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):

$$(\mathbf{\Phi}\ U1\ U2\ U3)\ U4 \rightarrow U1(U2\ U4)\ (U3\ U4)$$
$$((\mathbf{B}\ U_1\ U_2)\ U3) \rightarrow (U_1\ (U2\ U_3))$$
$$((\mathbf{C}_*\ U_1)\ U_2) \rightarrow (U2\ U_1)$$

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**.

<u>Application rules :</u>

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

<u>Type raising rules :</u>

[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)]

<u>functional composition rules :</u>

[X/Y : u_1]-[Y/Z : u_2] ------------------------->**B** ; [X/Z : (**B** u_1 u_2)]	[Y\Z : u_1]-[X\Y : u_2] -------------------------<**B** [X\Z : (**B** u_2 u_1)]
[X/Y : u_1]-[Y\Z : u_2] --------------------------->**Bx**; [X\Z : (**B** u_1 u_2)]	[Y/Z : u_1]-[X\Y : u_2] -------------------------<**Bx** [X/Z : (**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.

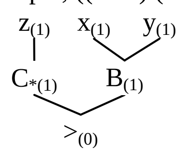
The principle is simple. The functional semantic interpretation is constructed using combinators introduced in the syntagmatic structure. When a false constituent is obtained, it is necessary to reduce a combinator (with using its b-reduction rule) and to test if the right constituent emerges (see steps 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 (t*endrement* (*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)

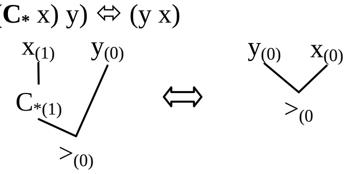
These rules are static. What about if other cases appear ? We need an algorithm that completely automates the process of introducing combinators.

2. Combinators Introduction : an Algorithm

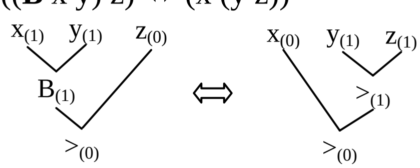
Every combinatory expression can be translated into a binary tree for convenience and visualization. For example, ((**C**$_*$ z) (**B** x y)) becomes:



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:

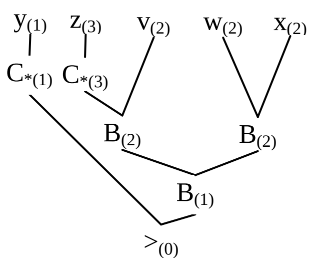
method calculateNodeLevel
*if the current node is the z argument of a **B** combinator (see the scheme) then*
add 1 to the level adjustment factor
end if
current node's calculated introduction level = initial current node level + level adjustment factor
if the current node is a forward application then
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.

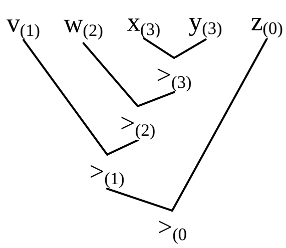
main method
- build the binary tree corresponding to the combinatory expression
- calculate nodes levels
- introduce one by one the combinators in the normal form in the reverse order they appear in the combinatory expression
end of method

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,

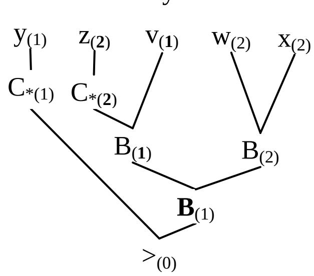
for simplification, they will be applied immediately for every child nodes, instead of waiting to reach each node and add the overall level adjustment factor.
The combinatory expression we will take as an example is (($\mathbf{C}_*$ y) (**B** (**B** ($\mathbf{C}_*$ z) v) (**B** w x))). Below, we have the resulting binary tree of the expression:



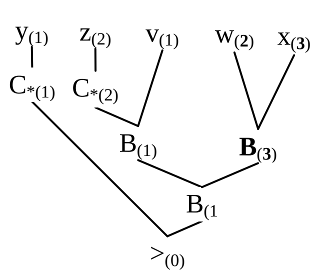
The normal form of the expression, after the ß-reduction, is ((v (w (x y))) z) and can be represented by the following tree:



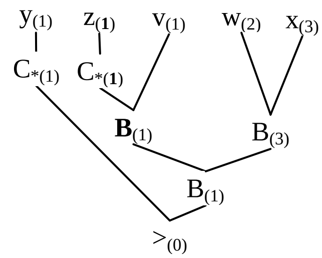
The next step consists to calculate the introduction combinators levels. In respect with the recursive algorithm, the nodes path will be >, **B**, **B**, x, w, **B**, v, $\mathbf{C}_*$, z, $\mathbf{C}_*$ and y.
First, from the root, we reach the **B** node at the right. We will have to clear the next 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.



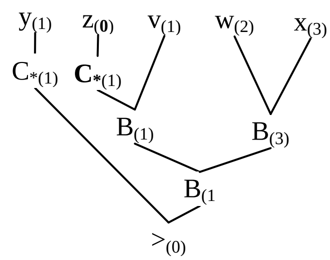
Again, we have a **B** combinator and the same rule applies. So, w level will eventually be reduced by 1. In addition, this **B** node has a sidling **B** node to the left, which means we have to add 1 level to the **B**, w and x nodes.



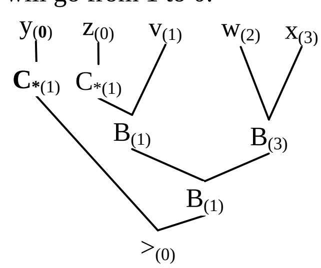
After the x and w nodes, we reach the last **B** node. Once more, the $\mathbf{C}_*$ and z nodes will lose one level.



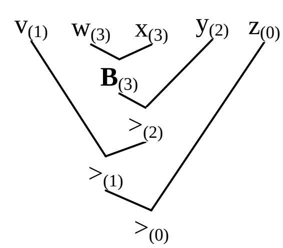
The next node will be v, then the $\mathbf{C}_*$ combinator. As a consequence, z level will be decreased again by one.



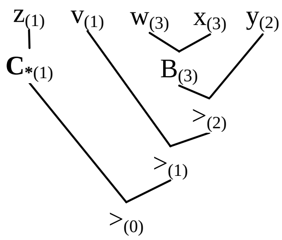
Finally, after y and back to the root, we take the left path and reach the last combinator. By the same logic, y level will go from 1 to 0.



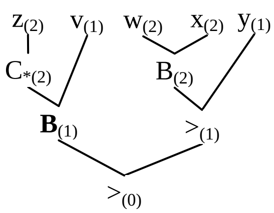
Now that we have calculated levels of the combinators, we can introduce them, as we said, from the right to the left, at their arguments levels.
The top right side combinatory will be the first to be introduced. Because its calculated level is 3, it means that before introducing it, its first argument was at level 2 and the two others were at level 3.

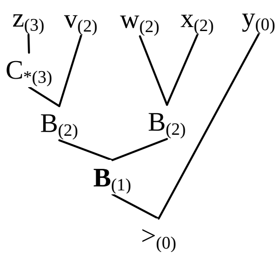


The second node to be introduced will be the $\mathbf{C}_*$ combinatory with the z argument. Being at level 1, its argument is one level below before the $\mathbf{C}_*$ introduction (level 0).

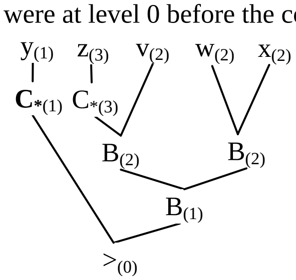


The next two **B** combinators will be introduced at levels 0 and 1, because their calculated introduction levels are both 1.





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.