) u3))

((B (C* u4) ((C* u2) u1)) u3)

((C* u3) (B (C* u4) ((C* u2) u1)))

8. ((B ((C* u_2) u_1) u_3) u_4)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] ((C*[3:GGG] u2[4:GGGG]) u1[3:GGD]) u3[2:GD]) u4[1:D])

((B[1:G] ((C*[2:GG] u2[3:GGG]) u1[2:GD]) u3[2:DG]) u4[2:DD])

((B[1:G] ((C*[2:GG] u2[2:GD]) u1[2:GG]) u3[2:DG]) u4[2:DD])

Ordre d'introduction calculé:

C*[2:GG], B[1:G]

Introduction des combinateurs dans la forme normale:

((u1 u2) (u3 u4))

(((C* u2) u1) (u3 u4))

((B ((C* u2) u1) u3) u4)

9. ((B (B (C* (u_4 u_5)) (C* u_3)) u_1) u_2)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (B[2:GG] (C*[3:GGG] (u4[5:GGGGG] u5[5:GGGGD])) (C*[3:GGD] u3[4:GGDG])) u1[2:GD]) u2[1:D])

((B[1:G] (B[1:G] (C*[2:GG] (u4[4:GGGG] u5[4:GGGD])) (C*[2:GD] u3[3:GDG])) u1[2:DG]) u2[2:DD])

((B[1:G] (B[1:G] (C*[1:G] (u4[3:GGG] u5[3:GGD])) (C*[2:DG] u3[3:DGG])) u1[3:DDG]) u2[3:DDD])

((B[1:G] (B[1:G] (C*[1:G] (u4[2:DG] u5[2:DD])) (C*[2:GG] u3[3:GGG])) u1[3:GDG]) u2[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] (u4[2:DG] u5[2:DD])) (C*[2:GG] u3[2:GD])) u1[3:GGG]) u2[3:GGD])

Ordre d'introduction calculé:

C*[2:GG], C*[1:G], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) (u4 u5))

(((C* u3) (u1 u2)) (u4 u5))

((C* (u4 u5)) ((C* u3) (u1 u2)))

((B (C* (u4 u5)) (C* u3)) (u1 u2))

((B (B (C* (u4 u5)) (C* u3)) u1) u2)

10. ((B ((B (C* u_3) u_1) u_2) u_4) u_5)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] ((B[3:GGG] (C*[4:GGGG] u3[5:GGGGG]) u1[4:GGGD]) u2[3:GGD]) u4[2:GD]) u5[1:D])

((B[1:G] ((B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD]) u2[2:GD]) u4[2:DG]) u5[2:DD])

((B[1:G] ((B[2:GG] (C*[2:GG] u3[3:GGG]) u1[3:GDG]) u2[3:GDD]) u4[2:DG]) u5[2:DD])

((B[1:G] ((B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG]) u2[3:GGD]) u4[2:DG]) u5[2:DD])

Ordre d'introduction calculé:

C*[2:GG], B[2:GG], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) (u4 u5))

(((C* u3) (u1 u2)) (u4 u5))

(((B (C* u3) u1) u2) (u4 u5))

((B ((B (C* u3) u1) u2) u4) u5)

11. ((B (((C* u_2) u_1) u_3) (C* u_5)) u_4)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (((C*[4:GGGG] u2[5:GGGGG]) u1[4:GGGD]) u3[3:GGD]) (C*[2:GD] u5[3:GDG])) u4[1:D])

((B[1:G] (((C*[3:GGG] u2[4:GGGG]) u1[3:GGD]) u3[2:GD]) (C*[2:DG] u5[3:DGG])) u4[2:DD])

((B[1:G] (((C*[3:GGG] u2[3:GGD]) u1[3:GGG]) u3[2:GD]) (C*[2:DG] u5[3:DGG])) u4[2:DD])

((B[1:G] (((C*[3:GGG] u2[3:GGD]) u1[3:GGG]) u3[2:GD]) (C*[2:DG] u5[2:DD])) u4[2:DG])

Ordre d'introduction calculé:

C*[2:DG], C*[3:GGG], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) (u4 u5))

(((u1 u2) u3) ((C* u5) u4))

((((C* u2) u1) u3) ((C* u5) u4))

((B (((C* u2) u1) u3) (C* u5)) u4)

12. ((B (B u_1 u_2) u_3) u_4)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (B[2:GG] u1[3:GGG] u2[3:GGD]) u3[2:GD]) u4[1:D])
((B[1:G] (B[1:G] u1[2:GG] u2[2:GD]) u3[2:DG]) u4[2:DD])
((B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) u4[3:DDD])

Ordre d'introduction calculé:
B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:
(u1 (u2 (u3 u4)))
((B u1 u2) (u3 u4))
((B (B u1 u2) u3) u4)

13. $((B\ (B\ u_1\ u_2)\ (B\ u_3\ u_4))\ u_5)$

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (B[2:GG] u1[3:GGG] u2[3:GGD]) (B[2:GD] u3[3:GDG] u4[3:GDD])) u5[1:D])
((B[1:G] (B[1:G] u1[2:GG] u2[2:GD]) (B[2:DG] u3[3:DGG] u4[3:DGD])) u5[2:DD])
((B[1:G] (B[1:G] u1[1:G] u2[2:DG]) (B[3:DDG] u3[4:DDGG] u4[4:DDGD])) u5[3:DDD])
((B[1:G] (B[1:G] u1[1:G] u2[2:DG]) (B[3:DDG] u3[3:DDG] u4[4:DDDG])) u5[4:DDDD])

Ordre d'introduction calculé:
B[3:DDG], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:
(u1 (u2 (u3 (u4 u5))))
(u1 (u2 ((B u3 u4) u5)))
((B u1 u2) ((B u3 u4) u5))
((B (B u1 u2) (B u3 u4)) u5)

14. ((B u_1 (B (B u_2 u_3) (B u_4 u_5))) u_6)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] u1[2:GG] (B[2:GD] (B[3:GDG] u2[4:GDGG] u3[4:GDGD]) (B[3:GDD] u4[4:GDDG] u5[4:GDDD]))) u6[1:D])

((B[1:G] u1[1:G] (B[2:DG] (B[3:DGG] u2[4:DGGG] u3[4:DGGD]) (B[3:DGD] u4[4:DGDG] u5[4:DGDD]))) u6[2:DD])

((B[1:G] u1[1:G] (B[2:DG] (B[2:DG] u2[3:DGG] u3[3:DGD]) (B[3:DDG] u4[4:DDGG] u5[4:DDGD]))) u6[3:DDD])

((B[1:G] u1[1:G] (B[2:DG] (B[2:DG] u2[2:DG] u3[3:DDG]) (B[4:DDDG] u4[5:DDDGG] u5[5:DDDGD]))) u6[4:DDDD])

((B[1:G] u1[1:G] (B[2:DG] (B[2:DG] u2[2:DG] u3[3:DDG]) (B[4:DDDG] u4[4:DDDG] u5[5:DDDDG]))) u6[5:DDDDD])

Ordre d'introduction calculé:

B[4:DDDG], B[2:DG], B[2:DG], B[1:G]

Introduction des combinateurs dans la forme normale:

(u1 (u2 (u3 (u4 (u5 u6)))))

(u1 (u2 (u3 ((B u4 u5) u6))))

(u1 ((B u2 u3) ((B u4 u5) u6)))

(u1 ((B (B u2 u3) (B u4 u5)) u6))

((B u1 (B (B u2 u3) (B u4 u5))) u6)

15. ((B u_1 (B u_2 u_3)) u_4)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] u1[2:GG] (B[2:GD] u2[3:GDG] u3[3:GDD])) u4[1:D])
((B[1:G] u1[1:G] (B[2:DG] u2[3:DGG] u3[3:DGD])) u4[2:DD])
((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] u3[3:DDG])) u4[3:DDD])

Ordre d'introduction calculé:
B[2:DG], B[1:G]

Introduction des combinateurs dans la forme normale:
(u1 (u2 (u3 u4)))
(u1 ((B u2 u3) u4))
((B u1 (B u2 u3)) u4)

16. ((B u_1 (B ((B u_2 u_3) u_4) u_5)) u_6)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] u1[2:GG] (B[2:GD] ((B[4:GDGG] u2[5:GDGGG] u3[5:GDGGD]) u4[4:GDGD]) u5[3:GDD])) u6[1:D])

((B[1:G] u1[1:G] (B[2:DG] ((B[4:DGGG] u2[5:DGGGG] u3[5:DGGGD]) u4[4:DGGD]) u5[3:DGD])) u6[2:DD])

((B[1:G] u1[1:G] (B[2:DG] ((B[3:DGG] u2[4:DGGG] u3[4:DGGD]) u4[3:DGD]) u5[3:DDG])) u6[3:DDD])

((B[1:G] u1[1:G] (B[2:DG] ((B[3:DGG] u2[3:DGG] u3[4:DGDG]) u4[4:DGDD]) u5[3:DDG])) u6[3:DDD])

Ordre d'introduction calculé:

B[3:DGG], B[2:DG], B[1:G]

Introduction des combinateurs dans la forme normale:

(u1 ((u2 (u3 u4)) (u5 u6)))

(u1 (((B u2 u3) u4) (u5 u6)))

(u1 ((B ((B u2 u3) u4) u5) u6))

((B u1 (B ((B u2 u3) u4) u5)) u6)

17. ((C* u_6) (B ((B u_1 u_2) u_3) (u_4 u_5)))

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] u6[2:GG]) (B[1:D] ((B[3:DGG] u1[4:DGGG] u2[4:DGGD]) u3[3:DGD]) (u4[3:DDG] u5[3:DDD])))

((C*[1:G] u6[1:D]) (B[1:G] ((B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD]) (u4[3:GDG] u5[3:GDD])))

((C*[1:G] u6[1:D]) (B[1:G] ((B[2:GG] u1[3:GGG] u2[3:GGD]) u3[2:GD]) (u4[3:DGG] u5[3:DGD])))

((C*[1:G] u6[1:D]) (B[1:G] ((B[2:GG] u1[2:GG] u2[3:GDG]) u3[3:GDD]) (u4[3:DGG] u5[3:DGD])))

Ordre d'introduction calculé:

B[2:GG], B[1:G], C*[1:G]

Introduction des combinateurs dans la forme normale:

((u1 (u2 u3)) ((u4 u5) u6))

(((B u1 u2) u3) ((u4 u5) u6))

((B ((B u1 u2) u3) (u4 u5)) u6)

((C* u6) (B ((B u1 u2) u3) (u4 u5)))

18. ((B (B ((B (C* u_3) u_1) u_2) (C* u_6)) u_4) u_5)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (B[2:GG] ((B[4:GGGG] (C*[5:GGGGG] u3[6:GGGGGG]) u1[5:GGGGD]) u2[4:GGGD]) (C*[3:GGD] u6[4:GGDG])) u4[2:GD]) u5[1:D])

((B[1:G] (B[1:G] ((B[3:GGG] (C*[4:GGGG] u3[5:GGGGG]) u1[4:GGGD]) u2[3:GGD]) (C*[2:GD] u6[3:GDG])) u4[2:DG]) u5[2:DD])

((B[1:G] (B[1:G] ((B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD]) u2[2:GD]) (C*[2:DG] u6[3:DGG])) u4[3:DDG]) u5[3:DDD])

((B[1:G] (B[1:G] ((B[2:GG] (C*[2:GG] u3[3:GGG]) u1[3:GDG]) u2[3:GDD]) (C*[2:DG] u6[3:DGG])) u4[3:DDG]) u5[3:DDD])

((B[1:G] (B[1:G] ((B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG]) u2[3:GGD]) (C*[2:DG] u6[3:DGG])) u4[3:DDG]) u5[3:DDD])

((B[1:G] (B[1:G] ((B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG]) u2[3:GGD]) (C*[2:DG] u6[2:DD])) u4[3:DGG]) u5[3:DGD])

Ordre d'introduction calculé:

C*[2:DG], C*[2:GG], B[2:GG], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) ((u4 u5) u6))

(((u1 u2) u3) ((C* u6) (u4 u5)))

(((C* u3) (u1 u2)) ((C* u6) (u4 u5)))

(((B (C* u3) u1) u2) ((C* u6) (u4 u5)))

((B ((B (C* u3) u1) u2) (C* u6)) (u4 u5))

((B (B ((B (C* u3) u1) u2) (C* u6)) u4) u5)

19. $((C_* u_5)\ (B\ (B\ ((C_* u_2)\ (B\ (C_* u_3)\ u_1))\ (C_* u_6))\ u_4))$

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] u5[2:GG]) (B[1:D] (B[2:DG] ((C*[4:DGGG] u2[5:DGGGG]) (B[4:DGGD] (C*[5:DGGDG] u3[6:DGGDGG]) u1[5:DGGDD])) (C*[3:DGD] u6[4:DGDG])) u4[2:DD]))

((C*[1:G] u5[1:D]) (B[1:G] (B[2:GG] ((C*[4:GGGG] u2[5:GGGGG]) (B[4:GGGD] (C*[5:GGGDG] u3[6:GGGDGG]) u1[5:GGGDD])) (C*[3:GGD] u6[4:GGDG])) u4[2:GD]))

((C*[1:G] u5[1:D]) (B[1:G] (B[1:G] ((C*[3:GGG] u2[4:GGGG]) (B[3:GGD] (C*[4:GGDG] u3[5:GGDGG]) u1[4:GGDD])) (C*[2:GD] u6[3:GDG])) u4[2:DG]))

((C*[1:G] u5[2:DD]) (B[1:G] (B[1:G] ((C*[2:GG] u2[3:GGG]) (B[2:GD] (C*[3:GDG] u3[4:GDGG]) u1[3:GDD])) (C*[2:DG] u6[3:DGG])) u4[3:DDG]))

((C*[1:G] u5[2:DD]) (B[1:G] (B[1:G] ((C*[2:GG] u2[2:GD]) (B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD])) (C*[2:DG] u6[3:DGG])) u4[3:DDG]))

((C*[1:G] u5[2:DD]) (B[1:G] (B[1:G] ((C*[2:GG] u2[2:GD]) (B[2:GG] (C*[2:GG] u3[3:GGG]) u1[3:GDG])) (C*[2:DG] u6[3:DGG])) u4[3:DDG]))

((C*[1:G] u5[2:DD]) (B[1:G] (B[1:G] ((C*[2:GG] u2[2:GG]) (B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG])) (C*[2:DG] u6[3:DGG])) u4[3:DDG]))

((C*[1:G] u5[2:DG]) (B[1:G] (B[1:G] ((C*[2:GG] u2[2:GG]) (B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG])) (C*[2:DG] u6[2:DD])) u4[3:DGG]))

Ordre d'introduction calculé:

C*[2:DG], C*[2:GG], B[2:GG], C*[2:GG], B[1:G], B[1:G], C*[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) ((u4 u5) u6))

(((u1 u2) u3) ((C* u6) (u4 u5)))

(((C* u3) (u1 u2)) ((C* u6) (u4 u5)))

(((B (C* u3) u1) u2) ((C* u6) (u4 u5)))

(((C* u2) (B (C* u3) u1)) ((C* u6) (u4 u5)))

((B ((C* u2) (B (C* u3) u1)) (C* u6)) (u4 u5))

((B (B ((C* u2) (B (C* u3) u1)) (C* u6)) u4) u5)

((C* u5) (B (B ((C* u2) (B (C* u3) u1)) (C* u6)) u4))

20. ((B (C* ((B (C* (u_7 u_8)) (B (u_3 u_4) u_5)) u_6)) u_1) u_2)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (C*[2:GG] ((B[4:GGGG] (C*[5:GGGGG] (u7[7:GGGGGGG] u8[7:GGGGGGD])) (B[5:GGGGD] (u3[7:GGGGDGG] u4[7:GGGGDGD]) u5[6:GGGGDD])) u6[4:GGGD])) u1[2:GD]) u2[1:D])

((B[1:G] (C*[1:G] ((B[3:GGG] (C*[4:GGGG] (u7[6:GGGGGG] u8[6:GGGGGD])) (B[4:GGGD] (u3[6:GGGDGG] u4[6:GGGDGD]) u5[5:GGGDD])) u6[3:GGD])) u1[2:DG]) u2[2:DD])

((B[1:G] (C*[1:G] ((B[2:DG] (C*[3:DGG] (u7[5:DGGGG] u8[5:DGGGD])) (B[3:DGD] (u3[5:DGDGG] u4[5:DGDGD]) u5[4:DGDD])) u6[2:DD])) u1[2:GG]) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] (u7[4:DGGG] u8[4:DGGD])) (B[3:DDG] (u3[5:DDGGG] u4[5:DDGGD]) u5[4:DDGD])) u6[3:DDD])) u1[2:GG]) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] (u7[3:DDG] u8[3:DDD])) (B[3:DGG] (u3[5:DGGGG] u4[5:DGGGD]) u5[4:DGGD])) u6[3:DGD])) u1[2:GG]) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] (u7[3:DDG] u8[3:DDD])) (B[3:DGG] (u3[4:DGGG] u4[4:DGGD]) u5[4:DGDG])) u6[4:DGDD])) u1[2:GG]) u2[2:GD])

Ordre d'introduction calculé:

B[3:DGG], C*[2:DG], B[2:DG], C*[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

((u1 u2) (((u3 u4) (u5 u6)) (u7 u8)))

((u1 u2) (((B (u3 u4) u5) u6) (u7 u8)))

((u1 u2) ((C* (u7 u8)) ((B (u3 u4) u5) u6)))

((u1 u2) ((B (C* (u7 u8)) (B (u3 u4) u5)) u6))

((C* ((B (C* (u7 u8)) (B (u3 u4) u5)) u6)) (u1 u2))

((B (C* ((B (C* (u7 u8)) (B (u3 u4) u5)) u6)) u1) u2)

21. ((C* ((C* u_6) u_5)) (B ((B u_1 u_2) u_3) u_4))

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] ((C*[3:GGG] u6[4:GGGG]) u5[3:GGD])) (B[1:D] ((B[3:DGG] u1[4:DGGG] u2[4:DGGD]) u3[3:DGD]) u4[2:DD]))

((C*[1:G] ((C*[2:DG] u6[3:DGG]) u5[2:DD])) (B[1:G] ((B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD]) u4[2:GD]))

((C*[1:G] ((C*[2:DG] u6[2:DD]) u5[2:DG])) (B[1:G] ((B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD]) u4[2:GD]))

((C*[1:G] ((C*[2:DG] u6[2:DD]) u5[2:DG])) (B[1:G] ((B[2:GG] u1[3:GGG] u2[3:GGD]) u3[2:GD]) u4[2:DG]))

((C*[1:G] ((C*[2:DG] u6[2:DD]) u5[2:DG])) (B[1:G] ((B[2:GG] u1[2:GG] u2[3:GDG]) u3[3:GDD]) u4[2:DG]))

Ordre d'introduction calculé:

B[2:GG], B[1:G], C*[2:DG], C*[1:G]

Introduction des combinateurs dans la forme normale:

((u1 (u2 u3)) (u4 (u5 u6)))

(((B u1 u2) u3) (u4 (u5 u6)))

((B ((B u1 u2) u3) u4) (u5 u6))

((B ((B u1 u2) u3) u4) ((C* u6) u5))

((C* ((C* u6) u5)) (B ((B u1 u2) u3) u4))

22. ((B (B (B u_1 u_2) u_3) (B (C* u_6) u_4)) u_5)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (B[2:GG] (B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD]) (B[2:GD] (C*[3:GDG] u6[4:GDGG]) u4[3:GDD])) u5[1:D])

((B[1:G] (B[1:G] (B[2:GG] u1[3:GGG] u2[3:GGD]) u3[2:GD]) (B[2:DG] (C*[3:DGG] u6[4:DGGG]) u4[3:DGD])) u5[2:DD])

((B[1:G] (B[1:G] (B[1:G] u1[2:GG] u2[2:GD]) u3[2:DG]) (B[3:DDG] (C*[4:DDGG] u6[5:DDGGG]) u4[4:DDGD])) u5[3:DDD])

((B[1:G] (B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) (B[4:DDDG] (C*[5:DDDGG] u6[6:DDDGGG]) u4[5:DDDGD])) u5[4:DDDD])

((B[1:G] (B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) (B[4:DDDG] (C*[4:DDDG] u6[5:DDDGG]) u4[5:DDDDG])) u5[5:DDDDD])

((B[1:G] (B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) (B[4:DDDG] (C*[4:DDDG] u6[4:DDDD]) u4[5:DDDGG])) u5[5:DDDGD])

Ordre d'introduction calculé:

C*[4:DDDG], B[4:DDDG], B[1:G], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(u1 (u2 (u3 ((u4 u5) u6))))

(u1 (u2 (u3 ((C* u6) (u4 u5)))))

(u1 (u2 (u3 ((B (C* u6) u4) u5))))

((B u1 u2) (u3 ((B (C* u6) u4) u5)))

((B (B u1 u2) u3) ((B (C* u6) u4) u5))

((B (B (B u1 u2) u3) (B (C* u6) u4)) u5)

23. ((B u_1 (B u_2 (C* ((B (C* u_6) (C* u_5))))) u_4) u_3)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] u1[2:GG] (B[2:GD] u2[3:GDG] (C*[3:GDD] ((B[5:GDDGG] (C*[6:GDDGGG] u6[7:GDDGGGG]) (C*[6:GDDGGD] u5[7:GDDGGDG])) u4[5:GDDGD]))))) u3[1:D])

((B[1:G] u1[1:G] (B[2:DG] u2[3:DGG] (C*[3:DGD] ((B[5:DGDGG] (C*[6:DGDGGG] u6[7:DGDGGGG]) (C*[6:DGDGGD] u5[7:DGDGGDG])) u4[5:DGDGD]))))) u3[2:DD])

((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] (C*[3:DDG] ((B[5:DDGGG] (C*[6:DDGGGG] u6[7:DDGGGGG]) (C*[6:DDGGGD] u5[7:DDGGGDG])) u4[5:DDGGD]))))) u3[3:DDD])

((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] (C*[3:DDG] ((B[4:DDDG] (C*[5:DDDGG] u6[6:DDDGGG]) (C*[5:DDDGD] u5[6:DDDGDG])) u4[4:DDDD]))))) u3[3:DDG])

((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] (C*[3:DDG] ((B[4:DDDG] (C*[4:DDDG] u6[5:DDDGG]) (C*[5:DDDDG] u5[6:DDDDGG])) u4[5:DDDDD]))))) u3[3:DDG])

((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] (C*[3:DDG] ((B[4:DDDG] (C*[4:DDDG] u6[4:DDDD]) (C*[5:DDDGG] u5[6:DDDGGG])) u4[5:DDDGD]))))) u3[3:DDG])

((B[1:G] u1[1:G] (B[2:DG] u2[2:DG] (C*[3:DDG] ((B[4:DDDG] (C*[4:DDDG] u6[4:DDDD]) (C*[5:DDDGG] u5[5:DDDGD])) u4[5:DDDGG]))))) u3[3:DDG])

Ordre d'introduction calculé:

C*[5:DDDGG], C*[4:DDDG], B[4:DDDG], C*[3:DDG], B[2:DG], B[1:G]

Introduction des combinateurs dans la forme normale:

(u1 (u2 (u3 ((u4 u5) u6))))

(u1 (u2 (u3 (((C* u5) u4) u6))))

(u1 (u2 (u3 ((C* u6) ((C* u5) u4)))))

(u1 (u2 (u3 ((B (C* u6) (C* u5)) u4))))

(u1 (u2 ((C* ((B (C* u6) (C* u5)) u4)) u3)))

(u1 ((B u2 (C* ((B (C* u6) (C* u5)) u4))) u3))

((B u1 (B u2 (C* ((B (C* u6) (C* u5)) u4)))) u3)

24. ((B (C* ((B (B (u_4 u_5) u_6) u_7) u_8)) (B (C* u_3) u_1)) u_2)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (C*[2:GG] ((B[4:GGGG] (B[5:GGGGG] (u4[7:GGGGGGG] u5[7:GGGGGGD]) u6[6:GGGGGD]) u7[5:GGGGD]) u8[4:GGGD])) (B[2:GD] (C*[3:GDG] u3[4:GDGG]) u1[3:GDD])) u2[1:D])

((B[1:G] (C*[1:G] ((B[3:GGG] (B[4:GGGG] (u4[6:GGGGGG] u5[6:GGGGGD]) u6[5:GGGGD]) u7[4:GGGD]) u8[3:GGD])) (B[2:DG] (C*[3:DGG] u3[4:DGGG]) u1[3:DGD])) u2[2:DD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[3:DGG] (u4[5:DGGGG] u5[5:DGGGD]) u6[4:DGGD]) u7[3:DGD]) u8[2:DD])) (B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD])) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] (u4[4:DGGG] u5[4:DGGD]) u6[3:DGD]) u7[3:DDG]) u8[3:DDD])) (B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD])) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] (u4[3:DGG] u5[3:DGD]) u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (C*[3:GGG] u3[4:GGGG]) u1[3:GGD])) u2[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] (u4[3:DGG] u5[3:DGD]) u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (C*[2:GG] u3[3:GGG]) u1[3:GDG])) u2[3:GDD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] (u4[3:DGG] u5[3:DGD]) u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (C*[2:GG] u3[2:GD]) u1[3:GGG])) u2[3:GGD])

Ordre d'introduction calculé:

C*[2:GG], B[2:GG], B[2:DG], B[2:DG], C*[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) ((u4 u5) (u6 (u7 u8))))

(((C* u3) (u1 u2)) ((u4 u5) (u6 (u7 u8))))

(((B (C* u3) u1) u2) ((u4 u5) (u6 (u7 u8))))

(((B (C* u3) u1) u2) ((B (u4 u5) u6) (u7 u8)))

(((B (C* u3) u1) u2) ((B (B (u4 u5) u6) u7) u8))

((C* ((B (B (u4 u5) u6) u7) u8)) ((B (C* u3) u1) u2))

((B (C* ((B (B (u4 u5) u6) u7) u8)) (B (C* u3) u1)) u2)

25. ((B (B (C* ((B (C* ((C* u_8) (B u_6 u_7))) (C* u_5)) u_4)) (C* u_3)) (C* u_2)) u_1)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (B[2:GG] (C*[3:GGG] ((B[5:GGGGG] (C*[6:GGGGGG] ((C*[8:GGGGGGGG] u8[9:GGGGGGGGG]) (B[8:GGGGGGGD] u6[9:GGGGGGGDG] u7[9:GGGGGGGDD])))) (C*[6:GGGGGD] u5[7:GGGGGDG])) u4[5:GGGGD])) (C*[3:GGD] u3[4:GGDG])) (C*[2:GD] u2[3:GDG])) u1[1:D])

((B[1:G] (B[1:G] (C*[2:GG] ((B[4:GGGG] (C*[5:GGGGG] ((C*[7:GGGGGGG] u8[8:GGGGGGGG]) (B[7:GGGGGGD] u6[8:GGGGGGDG] u7[8:GGGGGGDD])))) (C*[5:GGGGD] u5[6:GGGGDG])) u4[4:GGGD])) (C*[2:GD] u3[3:GDG])) (C*[2:DG] u2[3:DGG])) u1[2:DD])

((B[1:G] (B[1:G] (C*[1:G] ((B[3:GGG] (C*[4:GGGG] ((C*[6:GGGGGG] u8[7:GGGGGGG]) (B[6:GGGGGD] u6[7:GGGGGDG] u7[7:GGGGGDD])))) (C*[4:GGGD] u5[5:GGGDG])) u4[3:GGD])) (C*[2:DG] u3[3:DGG])) (C*[3:DDG] u2[4:DDGG])) u1[3:DDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[3:DGG] ((C*[5:DGGGG] u8[6:DGGGGG]) (B[5:DGGGD] u6[6:DGGGDG] u7[6:DGGGDD])))) (C*[3:DGD] u5[4:DGDG])) u4[2:DD])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[4:DGGG] u8[5:DGGGG]) (B[4:DGGD] u6[5:DGGDG] u7[5:DGGDD])))) (C*[3:DDG] u5[4:DDGG])) u4[3:DDD])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[4:DDGG]) (B[3:DDD] u6[4:DDDG] u7[4:DDDD])))) (C*[3:DGG] u5[4:DGGG])) u4[3:DGD])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[3:DDD]) (B[3:DDG] u6[4:DDGG] u7[4:DDGD])))) (C*[3:DGG] u5[4:DGGG])) u4[3:DGD])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[3:DDD]) (B[3:DDG] u6[3:DDG] u7[4:DDDG]))) (C*[3:DGG] u5[4:DGGG])) u4[3:DGD])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[3:DDD]) (B[3:DDG] u6[3:DDG] u7[4:DDDG]))) (C*[3:DGG] u5[3:DGD])) u4[3:DGG])) (C*[2:GG] u3[3:GGG])) (C*[3:GDG] u2[4:GDGG])) u1[3:GDD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[3:DDD]) (B[3:DDG] u6[3:DDG] u7[4:DDDG]))) (C*[3:DGG] u5[3:DGD])) u4[3:DGG])) (C*[2:GG] u3[2:GD])) (C*[3:GGG] u2[4:GGGG])) u1[3:GGD])

((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] ((C*[3:DDG] u8[3:DDD]) (B[3:DDG] u6[3:DDG] u7[4:DDDG]))) (C*[3:DGG] u5[3:DGD])) u4[3:DGG])) (C*[2:GG] u3[2:GD])) (C*[3:GGG] u2[3:GGD])) u1[3:GGG])

Ordre d'introduction calculé:

C*[3:GGG], C*[2:GG], C*[3:DGG], B[3:DDG], C*[3:DDG], C*[2:DG], B[2:DG], C*[1:G], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(((u1 u2) u3) ((u4 u5) (u6 (u7 u8))))

((((C* u2) u1) u3) ((u4 u5) (u6 (u7 u8))))

(((C* u3) ((C* u2) u1)) ((u4 u5) (u6 (u7 u8))))

(((C* u3) ((C* u2) u1)) (((C* u5) u4) (u6 (u7 u8))))

(((C* u3) ((C* u2) u1)) (((C* u5) u4) ((B u6 u7) u8)))

(((C* u3) ((C* u2) u1)) (((C* u5) u4) ((C* u8) (B u6 u7))))

(((C* u3) ((C* u2) u1)) ((C* ((C* u8) (B u6 u7))) ((C* u5) u4)))

(((C* u3) ((C* u2) u1)) ((B (C* ((C* u8) (B u6 u7))) (C* u5)) u4))

((C* ((B (C* ((C* u8) (B u6 u7))) (C* u5)) u4)) ((C* u3) ((C* u2) u1)))

((B (C* ((B (C* ((C* u8) (B u6 u7))) (C* u5)) u4)) (C* u3)) ((C* u2) u1))

((B (B (C* ((B (C* ((C* u8) (B u6 u7))) (C* u5)) u4)) (C* u3)) (C* u2)) u1)

26. ((B (C* ((B (B u_5 u_6) u_7) u_8)) (B (B u_1 u_2) u_3)) u_4)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (C*[2:GG] ((B[4:GGGG] (B[5:GGGGG] u5[6:GGGGGG] u6[6:GGGGGD]) u7[5:GGGGD]) u8[4:GGGD])) (B[2:GD] (B[3:GDG] u1[4:GDGG] u2[4:GDGD]) u3[3:GDD])) u4[1:D])

((B[1:G] (C*[1:G] ((B[3:GGG] (B[4:GGGG] u5[5:GGGGG] u6[5:GGGGD]) u7[4:GGGD]) u8[3:GGD])) (B[2:DG] (B[3:DGG] u1[4:DGGG] u2[4:DGGD]) u3[3:DGD])) u4[2:DD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[3:DGG] u5[4:DGGG] u6[4:DGGD]) u7[3:DGD]) u8[2:DD])) (B[2:GG] (B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD])) u4[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] u5[3:DGG] u6[3:DGD]) u7[3:DDG]) u8[3:DDD])) (B[2:GG] (B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD])) u4[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] u5[2:DG] u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD])) u4[2:GD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] u5[2:DG] u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (B[2:GG] u1[3:GGG] u2[3:GGD]) u3[3:GDG])) u4[3:GDD])

((B[1:G] (C*[1:G] ((B[2:DG] (B[2:DG] u5[2:DG] u6[3:DDG]) u7[4:DDDG]) u8[4:DDDD])) (B[2:GG] (B[2:GG] u1[2:GG] u2[3:GDG]) u3[4:GDDG])) u4[4:GDDD])

Ordre d'introduction calculé:

B[2:GG], B[2:GG], B[2:DG], B[2:DG], C*[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

((u1 (u2 (u3 u4))) (u5 (u6 (u7 u8))))

(((B u1 u2) (u3 u4)) (u5 (u6 (u7 u8))))

(((B (B u1 u2) u3) u4) (u5 (u6 (u7 u8))))

(((B (B u1 u2) u3) u4) ((B u5 u6) (u7 u8)))

(((B (B u1 u2) u3) u4) ((B (B u5 u6) u7) u8))

((C* ((B (B u5 u6) u7) u8)) ((B (B u1 u2) u3) u4))

((B (C* ((B (B u5 u6) u7) u8)) (B (B u1 u2) u3)) u4)

27. ((B (B (B (B (B (B (B u_1 u_2) u_3) u_4) u_5) u_6) (C* u_9)) u_7) u_8)

Calcul des chemins d'introduction des combinateurs:

((B[1:G] (B[2:GG] (B[3:GGG] (B[4:GGGG] (B[5:GGGGG] (B[6:GGGGGG] (B[7:GGGGGGG] u1[8:GGGGGGGG] u2[8:GGGGGGGD]) u3[7:GGGGGGD]) u4[6:GGGGGD]) u5[5:GGGGD]) u6[4:GGGD]) (C*[3:GGD] u9[4:GGDG])) u7[2:GD]) u8[1:D])

((B[1:G] (B[1:G] (B[2:GG] (B[3:GGG] (B[4:GGGG] (B[5:GGGGG] (B[6:GGGGGG] u1[7:GGGGGGG] u2[7:GGGGGGD]) u3[6:GGGGGD]) u4[5:GGGGD]) u5[4:GGGD]) u6[3:GGD]) (C*[2:GD] u9[3:GDG])) u7[2:DG]) u8[2:DD])

((B[1:G] (B[1:G] (B[1:G] (B[2:GG] (B[3:GGG] (B[4:GGGG] (B[5:GGGGG] u1[6:GGGGGG] u2[6:GGGGGD]) u3[5:GGGGD]) u4[4:GGGD]) u5[3:GGD]) u6[2:GD]) (C*[2:DG] u9[3:DGG])) u7[3:DDG]) u8[3:DDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[2:GG] (B[3:GGG] (B[4:GGGG] u1[5:GGGGG] u2[5:GGGGD]) u3[4:GGGD]) u4[3:GGD]) u5[2:GD]) u6[2:DG]) (C*[3:DDG] u9[4:DDGG])) u7[4:DDDG]) u8[4:DDDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[2:GG] (B[3:GGG] u1[4:GGGG] u2[4:GGGD]) u3[3:GGD]) u4[2:GD]) u5[2:DG]) u6[3:DDG]) (C*[4:DDDG] u9[5:DDDGG])) u7[5:DDDDG]) u8[5:DDDDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[2:GG] u1[3:GGG] u2[3:GGD]) u3[2:GD]) u4[2:DG]) u5[3:DDG]) u6[4:DDDG]) (C*[5:DDDDG] u9[6:DDDDGG])) u7[6:DDDDDG]) u8[6:DDDDDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] u1[2:GG] u2[2:GD]) u3[2:DG]) u4[3:DDG]) u5[4:DDDG]) u6[5:DDDDG]) (C*[6:DDDDDG] u9[7:DDDDDGG])) u7[7:DDDDDDG]) u8[7:DDDDDDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) u4[4:DDDG]) u5[5:DDDDG]) u6[6:DDDDDG]) (C*[7:DDDDDDG] u9[8:DDDDDDGG])) u7[8:DDDDDDDG]) u8[8:DDDDDDDD])

((B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] (B[1:G] u1[1:G] u2[2:DG]) u3[3:DDG]) u4[4:DDDG]) u5[5:DDDDG]) u6[6:DDDDDG]) (C*[7:DDDDDDG] u9[7:DDDDDDD])) u7[8:DDDDDDGG]) u8[8:DDDDDDGD])

Ordre d'introduction calculé:

C*[7:DDDDDDG], B[1:G], B[1:G], B[1:G], B[1:G], B[1:G], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:

(u1 (u2 (u3 (u4 (u5 (u6 ((u7 u8) u9)))))))

(u1 (u2 (u3 (u4 (u5 (u6 ((C* u9) (u7 u8))))))))

((B u1 u2) (u3 (u4 (u5 (u6 ((C* u9) (u7 u8)))))))

((B (B u1 u2) u3) (u4 (u5 (u6 ((C* u9) (u7 u8))))))

((B (B (B u1 u2) u3) u4) (u5 (u6 ((C* u9) (u7 u8)))))

((B (B (B (B u1 u2) u3) u4) u5) (u6 ((C* u9) (u7 u8))))

((B (B (B (B (B u1 u2) u3) u4) u5) u6) ((C* u9) (u7 u8)))

((B (B (B (B (B (B u1 u2) u3) u4) u5) u6) (C* u9)) (u7 u8))

((B (B (B (B (B (B (B u1 u2) u3) u4) u5) u6) (C* u9)) u7) u8)

28. ((B u_1 (C* u_3)) u_2)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] u1[2:GG] (C*[2:GD] u3[3:GDG])) u2[1:D])
((B[1:G] u1[1:G] (C*[2:DG] u3[3:DGG])) u2[2:DD])
((B[1:G] u1[1:G] (C*[2:DG] u3[2:DD])) u2[2:DG])

Ordre d'introduction calculé:
C*[2:DG], B[1:G]

Introduction des combinateurs dans la forme normale:
(u1 (u2 u3))
(u1 ((C* u3) u2))
((B u1 (C* u3)) u2)

29. ((C* ((C* u_4) u_3)) (B u_1 u_2))

Calcul des chemins d'introduction des combinateurs:

((C*[1:G] ((C*[3:GGG] u4[4:GGGG]) u3[3:GGD])) (B[1:D] u1[2:DG] u2[2:DD]))

((C*[1:G] ((C*[2:DG] u4[3:DGG]) u3[2:DD])) (B[1:G] u1[2:GG] u2[2:GD]))

((C*[1:G] ((C*[2:DG] u4[2:DD]) u3[2:DG])) (B[1:G] u1[2:GG] u2[2:GD]))

((C*[1:G] ((C*[2:DG] u4[2:DD]) u3[2:DG])) (B[1:G] u1[1:G] u2[2:DG]))

Ordre d'introduction calculé:

B[1:G], C*[2:DG], C*[1:G]

Introduction des combinateurs dans la forme normale:

(u1 (u2 (u3 u4)))

((B u1 u2) (u3 u4))

((B u1 u2) ((C* u4) u3))

((C* ((C* u4) u3)) (B u1 u2))

30. ((B (B (C* ((B (C* u_6) u_4) u_5)) (C* u_3)) u_1) u_2)

Calcul des chemins d'introduction des combinateurs:
((B[1:G] (B[2:GG] (C*[3:GGG] ((B[5:GGGGG] (C*[6:GGGGGG] u6[7:GGGGGGG]) u4[6:GGGGGD]) u5[5:GGGGD])) (C*[3:GGD] u3[4:GGDG])) u1[2:GD]) u2[1:D])
((B[1:G] (B[1:G] (C*[2:GG] ((B[4:GGGG] (C*[5:GGGGG] u6[6:GGGGGG]) u4[5:GGGGD]) u5[4:GGGD])) (C*[2:GD] u3[3:GDG])) u1[2:DG]) u2[2:DD])
((B[1:G] (B[1:G] (C*[1:G] ((B[3:GGG] (C*[4:GGGG] u6[5:GGGGG]) u4[4:GGGD]) u5[3:GGD])) (C*[2:DG] u3[3:DGG])) u1[3:DDG]) u2[3:DDD])
((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[3:DGG] u6[4:DGGG]) u4[3:DGD]) u5[2:DD])) (C*[2:GG] u3[3:GGG])) u1[3:GDG]) u2[3:GDD])
((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] u6[3:DGG]) u4[3:DDG]) u5[3:DDD])) (C*[2:GG] u3[3:GGG])) u1[3:GDG]) u2[3:GDD])
((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] u6[2:DD]) u4[3:DGG]) u5[3:DGD])) (C*[2:GG] u3[3:GGG])) u1[3:GDG]) u2[3:GDD])
((B[1:G] (B[1:G] (C*[1:G] ((B[2:DG] (C*[2:DG] u6[2:DD]) u4[3:DGG]) u5[3:DGD])) (C*[2:GG] u3[2:GD])) u1[3:GGG]) u2[3:GGD])

Ordre d'introduction calculé:
C*[2:GG], C*[2:DG], B[2:DG], C*[1:G], B[1:G], B[1:G]

Introduction des combinateurs dans la forme normale:
(((u1 u2) u3) ((u4 u5) u6))
(((C* u3) (u1 u2)) ((u4 u5) u6))
(((C* u3) (u1 u2)) ((C* u6) (u4 u5)))
(((C* u3) (u1 u2)) ((B (C* u6) u4) u5))
((C* ((B (C* u6) u4) u5)) ((C* u3) (u1 u2)))
((B (C* ((B (C* u6) u4) u5)) (C* u3)) (u1 u2))
((B (B (C* ((B (C* u6) u4) u5)) (C* u3)) u1) u2)

ANNEXE 2

PUBLICATIONS

Combinators Introduction : An Algorithm

Adam Joly & Ismaïl Biskri

Département de Mathématiques et Informatique – Université du Québec à Trois-Rivières
CP 500, Trois-Rivières, (QC) G9A 5H7, Canada
{ismail.biskri ; adam.joly}@uqtr.ca

ASTRACT

The accurate use of combinatory logic and combinators in natural language processing needs a strategy for the removal of combinators, but also for their introduction. The tour of scientific literature teaches us how to reduce combinators and construct from a combinatory expression a normal form without combinators, however no strategy has been proposed to automate the introduction of combinators and construct from one normal form one combinatory expression. We show in our paper that such a strategy is possible. An algorithm is also described.

1. Introduction to Applicative Combinatory Categorial Grammar

According to the framework of Applicative and Cognitive Grammar (Desclés, 1996) (Desclés, 1990) and Universal Applicative Grammar (Shaumyan, 1998), language analysis must postulate three levels of representation: (i) the morpho-syntactical level, where specific characteristics of the language are expressed (such as word order, morphological cases, ellipsis, etc). The expressions of this level are concatenated linguistic units u1 - u2 - u3 obeying the syntagmatic rules of the concerned language; (ii) the predicative level, where the logical and grammatical representations of the statements of the phenotype are expressed. This level uses a formal applicative language without variables as a formal meta-language to describe the languages. It makes it possible to express functional semantic interpretation. (iii) The cognitive level, where the meanings of the lexical predicates are semantically expressed by means of the combinators of typed combinatory logic. The representations of levels two and three are expressions of typed combinatory logic (Shaumyan, 1998) (Curry, Feys, 1958). This logic was developed to analyze Russell paradoxes and the concept of substitution. Just as in the lambda-calculus of Church, combinatory logic is currently used by specialists in informatics to analyze the semantic properties of high level programming languages.

The principal difference between the two logics lies in the fact that combinatory logic is a variable-free logic. It allows for the avoidance of one of the known problems of Lambda-Calculus, which is the telescoping of variables (two different variables with the same identifier). Combinatory logic uses abstract operators called combinators to express complex concepts. They make it possible to construct more complex operators starting from more elementary operators. Each combinator is introduced or eliminated by a β-reduction. For illustration, we present the β-reduction rules of **Φ, B** and **C*** [1] (U1, U2, U3, U4 being typed applicative expressions which function either like operators or like operands):

$$(\Phi\ U1\ U2\ U3)\ U4 \rightarrow U1(U2\ U4)\ (U3\ U4)$$
$$((\mathbf{B}\ U_1\ U_2)\ U3) \rightarrow (U_1\ (U2\ U_3))$$
$$((\mathbf{C}_*\ U_1)\ U_2) \rightarrow (U2\ U_1)$$

The combinator **Φ** makes it possible to distribute the application of two typed applicative expressions U2 and U3 (that function as operators) to the typed applicative expression U4 (that functions like an operand). The combinator **B** allows for the composition of two typed applicative expressions U1 and U2 (U1 and U2 function as operators). The result (**B** U1 U2) would then be the complex operator of the typed applicative expression U3 (U3 functions like an operand). The combinator **C*** is applied to a typed applicative expression U1 (U1 functions as the operand of U2). This makes it possible to build the complex operator (**C*** U1) which can be applied to the typed applicative expression U2. According to the Church-Rosser Theorem, these rules establish a relationship, which is independent of the meaning of the arguments, between an expression with combinators and a single expression (if it exists) without combinators equivalent to the first (from a certain point of view). This relationship is called the normal form. In the ACCG model, normal forms represent functional semantic interpretation. In addition, a paraphrastic reduction to a normal form is also possible.

[1] There are other combinators. Here we are only interested in those used in this paper. For more details the reader might have a look at (Desclés, 1990).

The reduction of a complex combinatory expression in a normal form is obtained by eliminating combinators, according to the β-reduction rules, from left to right. With this strategy, a unique sequence for the elimination of combinators is possible.

((**B** U1 (**C*** U2)) U3)
(U1 ((**C*** U2) U3)
(U1 (U3 U2))

The model of Applicative and Combinatory Categorial Grammar (ACCG) (Biskri, Desclés, 1997), as do most of the categorial models (Dowty, 2000) (Morrill, 1994) (Moorgat, 1997) (Steedman, 2000) (Baldridge, Kruijff, 2003), falls under a paradigm of language analysis that favours complete abstraction of grammatical structure from its linear representation, due to the linearity of the linguistic signs, and a complete abstraction of grammar from the lexicon. ACCG conceptualizes languages as a sequence of linguistic units, of which some function as operators whereas others function as operands. Concretely, ACCG assigns syntactical categories to each linguistic unit in order to express its function. The basic syntactical categories N and S are assigned respectively to noun phrases and sentences. The orientated syntactical categories, developed from basic types by means of the two operators of type construction "/" and "\", are assigned to the linguistic units which function as operators. For example, the category (S\N)/N is assigned to transitive verbs which are consequently seen as operators with two operands, the first being the object of type N positioned to its right, and the second one being the subject of type N positioned to its left. In our paper, a linguistic unit u with the type X will be noted by [X : u].
According to the postulate that the representation of language is performed on three levels, ACCG makes it possible, by means of rules, to: (1) ascertain syntactic correctness; (2) progressively construct the semantic functional interpretation; (3) allow a functional analysis of a linguistic marker (example: and,…).
The premise of each rule is a concatenation of linguistic units with oriented types. The consequence of each rule is an applicative typed expression with the possible introduction of one combinator. The type-raising of one unit u introduces the combinator **C***; the composition of two concatenated units introduces the combinator **B**.

Application rules :

$[X/Y : u_1] - [Y : u_2]$ ----------> ; $[X : (u_1\ u_2)]$

$[Y : u_1] - [X\Y : u_2]$ ----------< $[X : (u_2\ u_1)]$

Type raising rules :

$[X : u]$ ----------**>T** ; $[Y/(Y\X) : (\mathbf{C}_{*}\ u)]$

$[X : u]$ ----------**<T** $[Y\(Y/X) : (\mathbf{C}_{*}\ u)]$

$[X : u]$ ----------**>Tx** ; $[Y/(Y/X) : (\mathbf{C}_{*}\ u)]$

$[X : u]$ ----------**<Tx** $[Y\(Y\X) : (\mathbf{C}_{*}\ u)]$

functional composition rules :

$[X/Y : u_1]$-$[Y/Z : u_2]$ ----------**>B** ; $[X/Z : (\mathbf{B}\ u_1\ u_2)]$

$[Y\Z : u_1]$-$[X\Y : u_2]$ ----------**<B** $[X\Z : (\mathbf{B}\ u_2\ u_1)]$

$[X/Y : u_1]$-$[Y\Z : u_2]$ ----------**>Bx**; $[X\Z : (\mathbf{B}\ u_1\ u_2)]$

$[Y/Z : u_1]$-$[X\Y : u_2]$ ----------**<Bx** $[X/Z : (\mathbf{B}\ u_2\ u_1)]$

An analysis based on ACCG rests on the General following steps:
(i) A first step which consists in assigning syntactic types to the lexical units. Those are entries of a dictionary where each unit is associated to one or more types.
(ii) A second step consists in operating the rules of the ACCG in the way to check the syntactic correctness on the one hand and progressively to build the applicative structures by the introduction of combinators with the syntactic process. Two results are obtained at the end of this step. The first one is the type S (or another basic type) which confirms the syntactic correction of the analyzed statement. The second one is the applicative expression with combinators which after their reduction gives the functional semantic interpretation in which each operator is followed by its operands. This analysis looks like a compilation process.

Let us deal with this example with a non-correlative coordination : *Jean aime Marie tendrement et Sophie Sauvagement* (*Jean Loves Marie madly and Sophie wildly*).

1 [N: *Jean*] - [(S\N)/N : *aime*]-[N : *Marie*] - [(S\N)\(S\N) : *tendrement*]-[(X\X)/X : *et*]-[N : *Sophie*]-[(S\N)\(S\N) : *sauvagement*]
2 [S/(S\N) : (**C*** *Jean*)] - [(S\N)/N : *aime*]-[N : *Marie*] - [(S\N)\(S\N) : *tendrement*] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*] (**>T**)
3 [S/N : (**B** (**C*** *Jean*) *aime*)] - [N : *Marie*] - [(S\N)\(S\N) : *tendrement*] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*] (**>B**)
4 [S : ((**B** (**C*** *Jean*) *aime*) *Marie*)] - [(S\N)\(S\N) : *tendrement*] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*] (**>**)
5 [S : ((**C*** *Jean*) (*aime Marie*))] - [(S\N)\(S\N) : *tendrement*] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*]
6 [S/(S\N) : (**C*** *Jean*)] – [S\N : (*aime Marie*)] - [(S\N)\(S\N) : *tendrement*] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*]
7 [S/(S\N) : (**C*** *Jean*)] – [S\N : (*tendrement* (*aime Marie*))] - [(X\X)/X : *et*] - [N : *Sophie*] - [(S\N)\(S\N) : *sauvagement*] (**<**)
8 [S:((**C*** *Jean*) (*tendrement* (*aime Marie*))]-[(X\X)/X : *et*]-[N:*Sophie*]-[(S\N)\(S\N):*sauvagement*] (**>**)
9 [S:((**C*** *Jean*) (*tendrement* (*aime Marie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**C***Sophie*)]-[(S\N)\(S\N):*sauvagement*] (**<T**)
10 [S:((**C***Jean*) (*tendrement* (*aime Marie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**B** *sauvagement* (**C*** *Sophie*))] (**<B**)
11 [S/(S\N):(**C*** *Jean*)]-[S\N:(*tendrement* (*aime Marie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**B** *sauvagement* (**C*** *Sophie*))]

12 [S/(S\N):(C* *Jean*)]-[S\N:((B *tendrement* (C* *Marie*)) *aime*)]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N) : (B *sauvagement* (C* *Sophie*))]
13 [S/(S\N) : (C**Jean*)]-[(S\N)/N:*aime*]-[(S\N)\((S\N)/N) : (B *tendrement* (C* *Marie*))]-[(X\X)/X : *et*]-(S\N)\((S\N)/N) : (B *sauvagement* (C* *Sophie*))]
14 [S/(S\N) : (C**Jean*)]-[(S\N)/N:*aime*]-[(S\N)\((S\N)/N) : (B *tendrement* (C* *Marie*))]-[((S\N)\((S\N)/N))\((S\N)\((S\N)/N)) : (*et* (B *sauvagement* (C* *Sophie*)))] (>)
15 [S/(S\N) : (C**Jean*)]-[(S\N)/N:*aime*]-[(S\N)\((S\N)/N) : ((*et* (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*)))] (<)
16 [S/(S\N) : (C**Jean*)] - [(S\N) : (((*et* (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*))) *aime*)] (<)
17 [S : ((C**Jean*) (((*et* (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*))) *aime*))] (>)

18 ((C**Jean*) (((*et* (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*))) *aime*))
19 ((((*et* (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*))) *aime*) *Jean*) C*
20 ((((Φ ∧ (B *sauvagement* (C* *Sophie*))) (B *tendrement* (C* *Marie*))) *aime*) *Jean*) (et = Φ ∧)
21 ((∧ ((B *sauvagement* (C* *Sophie*)) *aime*) ((B *tendrement* (C* *Marie*)) *aime*)) *Jean*) Φ
22 ((∧ (*tendrement* ((C* *Marie*) *aime*)) ((B *sauvagement* (C* *Sophie*)) *aime*)) *Jean*) B
23 ((∧ (*tendrement* (*aime Marie*)) ((B *sauvagement* (C* *Sophie*)) *aime*)) *Jean*) C*
24 ((∧ (*tendrement* (*aime Marie*)) (*sauvagement* ((C* *Sophie*) *aime*))) *Jean*) B
25 ((∧ (*tendrement* (*aime Marie*)) (*sauvagement* (*aime Sophie*))) *Jean*) C*

First of all, this sentence is ambiguous. Two interpretations are possible. The first one is : *Jean aime Marie tendrement et Jean aime Sophie sauvagement* (*Jean loves Marie madly and Jean loves Sophie wildly*). The second one is : *Jean aime Marie tendrement et Sophie aime Marie sauvagement* (*Jean loves Marie madly and Sophie loves Marie wildly*). We chose to present the analysis which carries out towards the first interpretation to facilitate the reading of this paper.

Thus, the analysis starts with the assignment of the syntactic categories to the lexemes. For recall, each syntactic category describes the way in which a lexeme operates on its arguments. The category (X\X)/X assigned to the conjunction is in fact a scheme of type which describes the conjunction like an operator whose first and second operands, of type X, are respectively the second member and the first member of the coordination. The type of the coordination (S\N)\((S\N)/N) which will be substituted to X is known after the construction of the second member of coordination (step 10).

Steps 1 to 17 represent the application of ACCG rules. With these steps we verify the correctness of the sentence (the type S obtained at 17).

Steps 18 to 25 are in the predicative level. They reduce combinators in order to construct the functional semantic interpretation (the normal form): ((∧ (*tendrement* (*aime Marie*)) (*sauvagement* (*aime Sophie*))) *Jean*), which is structured like a conjunctive clause. At the step 20 the linguistic predicate *et* (and) is replaced by its meaning in the cognitive level Φ ∧ in order to express the distributive and the conjunctive nature of et by respectively the combinator Φ and the logical connector ∧.

A strategy of incremental analysis (from left to right) with an "intelligent" backtrack (Biskri & Desclès, 1997) supplements the model of the GCCA in order to solve the problem of the pseudo-ambiguity which consists in a multitude of syntactic derivations (which are from a certain point of view equivalent) for the analysis of the same statement and which corresponds to the same semantic interpretation. However, this strategy leads to the construction of false constituents that require decompositions (Steedman, 2000) or structural reorganization (intelligent back track) (Biskri & Desclès, 2005; 1997) to bring out the right constituents.

The decomposition enshrines the principle of parametric neutrality to determine the category of the constituent to identify. This principle states that the result in a categorical rule associated with a premise used to determine the other premise. Specifically, the decomposition is a calculation on the categorical types only. It does not really take into account the functional semantic interpretation as a criterion. Therefore, the decomposition runs only on simple cases. Cases of coordination of non-constituents, for example, cause serious problems.

The structural reorganization uses semantic interpretation, in addition to a calculation on the categories, in the process of back tracking. This gives it more linguistic credibility, in addition to computational one.

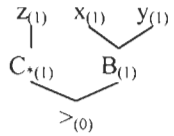
The principle is simple. The functional semantic interpretation is constructed using combinators introduced in the syntagmatic structure. When a false constituent is obtained, it is necessary to reduce a combinator (with using its b-reduction rule) and to test if the right constituent emerges (see steps 5, 6 for the first structural reorganization). The process is repeated until the right constituent emerges or there is no combinator to reduce (Biskri, Desclés, 1997). This process is complemented by some rules (a, b, c, d) of combinators introduction in the case of non-constituents coordination analysis (see steps 11, 12, 13). The analysis, based on the combinatory structure of the second member of the coordination must extract a combinatory structure for the first member of the coordination similar to the one of the second member. We apply at step 12 to (*tendrement* (*aime Marie*)) the rule (c) in order to get ((B *tendrement* (C* *Marie*)) *aime*) in which (B *tendrement* (C* *Marie*)) is the first member of the coordination.

(a) (u1 (u2 u3))<==>((B u1 u2) u3)
(b) ((u1 u2) u3)<==>((B (C* u3) u1) u2)
(c) (u1 (u2 u3))<==>((B u1 (C* u3)) u2)
(d) ((u1 u2) u3)<==>((B (C* u3) (C* u2)) u1)

These rules are static. What about if other cases appear ? We need an algorithm that completely automates the process of introducing combinators.

2. Combinators Introduction : an Algorithm

Every combinatory expression can be translated into a binary tree for convenience and visualization. For example, ((**C**$_*$ z) (**B** x y)) becomes:



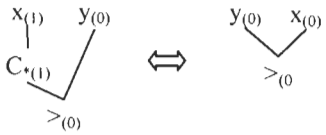
The number in parenthesis represents the node level, which can be a unit, a **C**$_*$ or **B** combinator or a forward application. We can notice that the level does not necessarily represent the deepness level. In the previous example, ***B***, *x* and *y* are at the same levels, just like ***C***$_*$ and *z* also are.

What we want is to reach a known combinatory expression, starting from its normal form.

Before inserting the combinators of the combinatory expression in the normal form, we must calculate their insertion levels. We will find them by taking in consideration how the **C**$_*$ and the **B** combinator's arguments levels change when we remove them.

First, let's take a look at the basic reduction of the **C**$_*$ combinator:

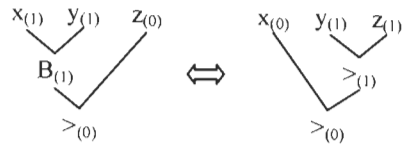
((**C**$_*$ x) y) ⇔ (y x)



We can observe that *x* level decreases by one with the **C**$_*$ removal, while y level still the same. It means that every time we reach a **C**$_*$ node, all children's levels will be reduce by one.

With the **B** combinator, the basic reduction expression is:

((**B** x y) z) ⇔ (x (y z))



Before the **B** combinator's insertion, the *x* argument is one level lower, the y level stays the same and the last argument level is one level higher. Each time we meet a **B** node, we have to reduce by one the level of every nodes representing the x argument. Likewise, the sidling node of the **B** combinator and its children have to add one level each.

The mechanism of leveling requires a binary tree structure and must be done recursively, starting from the root, then the right side of the binary tree, and finally the deepest node. As we stated before, each node in the tree of the combinatory expression has an initial level that will be adjust with what we will call a *level adjustment factor* to find the level where the corresponding combinator should be added.

Thereafter, the combinators will be introduced in the normal form in the reverse order where they appear in the combinatory expression (from the right to the left). The introduction levels will be found by taking the arguments levels of the combinators to be introduced.

Considering that the method takes as inputs a node and a level adjustment factor and that forward application's arguments are, x and y, **C**$_*$ combinator's argument is x and **B** combinator's arguments are x, y and z, the recursive algorithm goes as following:

```
method calculateNodeLevel
   if the current node is the z argument of a B combinator (see
   the scheme) then
         add 1 to the level adjustment factor
   end if
   current node's calculated introduction level = initial current
   node level + level adjustment factor
   if the current node is a forward application then
      call calculateNodeLevel method for y and with the level
      adjustment factor
      call calculateNodeLevel method for x and with the level
      adjustment factor
   else if the current node is a B combinator then
      call calculateNodeLevel method for y and with the level
      adjustment factor
      call calculateNodeLevel method for x and with the level
      adjustment factor - 1
   else if the current node is a C* combinator then
      call calculateNodeLevel method for x and with the level
      adjustment factor - 1
   end if
   return
end of method
```

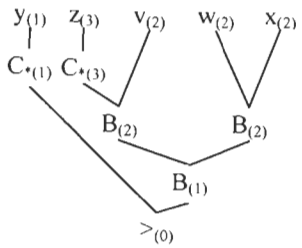
If the current node is a unit (not a combinator), it means it is a leave and there is no more recursive call for this branch.

The overall process can be translated in a main method that has two inputs: a combinatory expression and its normal form.

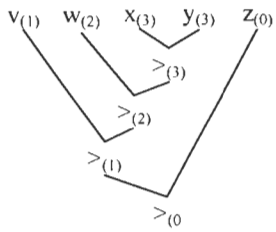
```
main method
   - build the binary tree corresponding to the combinatory
   expression
   - calculate nodes levels
   - introduce one by one the combinators in the normal form in
   the reverse order they appear in the combinatory expression
end of method
```

Leveling in the following example shows the algorithm execution, in relation with the nodes normal path. Only nodes causing level adjustments will be illustrated and,

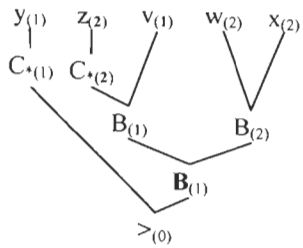
for simplification, they will be applied immediately for every child nodes, instead of waiting to reach each node and add the overall level adjustment factor.
The combinatory expression we will take as an example is ((**C*** y) (**B** (**B** (**C*** z) v) (**B** w x))). Below, we have the resulting binary tree of the expression:



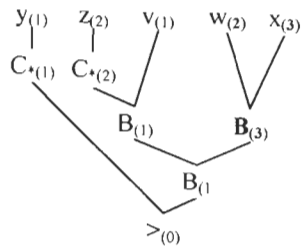
The normal form of the expression, after the ß-reduction, is ((v (w (x y))) z) and can be represented by the following tree:



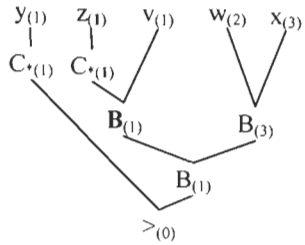
The next step consists to calculate the introduction combinators levels. In respect with the recursive algorithm, the nodes path will be >, **B**, **B**, x, w, **B**, v, **C***, z, **C*** and y.
First, from the root, we reach the **B** node at the right. We will have to clear the next right **B** path first, then the left **B**, but as we said, we immediately reduce levels of all left branch nodes by 1 for convenience.



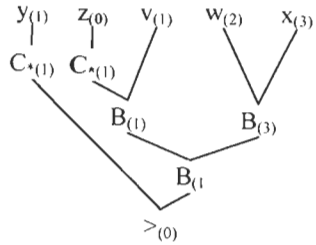
Again, we have a **B** combinator and the same rule applies. So, w level will eventually be reduced by 1. In addition, this **B** node has a sidling **B** node to the left, which means we have to add 1 level to the **B**, w and x nodes.



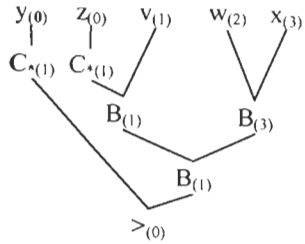
After the x and w nodes, we reach the last **B** node. Once more, the **C*** and z nodes will lose one level.



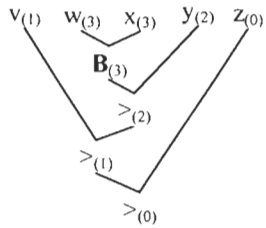
The next node will be v, then the **C*** combinator. As a consequence, z level will be decreased again by one.



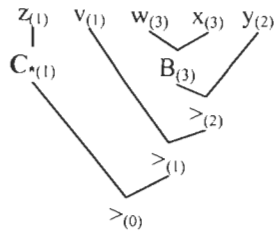
Finally, after y and back to the root, we take the left path and reach the last combinator. By the same logic, y level will go from 1 to 0.



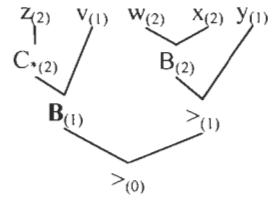
Now that we have calculated levels of the combinators, we can introduce them, as we said, from the right to the left, at their arguments levels.
The top right side combinatory will be the first to be introduced. Because its calculated level is 3, it means that before introducing it, its first argument was at level 2 and the two others were at level 3.

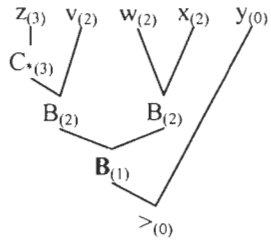


The second node to be introduced will be the **C*** combinatory with the z argument. Being at level 1, its argument is one level below before the **C*** introduction (level 0).

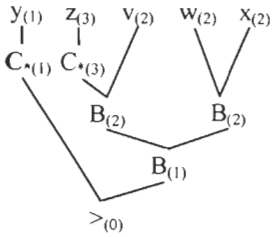


The next two **B** combinators will be introduced at levels 0 and 1, because their calculated introduction levels are both 1.





Lastly, the **C*** combinator will be added at the root (level 0), because just like the other **C*** combinator, its argument were at level 0 before the combinator was introduced.



The final result is the same combinatory expression we had at start.

3. Conclusion

The algorithm we presented here is very helpful. To reduce combinatory expression with combinators in an expression without combinators is a process that can be easily implemented. The choice of reducing combinators from left to right excludes any kind of ambiguity. To introduce combinators in a normal form without combinators, is not easy. The order in which the combinators are to be introduced is important. We, also, must identify their arguments.

Our algorithm is used in the case of structural reorganization. We believe it could be useful in the case of how to prove that two sentences are in fact paraphrases.

References

Beavers, J., 2004. Type-inheritance Combinatory Categorial Grammar. *In Proceedings of COLING 2004.* Geneva, Switzerland.

Beavers, J., Sag., I.A., 2004. Some Arguments for Coordinate Ellipsis in HPSG. *In Proceedings of the 2004 HPSG Conference. Katholike Universiteit Lueven, Belgium*

Biskri, I., Begin, C., 2004, "The analysis of the relative, completive and indirect interrogative subordinate constructions in French by means of the ACCG", Proceedings of FLAIRS 2004, Floride 2004, AAAI press.

Biskri, I., Desclés, J.P., 1997, Applicative and Combinatory Categorial Grammar (from syntax to functional semantics). *In Recent Advances in Natural Language Processing,* 71-84. John Benjamins Publishing Company.

Brun, C., 1999. Coordination et analyse du français écrit dans le cadre de la grammaire lexicale fonctionnelle. *In Revue électronique les enjeux de l'information et de la communication.*

Curry, B. H., Feys, R., 1958. *Combinatory logic* , Vol. I, North-Holland.

Desclés, J.P., 1996. Cognitive and Applicative Grammar: an Overview. *in C. Martin Vide, ed. Lenguajes Naturales y Lenguajes Formales, XII*, Universitat Rovra i Virgili. , 29-60.

Desclés, J. P, 1990. *Langages applicatifs, langues naturelles et cognition*, Hermes, Paris.

Dowty, D., 2000, The Dual Analysis of Adjuncts/Complements in Categorial Grammar. *In Linguistics 17.*

Hendriks, P., 2003. Coordination. In P. Strazny (ed.), Encyclopedia of Linguistics, Fitzroy Dearborn, New York.

Milward, D., 1994, Non-Constituent Coordination: Theory and Practice. *In Proceedings of COLING 94* , Kyoto, Japan, 935-941.

Moorgat, M., 1997. Categorial Type Logics. *In Johan Van Benthem and Alice Ter Meulen eds.*, Handbook of Logic and Language, 93-177. Amsterdam: North Holland.

Morrill, G., 1994, *Type-Logical Grammar.* Dordrecht: Kluwer.

Sag, I.A., 2003. Coordination and Underspecification. *In Procedings of the 2003 HPSG Conference.* CSLI Publications. 267-291.

Shaumyan, S. K., 1998, Two Paradigms Of Linguistics: The Semiotic Versus Non-Semiotic Paradigm. *In Web Journal of Formal, Computational and Cognitive Linguistics.*

Steedman, M., 2000. *The Syntactic Process*, MIT Press/Bradford Books.

Combinators' Introduction: An Enhanced Algorithm

Adam Joly, Ismail Biskri and Boucif Amar Bensaber
Département de Mathématiques et Informatique, Université du Québec à Trois-Rivières
C.P. 500, Trois-Rivières (QC) G9A 5H7, Canada
{adam.joly; ismail.biskri; bensaber}@uqtr.ca

Abstract

Strategies for removal and introduction of combinators are very important to assure an accurate use of combinatory logic and combinators in natural language processing, especially in structural reorganization of expressions that express semantic interpretation. Such a strategy already exists for the elimination of combinators in a combinatory expression to obtain a normal form without combinators, but none existed to automate the inverse process. In our previous work, we addressed this problem by proposing an algorithm for the automation of combinators' introduction, which finds the introduction level and introduces it at the first available spot. However, this algorithm shows its limits. There are some specific cases where a combinator can be introduced at more than one place. We needed to improve our algorithm so that it can automatically find the exact path to take in order to reach the correct place where we have to introduce the combinator, and then the algorithm would work for any combinatory expression. This paper presents the enhanced algorithm with an example of its execution.

1. Applicative Combinatory Categorial Grammar

In this model, the language units are considered as operators or operands and they will be translated in formal logical expressions of the combinatory logic (Curry, 1958 ; Shaumyan, 1998). The linguistics models of the Universal Applicative Grammar (Shaumyan, 1998) and from its extension the Applicative and Cognitive Grammar (Desclés, 1990, 1996) insist very explicitly on the relevance of the levels of representation of languages with interaction between these levels. Their main objective is the study of the formal properties of linguistic systems and an analysis of their logical structures. (i) *The level of the morpho-syntactical structures*, where the specific features of the language are expressed (for instance, the word order, morphological cases, etc.). It is the observable representation of the language; (ii) *The level of the predicative structures*, where the grammatical invariants and the predicative structures inherent to the phenotypic statements are expressed. This level makes it possible to express the functional semantic interpretation of the statements in which each linguistic unit is an operator followed by its operands; (iii) *The cognitive level*, where the meaning of the linguistic predicates is given (Desclés, 1997). It is in this general framework that the model of ACCG is set (Biskri, Desclés, 1997 ; 2006). This model, like other categorial model (Steedman, 2000 ; Morill, 1994 ; Dowty, 2000 ; Moorgat, 1997 ; Baldridge & Kruijf , 2003), falls under a paradigm of language analysis that allows a complete abstraction of the grammatical structure from its linear representation, and of the grammar from its lexicon. As well as the verification of the correct syntactic connection of the statements, it allows the explicit connection if the morpho-syntactical expression to its predicative representation by the means of combinators from the combinatory logic at which introduction and removal (β-reduction) rules are associated. For illustration, we present the β-reduction rules of **Φ**, **B** and **C***[1] (U1, U2, U3, U4 being typed applicative expressions):

(**Φ** U1 U2 U3) U4 → U1(U2 U4) (U3 U4)
((**B** U1 U2) U3) → (U1 (U2 U3))
((**C*** U1) U2) → (U2 U1)

In the ACCG model, the premise of each rule is a concatenation of linguistic units with oriented types. The consequence of each rule is an applicative typed expression with the possible introduction of one combinator. The type-raising of one unit u introduces the combinator **C***; the composition of two concatenated units introduces the combinator **B**. Let us show some rules used in this paper.

Application rules :

[X/Y : u1] - [Y : u2]	[Y : u1] - [X\Y : u2]
----------->;	-----------<
[X : (u1 u2)]	[X : (u2 u1)]

Type raising rules :

[X : u]	[X : u]
----------->**T** ;	-----------<**T**
[Y/(Y\X) : (**C*** u)]	[Y\(Y/X) : (**C*** u)]

Functional composition rules :

[X/Y : u1]-[Y/Z : u2]	[Y\Z : u1]-[X\Y : u2]
----------->**B** ;	-----------<**B**
[X/Z : (**B** u1 u2)]	[X\Z : (**B** u2 u1)]

[1] Here we are only interested in combinators used in this paper.

The combinatory logic is without variable. It enables the elimination of telescoping. In addition, it allows to systematize an intelligent structural reorganization of the constituents, cheaper in terms of computational resources, due to the fact that the combinators in combinatory expressions transport an intrinsic semantic content which plays the role of a memory during the analysis. We must understand that the *structural reorganization* we are talking about is fundamentally different from the *decomposition* (Steedman, 2000). The *decomposition*, based on the principle of parametric neutrality, takes as indications the categorial types only, while the *structural reorganization* uses in addition the functional semantic interpretation. The principle of parametric neutrality is: *two categorial types linked by a combinatory rule (which only take into consideration the categorial types) allow the determination of the third type.* In other terms, for the following combinatory categorial rule:

A - B
-------->
C

If we know the categorial types A and C, we can infer B; or if we know the categorial types B and C, we can infer A; or at last if we know the categorial types A and B, we can infer C. Let us consider the following example :

Jean-	vit -	sa-vie -	dangereusement	
NP	(S\NP)/NP	NP	(S\NP)\(S\NP)	(1)
---->**T**				
S/(S\NP)				(2)
----------->**B**				
S/NP				(3)
----------				>
S				(4)
==========>**Decomposition**				
S/(S\NP)	S\NP			(5)
	----------			<
	S\NP			(6)
----------				>
S				(7)

The decomposition occurs at step 5. Two categorial types are known : the category S, obtained in (4), that corresponds to the expression *Jean-vit-sa-vie* and the category S\NP, required to be combined with the categorial type (S\NP)\(S\NP) of *dangereusement.* From then on the rule (>) allows us to deduce the type S/(S\NP). In other terms, S, S\NP and S/(S\NP) are associated by the rule (>). The knowledge of two of those types allows the inference of the third type.

However, this method meets important limits: (i) the identification of the "required" category is not related to any clearly defined strategy; (ii) in the case of some elliptic forms of the coordination, the method turns out to be inoperative. Let us take the case of the sentence *Jean-vit-sa-vie-dangereusement-et-son-amitié-sincèrement.* The analysis will validate the expression *Jean-vit-sa-vie-dangereusement* of categorial type S before it meets the conjunction *et*. The second member of the coordination corresponds to the categorial type (S\NP)\((S\NP)/NP).

son-amitié-	*sincèrement*
--------	----------
NP	(S\NP)\(S\NP)
-----**<T**	
(S\NP)\((S\NP)/NP)	
----------**<B**	
(S\NP)\((S\NP)/NP)	

At this step, we have two categorial types: S, the obtained type, and (S\NP)\((S\NP)/NP), the required type. The decomposition of the expression *Jean-vit-sa-vie-dangereusement* to extract the first member of the coordination according to the principle discussed earlier, gives the categorial type S/((S\NP)\((S\NP)/NP)).

S/((S\NP)\((S\NP)/NP)) -	(S\NP)\((S\NP)/NP)
----------	>
S	

Although, this type does not correspond to any semantic interpretation. Now, we will present our structural reorganization, with the following analysis:

1 [N: *Jean*] - [(S\N)/N : *vit*]-[N : *sa-vie*] - [(S\N)\(S\N) : *dangereusement*]-[(X\X)/X : *et*]-[N : *son-amitié*]-[(S\N)\(S\N) : *sincèrement*]
2 [S/(S\N) : (**C*** *Jean*)]-[(S\N)/N : *vit*]-[N : *sa-vie*]-[(S\N)\(S\N) : *dangereusement*]-[(X\X)/X : *et*]-[N : *son-amitié*]-[(S\N)\(S\N) : *sincèrement*] (>**T**)
3 [S/N : (**B** (**C*** *Jean*) *vit*)] - [N : *sa-vie*] - [(S\N)\(S\N) : *dangereusement*] - [(X\X)/X : *et*] - [N : *son-amitié*] - [(S\N)\(S\N) : *sincèrement*] (>**B**)
4 [S : ((**B** (**C*** *Jean*) *vit*) *sa-vie*)] - [(S\N)\(S\N) : *dangereusement*] - [(X\X)/X : *et*] - [N : *son-amitié*] - [(S\N)\(S\N) : *sincèrement*] (>)
5 [S : ((**C*** *Jean*) (*vit sa-vie*))] - [(S\N)\(S\N) : *dangereusement*] - [(X\X)/X : *et*] - [N : *son-amitié*] - [(S\N)\(S\N) : *sincèrement*]
6 [S/(S\N) : (**C*** *Jean*)] – [S\N : (*vit sa-vie*)] - [(S\N)\(S\N) : *dangereusement*] - [(X\X)/X : *et*] - [N : *son-amitié*] - [(S\N)\(S\N) : *sincèrement*]
7 [S/(S\N) : (**C*** *Jean*)] – [S\N : (*dangereusement* (*vit sa-vie*))] - [(X\X)/X : *et*] - [N : *son-amitié*] - [(S\N)\(S\N) : *sincèrement*] (<)
8 [S:((**C*** *Jean*) (*dangereusement* (*vit sa-vie*))]-[(X\X)/X : *et*]-[N:*son-amitié*]-[(S\N)\(S\N):*sincèrement*] (>)
9 [S:((**C*** *Jean*) (*dangereusement* (*vit sa-vie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**C***son-amitié*)]-[(S\N)\(S\N):*sincèrement*] (<**T**)
10 [S:((**C***Jean*) (*dangereusement* (*vit sa-vie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**B** *sincèrement* (**C*** *son-amitié*))] (<**B**)
11 [S/(S\N):(**C*** *Jean*)]-[S\N:(*dangereusement* (*vit sa-vie*))]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N):(**B** *sincèrement* (**C*** *son-amitié*))]
12 [S/(S\N):(**C*** *Jean*)]-[S\N:((**B** *dangereusement* (**C*** *sa-vie*)) *vit*)]-[(X\X)/X : *et*]-[(S\N)\((S\N)/N) : (**B** *sincèrement* (**C*** *son-amitié*))]
13 [S/(S\N) : (**C***Jean*)]-[(S\N)/N:*vit*]-[(S\N)\((S\N)/N) : (**B** *dangereusement* (**C*** *sa-vie*))]-[(X\X)/X : *et*]-(S\N)\((S\N)/N) : (**B** *sincèrement* (**C*** *son-amitié*))]
14 [S/(S\N) : (**C***Jean*)]-[(S\N)/N:*vit*]-[(S\N)\((S\N)/N) : (**B** *dangereusement* (**C*** *sa-vie*))]-[((S\N)\((S\N)/N))\((S\N)\((S\N)/N)) : (*et* (**B** *sincèrement* (**C*** *son-amitié*)))] (>)
15 [S/(S\N) : (**C***Jean*)]-[(S\N)/N:*vit*]-[(S\N)\((S\N)/N) : ((*et* (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*)))] (<)

16 [S/(S\N) : (**C***Jean*)] - [(S\N) : (((*et* (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*))) *vit*)] (<)
17 [S : ((**C***Jean*) (((*et* (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*))) *vit*))] (>)

18 ((**C***Jean*) (((*et* (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*))) *vit*))
19 ((((*et* (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*))) *vit*) *Jean*) **C***
20 ((((**Φ** ∧ (**B** *sincèrement* (**C*** *son-amitié*))) (**B** *dangereusement* (**C*** *sa-vie*))) *vit*) *Jean*) (et = **Φ** ∧)
21 ((∧ ((**B** *sincèrement* (**C*** *son-amitié*)) *vit*) ((**B** *dangereusement* (**C*** *sa-vie*)) *vit*)) *Jean*) **Φ**
22 ((∧ (*dangereusement* ((**C*** *sa-vie*) *vit*)) ((**B** *sincèrement* (**C*** *son-amitié*)) *vit*)) *Jean*) **B**
23 ((∧ (*dangereusement* (*vit sa-vie*)) ((**B** *sincèrement* (**C*** *son-amitié*)) *vit*)) *Jean*) **C***
24 ((∧ (*dangereusement* (*vit sa-vie*)) (*sincèrement* ((**C*** *son-amitié*) *vit*))) *Jean*) **B**
25 ((∧ (*dangereusement* (*vit sa-vie*)) (*sincèrement* (*vit son-amitié*))) *Jean*) **C***

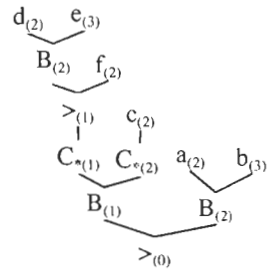
Steps 1 to 17 represent the application of ACCG rules. With these steps we verify the correctness of the sentence (the type S obtained at 17). Steps 18 to 25 are in the predicative level. They reduce combinators in order to construct the functional semantic interpretation (the normal form): ((∧ (*dangereusement* (*vit sa-vie*)) (*sincèrement* (*vit son-amitié*))) *Jean*), which is structured like a conjunctive clause. At the step 20 the linguistic predicate *et* (and) is replaced by its meaning in the cognitive level Φ in order to express the distributive and the conjunctive nature of et by respectively the combinator Φ and the logical connector .
A strategy of incremental analysis (from left to right) with an "intelligent" backtrack (Biskri & Desclès, 1997) supplements the model of the GCCA in order to solve the problem of the pseudo-ambiguity which consists in a multitude of syntactic derivations (which are from a certain point of view equivalent) for the analysis of the same statement and which corresponds to the same semantic interpretation. However, this strategy leads to the construction of false constituents that require structural reorganization (intelligent back track) (Biskri & Desclès, 2006; 1997) to bring out the right constituents. The structural reorganization uses semantic interpretation, in addition to a calculation on the categories, in the process of back tracking. This gives it more linguistic credibility, in addition to computational one. The principle is simple: the functional semantic interpretation is constructed using combinators introduced in the syntagmatic structure. When a false constituent is obtained, it is necessary to reduce a combinator (with using its b-reduction rule) and to test if the right constituent emerges (see steps 5, 6 for the first structural reorganization). The process is repeated until the right constituent emerges or there is no combinator to reduce (Biskri, Desclés, 1997). In the first version of the ACCG, this process was complemented by some rules (a, b, c, d) of combinators' introduction in the case of non-constituents coordination analysis (see steps 11, 12, 13). The analysis, based on the combinatory structure of the second member of the coordination must extract a combinatory structure for the first member of the coordination similar to the one of the second member. We apply at step 12 to (*dangereusement* (*vit sa-vie*)) the rule "(u1 (u2 u3)) <==> ((**B** u1 (**C*** u3)) u2) " in order to get ((**B** *dangereusement* (**C*** *sa-vie*)) *vit*) in which (**B** *dangereusement* (**C*** *sa-vie*)) is the first member of the coordination.

This rule is static. What about if other cases appear? We need an algorithm that completely automates the process of introducing combinators.

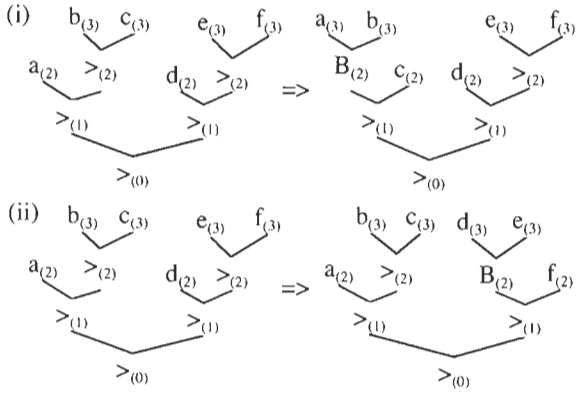
2. The Limits of the Algorithm for Combinators' Introduction

In our previous papers, we presented an algorithm that solves a big part of the problem (Joly & Biskri, 2008). We showed that we can calculate exactly at which level (from a binary tree representation of the combinatory expression) a combinator should be introduced. Once the algorithm has been run and we have every combinator's introduction level, the final step is to introduce them one by one, in the reverse order that they appear in the combinatory expression. We arbitrary chose to search from left to right for the first "available" place where the combinator can be introduced. In most of the cases, this works because there is only one possibility, but there is no guaranty that the combinator will be introduced at the right place when there are many possibilities. Likewise, we would have the same situation if the combinators were introduced by searching in the tree from left to right.

To illustrate the limits of the approach with a concrete example, let us take the following combinatory expression: ((B (C_* ((B d e) f)) (C_* c)) (B a b)). If we execute the algorithm, we will have this result:



Now, we have to introduce the combinators into the normal form. The introduction order will be $B_{(2)}$ $C_{*(2)}$ $B_{(2)}$ $C_{*(1)}$ $B_{(1)}$. We must remember that the introduction level represents the level where the combinator will be **after** its introduction. Let us begin with the first combinator. The B combinator can be introduced at two places, resulting in two different combinatory expressions: (i) or (ii).



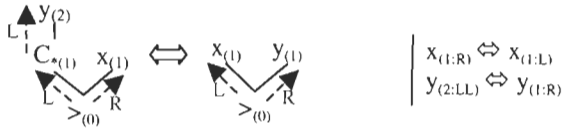
Because our initial strategy is to search from right to left, the B combinator would be introduced like (ii). This is wrong. In these situations, the resulting tree will not be the correct one or, worse, at some point, we will not be able to insert the next combinator anywhere. That is exactly what would happen in our example.

The reader must understand that the algorithm **does** find the correct introduction level, but it does not give any specific directions on the path to follow in the tree. Moreover, if we are blocked because a combinator has not been introduced where it should has been, we cannot simply retry the process by searching the other way around, because the path can be complex and need more than a single direction.

We needed a way to resolve this, so the algorithm will always find automatically the exact place where a combinator has to be introduced.

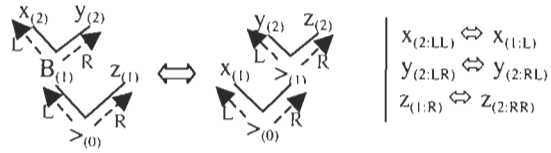
3. An Enhanced Algorithm

We realized that not only the level, but also the path to each combinator after its introduction can be deduced from a given combinatory expression. We will look back the reduction of combinators C_*, then B, and we will pay attention to the changes on paths to reach the nodes. To represent directions, we will use dotted arrows, along with "L" for left (the first argument of an operator) and "R" for right (the second argument). We will add the path to the node besides the node level for better visualization. First of all, we will analyze the combinator C_*. Its reduction and introduction go as following: $((C_* \; y) \; x) \Leftrightarrow (x \; y)$.



We can observe in the previous schema a variation not only in terms of levels, but also in terms of directions. More specifically, after the removal, "x" switched from right to left. Then, in order to reach "y", we had to go to the left two times, but once C_* is removed, the node loses one level and is directly to the right.

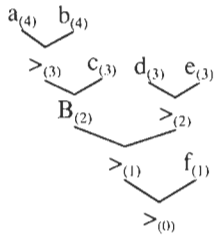
The combinator B is removed or introduced according to $((B \; x \; y) \; z) \Leftrightarrow (x \; (y \; z))$.



The removal of combinator B also has impacts on all arguments. Originally, "x" is to the left of B, which is itself to the left of the root. After the removal, one level is deducted and the path that was "left, left" becomes only "left". In the case of "y", its level does not vary, but both directions change: "left" goes "right" and vice versa. Finally, "z" gains a level and we must add the "right" direction.

These relations play a capital role, because they are the key to determine the path to a combinator after its introduction, as well as being used for the introduction itself.

To calculate the combinators' paths, it is very important to apply the impacts of the combinators, in the same order we remove them when we want to reach the normal form. We will have to take precautions to modify the right directions of the arguments of the combinators. For instance, let us have a look at the following expression:



In this expression, (a b) is the first argument of B, c the second and (d e), the third. The initial paths to b, c, and d are, in order, (LLLR), (LLR) and (LRL). After the removal, they will change for (LLR), (LRL) and (LRRL).

About the introduction, we realized that, in the last paper, we did not give much detail on how we introduce the combinators once we have found their introduction levels and what we meant by "inserting the combinator at the first available place". The pattern for the removal or introduction of a combinator always has a forward application as a root. When we will eliminate or insert a combinator, we will search for this point of reference that we will call the "**functional delimiter**". As we can see in both schemas, the path we will retrieve for a C_* or a B combinator should always end with a left direction beyond it. As a consequence, when introducing, we will need to ignore the last direction when we will follow to path in order to stop at the functional delimiter. It is important to note that not every combinator is the direct left argument of the delimiter (the ϕ combinator, for instance): we must adjust the strategy according to the combinator we are dealing with.

The strategy for the path finding can be translated into the following algorithm, considering that the combinatory

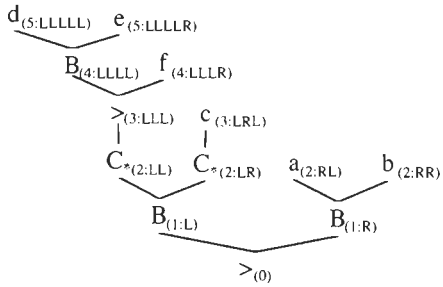
expression is organized in a binary tree's structure where each node has an initial path, and the only input of the method is a node (beginning with the root).

method calculatePath
if the current node is a combinator B then
replace the path "LL" by "L" for the first argument's nodes
replace the path "LR" by "RL" for the second argument's nodes
find the third argument's nodes, then change the path "R" by "RR"
else if the current node is a combinator C *then*
replace the path "LL" by "R" for the first argument's nodes
find the second argument's nodes, then change the path "R" by "L"
end if
for each child node, from the left to the right
call calculatePath for this node
end for
end of method

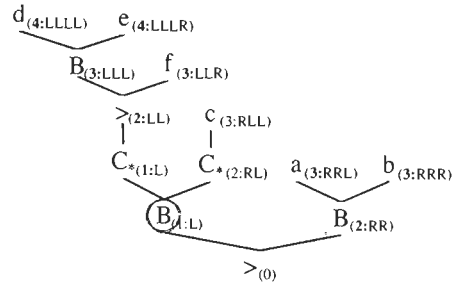
First of all, the first and second arguments of B and the first argument of C* are their child nodes, so the propagation of the directions to be modified is easily done. However, the third argument of B and the second argument of C* are not necessarily at the same position than we saw in the pattern, because of the different possible combinations of the combinators together. Though, we still can retrieve those nodes. What we need to do is to scan the whole tree (except for the sub nodes of the current combinator) in search of nodes with paths containing the path to the functional delimiter of the combinator, plus the path to the argument. For example, if the path to C* is (LRL), the functional delimiter is at (LR), so the nodes of the second argument are those with a path beginning by (LRR).

In the implementation of the algorithm, we do not need to store an introduction level, since it corresponds to the number of directions of a node's path. If the node is a forward application, no modification has to be done, and the method will be recursively called for its left, then right node. If it is a leaf (a lexical unit), it does not have any children, so there will be no more recursion from this node.

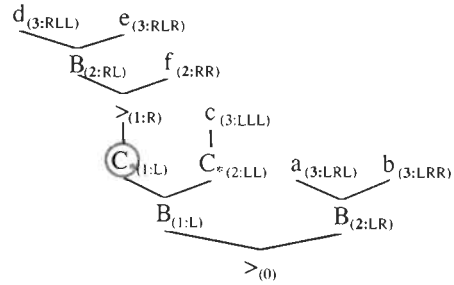
Let us take back the example we used in the previous section, which was the combinatory expression ((B (C* ((B d e) f)) (C* c)) (B a b)), as below, and follow the execution of the algorithm:



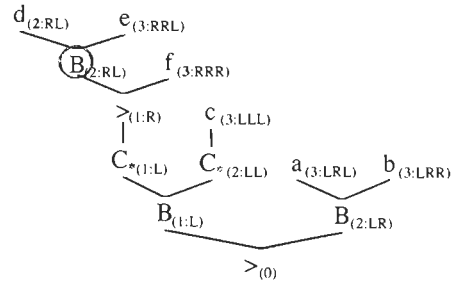
The root is the first node to be processed. According to the algorithm, we call recursively the method for the left node first. This brings us to the combinator $B_{(1:L)}$. The first argument of B is (C* ((B d e) f)). We have to replace "LL" by "L" for each node. The second argument is (C* c), and "LR" becomes "RL". At last, (B a b) corresponds to the third argument, because we search for nodes with paths beginning by "R". As a consequence, "R" turns into "RR".



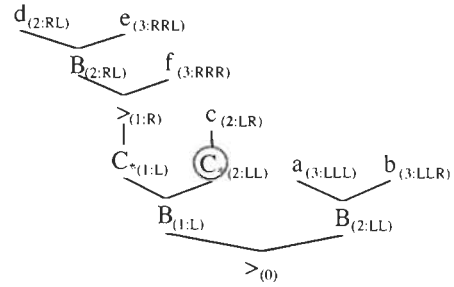
Then, the algorithm is called for $C_{*(1:L)}$. The first argument of C* is ((B d e) f), so "LL" changes for "R". For the second argument, we look for nodes with path starting by "R" and switch the direction to "L". The targeted nodes are (C* c) and (B a b).



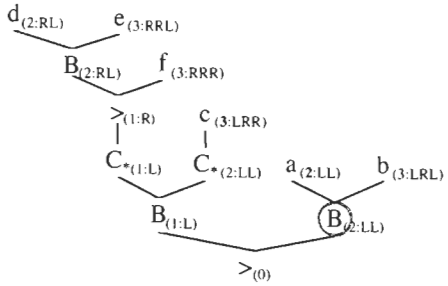
The three arguments of the next combinator, $B_{(2:RL)}$, are d, e, and f, and the directions are changed consequently.



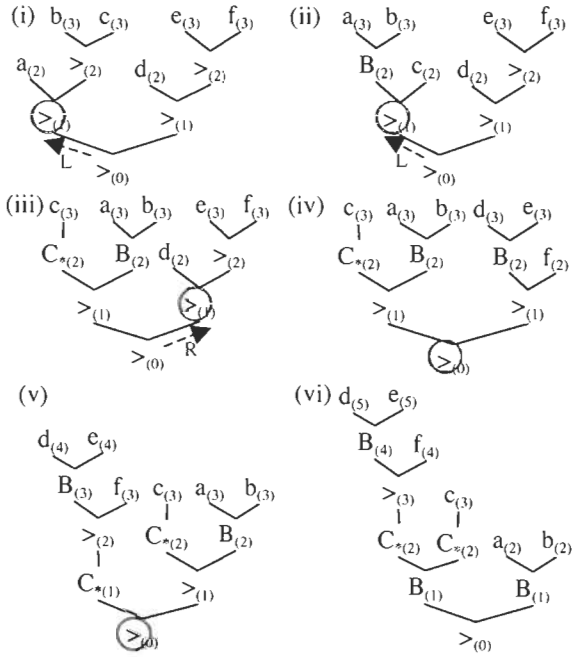
We have to go back to the right node of the first combinator B in order to reach another combinator. The first argument of $C_{*(2:LL)}$ is c, and its second argument is (B a b), which are the nodes beginning by "LR". This "R" becomes "L".



Finally, we cross $B_{(2:LL)}$. a and b are the first and second argument. The last argument corresponds to nodes with a path starting by "LR". This is the case of c.



There is no combinator left, so the process will eventually terminate after node "b". Now that the path to each combinator has been calculated, we are ready to introduce the combinators in the reverse order they are removed, that is to say $B_{(2:LL)}$ $C_{*(2:LL)}$ $B_{(2:RL)}$ $C_{*(1:L)}$ $B_{(1:L)}$. The introduction process will be simple, because we will only need to find the functional delimiter with the method we discussed earlier. Here are the introduction steps:



At step (i), we have the normal form. The first combinator to be introduced is $B_{(2:LL)}$. The position of its functional delimiter is the first argument of the root, because we have to ignore the last "left" direction. Using the pattern "(x (y z)) → ((B x y) z)", we introduce the combinator B. We can see at step (ii) that the B we inserted is located at (2:LL), just as we wanted. The next combinator is $C_{*(2:LL)}$. Again, we ignore the last direction and the delimiter is the left argument of the root. Then, we introduce C_* according to "(x y) → ((C* y) x)". At step (iii), the functional delimiter of the third combinator, $B_{(2:RL)}$, is situated to the right, from the root. Steps (iv) and (v) shows the introduction of $C_{*(1:L)}$ and $B_{(1:L)}$, for which the functional delimiter is directly the root. At last, after step (vi), we finally get the same combinatory expression we had at the beginning.

A great advantage of this strategy is that if we ever want to work with other combinators, we will only need to take in consideration the structure of the combinator alone when it is removed or introduced in a combinatory expression, and not how it behaves when it is along each other combinator. Moreover, we only need to stock the calculated path to the nodes.

4. Conclusion

The enhanced algorithm we presented considers the important concept of direction. This method assures us that we find the correct path to reach each combinator after its introduction. It is important that we search for combinators from left to right in the tree in order to impact the path of their arguments correctly. Afterward, it is easy to introduce it at the expression perspective.

In this paper, we used the algorithm for structural reorganization. Besides, more work still need to be done in order to handle the complete set of combinators and the algorithm will eventually be implemented for further testing.

References

Baldridge, J., and Kruijff, G.J., 2003. Multi-Modal Combinatory Categorial Grammar. *In Proceedings of EACL 2003.*

Biskri, I., Desclés, J.P., 2006. Coordination de categories différentes en français. *In Faits De Langue. No 28, (eds) Isabelle Bril & Georges Rebuschi. Editions OPHRYS, Paris.*

Biskri, I., and Desclés, J.P., 1997. Applicative and Combinatory Categorial Grammar (from syntax to functional semantics). In *Recent Advances in Natural Language Processing*, John Benjamins Publishing Company, 71-84.

Curry, B. H., and Feys, R., 1958. *Combinatory logic*, Vol. I, North-Holland.

Desclés, J.P., 1996. Cognitive and Applicative Grammar: an Overview. In *C. Martin Vide, ed. Lenguajes Naturales y Lenguajes Formales, XII*, Universitat Rovra i Virgili, 29-60.

Desclés, J. P, 1990. *Langages applicatifs, langues naturelles et cognition*, Hermes, Paris.

Dowty, D., 2000. The Dual Analysis of Adjuncts/Complements in Categorial Grammar. In *Linguistics 17.*

Joly, A., and Biskri, I., 2008, Combinators Introduction: An Algorithm, In *Proceedings of FLAIRS 2008*, Florida 2008, AAAI Press, 476-481

Moorgat, M., 1997. Categorial Type Logics. In *Johan Van Benthem and Alice Ter Meulen eds., Handbook of Logic and Language*, Amsterdam: North Holland, 93-177.

Morrill, G., 1994, *Type-Logical Grammar*. Dordrecht: Kluwer.

Sag, I.A., 2003. Coordination and Underspecification. In *Proceedings of the 2003 HPSG Conference*, CSLI Publications, 267-291.

Shaumyan, S. K., 1998. Two Paradigms Of Linguistics: The Semiotic Versus Non-Semiotic Paradigm. In *Web Journal of Formal, Computational and Cognitive Linguistics.*

Steedman, M., 2000. *The Syntactic Process*. MIT Press/Bradford Books.

BIBLIOGRAPHIE

Ajdukiewicz, K. (1935). Die Syntaktishe Konnexitat. *Studia philosophica* (Vol. 1), 1-27.

Anoun, H. (2006). Towards a Logical Analysis of Nominal Sentences in Standard Arabic. *Europeen Summer School in Logic, Language and Information 2006.*

Baldridge, J., Kruijff, G.J. (2003). Multi-Modal Combinatory Categorial Grammar. *European Chapter of the Association for Computational Linguistics 2003.*

Bar-Hillel, Y. (1953). A Quasi-arithmetical Notation for Syntactic Description. *Language* (29), 47-58.

Barry, G., Pickering, M. (1990). Dependancy and Constituency in Categorial Grammar. *L'Ordre des Mots dans les Grammaires Catégorielles*, 38-57.

Beavers, J. (2004). Type-inheritance Combinatory Categorial Grammar. Proceedings of the 20th International Conference on Data Engineering.

Biskri I., Bensaber, B.A. (2008). The Categorial Annotation of Coordination in Arabic. *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, 462-467.

Biskri I., Desclés, J.P. (2006). Coordination de Catégories Différentes en Français. *Faits de Langue* No 28.

Bozsahin, C. (2002). The Combinatory Morphemic Lexicon. *Computational Linguistics*, 28(2), 145-186.

Curry, B.H., Feys, R. (1958). *Combinatory Logic* (Vol. 1). North-Holland.

De Groote, P., Pogodalla, S. (2004). On the Expressive Power of Abstract Categorial Grammars : Representing Context-Free Formalisms. *Journal of Logic, Language and Information (Vol.13-4)*, 421-438.

Desclés, J.P. (1990). *Langages Applicatifs, Langues Naturelles et Cognition.* Hermès, Paris.

Desclés, J.P. (1996). Cognitive and Applicative Grammar : an Overview. *C. Martin Vide, ed. Lenguajes Naturales y Lenguajes Formales* XII, 29-60.

Desclés, J.P., Biskri, I. (1996). Logique Combinatoire et Linguistique : Grammaire Catégorielle Combinatoire Applicative. *Mathématiques et sciences humaines* (No.132), 39-68.

Dowty, D. (2000). The Dual Analysis of Adjuncts/Complements in Categorial Grammar. *Linguistics 17*.

Haddock, E. K., Morill, G. (1987). *Working Papers in Cognitive Science (Vol.I): Categorial Grammar, Unification Grammar and Parsing*. Edinburgh University Press.

Husserl, E. (1913). *Logische Untersuchungen*. Max Niemeyer, Halle.

Joly, A., Biskri, I. (2008). Combinators Introduction : An Algorithm. *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, 476-481.

Joly, A., Biskri, I. (2009). Combinators' Introduction : An Enhanced Algorithm. *Proceedings of the Twenty-Second International Florida Artificial Intelligence Research Society Conference*, 495-500.

Kang J., Desclés, J.P. (2008). Categorial Grammars, Combinatory Logic and the Korean Language Processing. *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, 488-493.

Kubota, Y. (2007). Solving theMorpho-Syntactic Puzzle of the Japanese -te Form Complex Predicate: A Multi-Modal Combinatory Categorial Grammar Analysis. *Questions Empiriques et Formalisation en Syntaxe et Sémantique 7*, 283-306.

Lambek, J. (1958). The Mathematics of Sentence Structure. *American Mathematical Monthly* (65), 154-165.

Lambek, J. (1961). On the Calculus Syntactic Types. *Proceedings of Symposia in Applied Mathematics* (Vol. XII), 166-178.

Moorgat, M. (1997). Categorial Type Logics. *Handbook of Logic and Language*, 93-177.

Morril, G. (1994). *Type-Logical Grammar*. Dordrecht : Kluwer.

Pareschi, R., Steedman, M. (1987). A Lazy Way to Chart Parse with Categorial Grammars. *Proceedings of ACL'87*.

Shaumyan S.K. (1998). Two Paradigms of Linguistics : The Semiotic Versus Non-Semiotic Paradigm. *Web Journal of Formal, Computational and Cognitive Linguistics*.

Steedman, M. (1989). Constituency and Coordination in a Combinatory Grammar. *Alternative Conceptions of Phrase Structure*, 201-231.

Steedman, M. (2000). *The Syntactic Process*. MIT Press/Bradford Books.

Szabolcsi, A. (1987). On Combinatory Categorial Grammar. *Proceedings of the Symposium on Logic and Languages,* 151-162.