Integration of Sequence of Computational Modules Dedicated to Text Analysis: A Combinatory Typed Approach

Ismaïl Biskri, Marie Anastacio, Adam Joly, Boucif Amar Bensaber

LAMIA, Département de Mathématiques et Informatique, Université du Québec à Trois-Rivières

C.P. 500, Trois-Rivières (QC) G9A 5H7, Canada

{ismail.biskri; marie.anastacio; adam.joly; boucif.amar.bensaber}@uqtr.ca

Abstract

In informational terms, a module dedicated to process information always has specific inputs and outputs. It describes a particular process constrained by specific rules. A processing chain can be a serial combination and/or a parallel combination of such modules. Thus, in an architecture of language engineering, each processing chain becomes a particular instantiation of all possible paths. A processing chain is built from a choice of modules underlying tasks that an engineer wants to apply to the text. In our paper we will present our theoretical model of logical representation of the processing chains, based on combinatory logic and a formal approach based on categorial grammars and applicative grammar, along with many cases of modules configurations.

Introduction

Language engineering, and by extension, information and knowledge engineering, has become a major and essential theme due to the critical need to assist human users to access information and extract knowledge from it.

Many methods, approaches and technologies have been proposed during the past four decades. Some exploit linguistic models whereas others are predicated on numerical and empirical approaches. Despite the high scientific value of these different models, they have yet to answer the various needs expressed by the scientific researcher community as well as the user community. The current and proposed technologies offer one or many from simple to complex functionalities, such as stemming, lemmatisation, classification, categorization, syntactic analysis, semantic analysis, morphological analysis, etc. However, these functionalities respond to very specific objectives and do not allow to adapt to initially unforeseen objectives gradually identified during the discovery process. This is, of course, a flexibility issue. Besides, the proposed technologies are often closed and thus cannot easily modify, replace or integrate new functionalities. Therefore, updating these tools requires a major computational investment. Consequently, the scientist researcher or the user feels unable to use them in a way that would allow him to integrate his own processing chains according to his own objectives.

Despite the high level of computational modeling offered by the programming paradigms such as object-oriented, and open source approaches, these limits remain persistent.

This kind of problems starts to find some echoes among scientists. It is in this way that an inclusive vision is being developed. Therefore we find in literature projects on : (i) the creation of complex processing chains (Hallab & al. 2000; Moscarola & al., 2002) that offers assembling of many functions and operations, and (ii) the creation of software platforms for language engineering which integrate statistical analysis, such as Aladin (Seffah & al, 1995), D2K/T2K (Downie & al., 2005) and Knime (Warr, 2007), or linguistic analysis, such as Context (Crispino & al, 1999) and Gate (Cunningham et Al., 2002)). These platforms are supposed to facilitates the fast prototyping of text mining and text analysis experiments.

From these new platforms emerge new interests on processing chains about their coherence, their flexibility, their adaptability, etc. Some of these platforms have been used in several projects in which researchers collaborate as NORA, TAPoR, etc.

Despite this progress, certain limits remain. In particular, the addition of new modules to the platform requires knowing the platform and the programming language used to implement it.

In our paper we will present our theoretical model of logical representation of the processing chains, based on combinatory logic and a formal approach based on categorial grammars and applicative grammar, along with many cases of modules configurations.

Combinatory Logic

Before introducing the formal model itself, let us first present combinatory logic. The origins of combinatory logic bring us back to the works of Schönfinkel who defined the notion of combinators in 1924, and also, sometime later, those of Curry and Feys (1958). This notion was introduced with the objective to bring a logical solution to some paradoxes, like the Russell's Paradox, but also to eliminate the need for variables in mathematics. Combinators are abstract operators that use other operators to build more complex operators. They act as functions over arguments, within an operator-operands structure. Each specific action is represented by a unique rule that defines the equivalence between a logical expression with a combinator versus one without a combinator, which is called a β-reduction rule. Although many more combinators exist, we show in the table opposite the combinators we used in our works and their corresponding β-reduction rule.

Combinator	Role	β-Reduction rule
B	Composition	**B** x y z → x (y z)
C	Permutation	**C** x z y → x y z
S	Distributive Composition	**S** x y u → x u (y u)
W	Duplication	**W** x y → x y y

The composition combinator **B** combines two operators x and y together in order to form the complex operator **B** x y that acts on an operand z according to the β-reduction rule. The permutation combinator **C** uses an operator x in order to build the complex operator **C** x such as if x acts on the operands y and z, **C** x will act on those operands in the reverse order, that is to say z and y. Given the two operators x and y, and the operand u, the general composition combinator **S** distributes the operand u with the two precedent operators x and y. (y u) becomes the operand of the complex operator (x u). Finally, given the binary operators x, and the operand y, the combinator **W** duplicates y so that the operator x will have two identical arguments.

We can also combine recursively many elementary combinators together to form an infinitely range of complex combinators. For example we could have combinatory expressions such as "**B C** x y z u" or "**S B C** x y z u v". Its global action is determined by the successive application of its elementary combinators, from left to right. If we have the combinatory expression "**B B C** x y z u v", the reduction order would be **B**, **B**, then **C**. The resulting expression without combinator is the normal form, which is, according to Church-Rosser, unique.

(i) **B B C** x y z u v
(ii) **B** (**C** x) y z u v
(iii) **C** x (y z) u v
(iv) x u (y z) v

There exist two more cases of complex combinators: combinators with "power combinators" and "distance combinators". In the first case, a power value of n reiterates n times the action of the combinator χ, such as $\chi^1 = \chi$ and $\chi^n = \mathbf{B}\ \chi\ \chi^{n-1}$. Thereby, the action of the expression $\mathbf{B}^2$ a b c d e would be **B B B** a b c d e → … → a (b c d) e.

In the latter case, an index value of n postpones the action of a combinator χ of n steps, such as $\chi_0 = \chi$ and $\chi_n = \mathbf{B}^{n-1}\ \chi$. If we consider the combinatory expression $\mathbf{C}_2$ a b c d e, the action of the complex combinator would be given by **B C** a b c d e → … → a b c e d.

The Formal Model

The main goal behind modular approaches is to reuse one or many already existing programs instead of having to write them from scratch again, which cost time and money, especially when the size of the programs are quite substantial.

Our model refers programs as modules and concerns systems for which the modules are processed serially only, that is, the so-called processing chains. We are particularly interested into natural language processing systems, for which it could be very useful to simply have to switch a module by another one with compatible inputs and outputs.

A module acts like a mathematic function that takes arguments, processes one specific action and gives a result. Each module is independent and can be seen like a black box: we are only interested to the general function it accomplished and not how it is programmed internally.

The modules must also have the capacity to communicate together with the help of a protocol.

A processing chain is a layout of modules. It is governed by three mains rules: (i) the chain must contain at least one module; (ii) the chain must be syntactically correct; (iii) the semantic aspects of the chain are the responsibility of the language engineer (we call language engineer any researcher or developer who has some interests into language engineering. The former can be a computer scientist as well as a linguist, a terminologist, a philosopher, etc.) to assure that the chosen modules serve the goals of the processing chain.

From a formal point of view, a processing chain is an integrated sequence of computational modules dedicated to specific processings, put together in a (pertinent) order according to a process goal determined by the language engineer. A processing chain will have to allow the composition of the modules. Therefore, it is essential to answer to two fundamental questions:

(1) Given a set of modules, what are the allowable arrangements which lead to coherent processing chains (the syntactic correctness)?

(2) Given a coherent processing chain, how can we automate (as much as possible) its assessment (in the sense of its calculability).

In order to do so, a formal system is needed. Such a system will be at the center of our theoretical model.

The model chosen is based on applicative and combinatory categorical grammar (Biskri & Desclés, 1997), a model we widely used in natural language processing.

Applicative and combinatory categorial grammar has won its spurs in syntactics and semantics. It proposes a dichotomous view on the linguistic units. Some of these linguistic units work as operators and others as operands. This is translated by an assignment of categories to the linguistic units in a way to reflect their nature. This view is, of course, applicative.

According to this view, a module accomplishes an operation which applies to one or many objectal entities from a given type and returns other objectal entities from another type. We therefore assign to each module a categorial type to reflect how it acts on its operands.

Categorial types are developed from basic types and from one constructive operators "F" as follow:

(i) Basic types are types.

(ii) If x and y are types then Fxy is a type.

We note a module (Figure 1) as follows: [M1: Fxy] in which M1 is the identifier of the module and Fxy is the type of M1. M1 is then considered as a function whose the operand is of type x and the result of the application of M1 on X is of type y. We note the module M2 (figure 2) by [M2: Fx_1Fx_2y]. M2 is a function with two operands: X1 and X2. M2 applies on X1 in order to construct a new function (M2 X1) whose operand is X2. The application of (M2 X1) on X2 gives Y. That is the meaning of the type Fx_1Fx_2y.

Figure 1: A graphical representation of a module with one input

Figure 2: A graphical representation of a module with two inputs

Within this approach, the processing chains become applicative "combinations" of typed functions. This view is in sum natural for computational modules given the fact that they are functions (in its general meaning, not the computational one) from the set of inputs to the set of outputs. Such combinations will be interpreted, like in some works in metaprogramming (Coquery, Fages, 2001), for the functional semantic interpretation of textual sentences (Steedman, 2000) or in artificial intelligence for scheduling issues, with the help of lambda-calculus (and unification) or using combinatory logic if we want to avoid a telescoping of variables (Curry, Feys, 1958; Hindley, Seldin, 2008). The interpretation of a processing chain will constitute the outcome of its underlying primitive operations and the way that these operations are organized accordingly to the principle of compositionality. The set of composed processing chains becomes a set of theorems for the proposed formal system. The system in itself is inferential. It proceeds by successive reductions of applicative categories assigned to operations concerned by the composition.

Let us, now, show the rules of our model[1] :

Applicative rule

[X : x] + [M1 : Fxy]

[Y : y]

Composition rule

(a)

[M1 : Fxy] + [M2 : Fyz]
---**B**
[(**B** M2 M1) : Fxz]

(b)

[M1 : FxFty] + [M2 : Fyz]
---**B^2**
[(**B^2** M2 M1) : FxFtz]

Distributive composition rule

[M1 : Fxy] + [M2 : FxFyz]
---**S**
[(**S** M2 M1) : Fxz]

Permutation rule

(a)

[M1 : FxFyz]
---**C**
[(**C** M1) : FyFxz]

(b)

[M1 : FxFyFtz]
---**C#**
[(**C** (**C_2** M1)) : FtFxFyz]

Duplication rule

[M1 : FxFxy]
---**W**
[(**W** M1) : Fxy]

The premises in each rule are typed "connected modules", and the results are typed applicative expressions (of modules) with an eventual introduction of one combinator. These applicative expressions allow the interpretation of the processing chains. Types of modules in the premises will allow us to validate the application of the rules, and therefore to accept or reject the connection of the modules. In other words, an inferential calculation on types will allow verifying the syntactic correctness of processing chains. Combinatory logic fills two major goals: (i) it gives an interoperable and formal representation of the solution and (ii) it gives the direct execution order of the modules which form the processing chain.

Within this formal system, in order to build a processing chain, we need specific data: (i) the list of the modules and (ii) the list of their inputs and outputs.

[1] Due to space limitations, we show here only the rules we use in this paper. In fact these rules suppose that modules have a maximum of two inputs.

Let us consider connection of two modules (Figure 3).

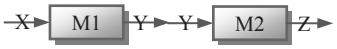

Figure 3: Accepted connection of two modules

The first module M1 is of type Fxy. M1 applies on the input X of type x in order to yield the output Y of type y. The second module M2 is of type Fyz. M2 applies on the input Y of type y in order to yield the output Z of type z. The graphical notation in Figure 3 will be expressed by the following expression: [M1 : Fxy] + [M2 : Fyz]. The first composition rule given above returns the complex module (**B** M2 M1) of type Fxz. In other words, the composition of M2 and M1 is possible, and the new module applies on one input of type x in order to yield one output of type y. In the case of the example given in Figure 4, graphical notation will be expressed by the following expression: [M1 : Fxy] + [M2 : Fza]. The first composition rule previously described does not allow the composition of M2 and M1 since the type z of the input of M2 given in the type Fza is not similar to the type y of the output of M1 given in the type Fxy. The connection of M1 and M2 is rejected.

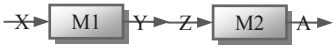

Figure 4: Rejected connection of two modules

Examples given in Figures 3 and 4 concern the connection of two modules. But what about if we have three modules (Figure 5)?

Figure 5: Accepted connection of three modules

The analysis begins by connecting modules M2 (with the type is Fyz) and M1 (with the type is Fxy). By using the first composition rule, the analysis yields the complex module (**B** M2 M1) whose type is Fxz. This module is then composed with M3. The resulting module is: (**B** M3 (**B** M2 M1)) whose type is Fxa (in other words the input of the obtained complex module in this case must be of type x whereas the output must be of type a).

Overall, when we have several modules connected in a linear chain processing, analysis iterates the application of the first composition rule to modules from left to right.

The first composition rule can also be used in the case of the example presented in Figure 6 in which two modules are connected to a third one. M1 is of type Fxy. M2 is of type Fza. M3 is of type FyFau.

Since M1 is of type Fxy and M3 of type FyFau, the first composition rule allows the construction of the complex module (**B** M3 M1) with the type FxFau. The processing chain given in Figure 6 will thus be equivalent to the processing chain given in Figure 7. In fact, the processing chain in Figure 7 corresponds to the following applicative expression: (**B** M3 M1) X (M2 Z), in which X is the first operand of the complex module (**B** M3 M1), and (M2 Z) the second. However, we need to have all inputs at the most right of our expressions. To do this, we apply the first permutation rule on the category [(**B** M3 M1) : FxFau]. It yields the equivalent category [(**C** (**B** M3 M1)) : FaFxu] that corresponds to the processing chain in Figure 8. We then carry on with the use of the first composition rule, given the types Fza for M2 and FaFxu for (**C** (**B** M3 M1)). We finally obtain the complex module (**B** (**C** (**B** M3 M1)) M2) whose type is FzFxu (Figure 9).

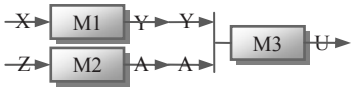

Figure 6: Two modules connected to a third module

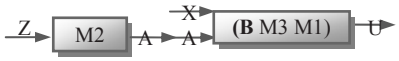

Figure 7: Module connected on the first input

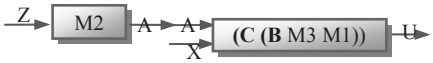

Figure 8: Permutation of the first input

Figure 9: Complex module with two inputs

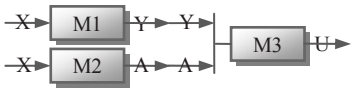

Figure 10: Two modules with the same input connected to a third module

Figure 11: Complex module with the same input repeated

Figure 12: Complex module with one input

We encounter the case of Figure 10 often in the domain of text mining. The same input is required for one or more modules whose outputs are used as inputs to another module. This processing chain is similar to the one given in Figure 6, even if the inputs of M1 and M2 are the same. The analysis will be also identical to the previous one applied to the processing chain described by Figure 6. The obtained complex module will be (**B** (**C** (**B** M3 M1)) M2) with the type FxFxu (Figure 11). For practical reasons, we must eliminate the duplication of input X. To do this, the application of the duplication rule to the category [(**B** (**C** (**B** M3 M1)) M2) : FxFxu] provides a module (**W** (**B** (**C** (**B** M3 M1)) M2)) that requires a single input (Figure 12) from a module that requires two inputs. The type of the new complex module is Fxu.

In the case of the processing chain given in Figure 13, the use of the distributive composition rule is crucial. Here

we have three parallel modules (M1, M2, and M3) connected to a fourth one (M4). M1 is of type Fxy, M2 of type Fxa, M3 of type Fxz, and M4 of type FyFaFzu. Since the types of M1 and M4 are respectively Fxy and FyFaFzu, the analyzer triggers the first composition rule. It yields the category [(**B** M4 M1): FxFaFzu]. At this stage, our processing chain takes the form shown in Figure 14. Since the obtained complex module (**B** M4 M1) is of type FxFaFzu and M2 of type Fxa, the analyzer applies the distributive composition rule. We obtain the complex module expressed by the category [(**S** (**B** M4 M1) M2) : FxFzu] (Figure 15). This last complex module is composed with M3 by using the distributive composition rule again since they respectively have types FxFzu and Fxz. The processing chain given in Figure 13 is then expressed by the complex module (**S** (**S** (**B** M4 M1) M2) M3) whose type is Fxu. The case of the processing chain given in Figure 13 can be generalized to more than three modules connected to a fourth one. It can be done by using iteratively the distributive composition rule.

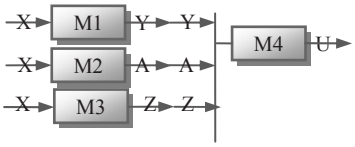

Figure 13: Three modules with the same input connected to a fourth module

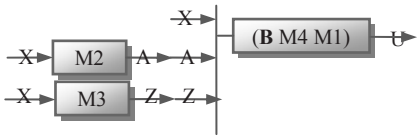

Figure 14: Two modules with the same input connected to a complex module

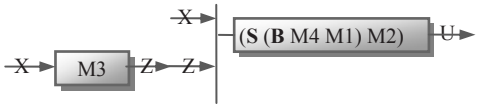

Figure 15: Complex module connected on the first input

Construction of a Processing Chain

We have presented some cases of basic processing chains and showed how to apply the rules of the formal model to verify the syntactic correctness and to construct their applicative representation, which will allow their interpretation and their execution. All cases, we have shown, have either arrangements of modules in series or parallel arrangements. A serial processing chain is composed of many modules connected together. When a processing chain contains at least one module with more than one input, we call it a parallel processing chain.

We tested many more particular arrangements of serials, parallels and output-distributed modules as well as very complex processing chains that we cannot show due to space limitation. However, we are willing to give the reader a glimpse of what an analysis based on a typed combinatory approach of a somewhat complex processing chain can look like. The processing chain given in Figure 16 is a combination of seven modules. M1 is of type Fx_1y_1. M2 is of type Fx_2z_1. M3 is of type Fy_1z_2. M4 is of type Fx_3y_2. M5 is of type $Fz_1Fz_2t_1$. M6 is of type Fy_2z_3. M7 is of type Ft_1Fz_3u.

The first step is to combine M3 and M1. Since M3 and M1 are respectively of type Fy_1z_2 and Fx_1y_1, the first composition rule is applied and the complex module (**B** M3 M1) is constructed. Its type is Fx_1z_2. The processing chain in Figure 16 is reduced to the one in Figure 17.

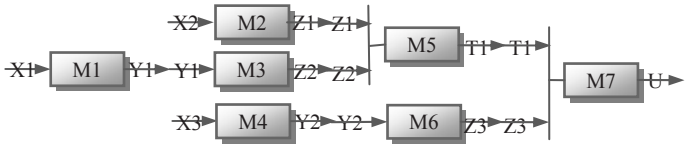

Figure 16: A complex processing chain

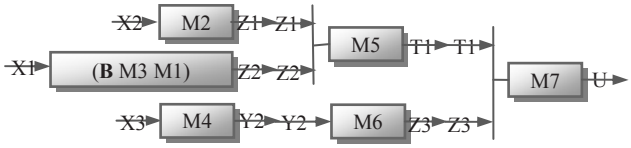

Figure 17: First step of the analysis

The second step is to combine M2, M5 and (**B** M3 M1). We will not show the details of this combination. Simply refer to the above analysis of the processing chain given in Figure 6. The analysis yields the complex module (**B** (**C** (**B** M5 M2)) (**B** M3M1)) whose type is $Fx_1Fx_2t_1$ (Figure 18).

The third step is to combine M4 and M6. Since M4 and M6 are of types respectively Fx_3y_2 and Fy_2z_3, the first composition rule is applied and the complex module (**B** M6 M4) is constructed. Its type is Fx_3z_3. The processing chain in Figure 18 is reduced to the one in Figure 19.

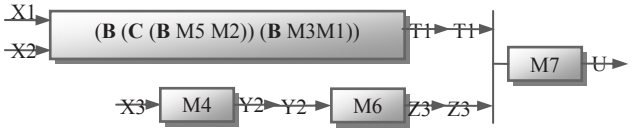

Figure 18: Second step of the analysis

Figure 19: Third step of the analysis

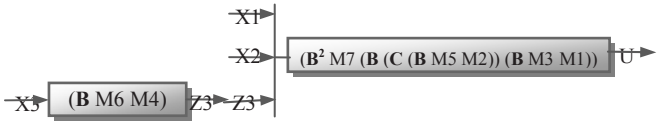

Figure 20: Fourth step of the analysis

The fourth step is the application of the second composition rule to M7 and (**B** (**C** (**B** M5 M2)) (**B** M3 M1)) since their respective types are Ft_1Fz_3u and $Fx_1Fx_2t_1$. We obtain the complex module ($\mathbf{B^2}$ M7 (**B** (**C** (**B** M5 M2)) (**B** M3 M1))) whose type is $Fx_1Fx_2Fz_3u$ (Figure 20).

At the fifth step the analysis applies the second permutation rule to ($\mathbf{B^2}$ M7 (**B** (**C** (**B** M5 M2)) (**B** M3 M1)). This operation yields the complex module (**C** ($\mathbf{C_2}$ ($\mathbf{B^2}$ M7 (**B** (**C** (**B** M5 M2)) (**B** M3 M1)))) whose type is $Fz_3Fx_1Fx_2u$ (Figure 21).

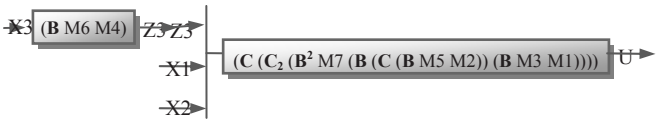

Figure 21: Fifth step of the analysis

The last step concerns the application of the first composition rule to complex modules (**C** ($\mathbf{C_2}$ ($\mathbf{B^2}$ M7 (**B** (**C** (**B** M5 M2)) (**B** M3 M1)))) and (**B** M6 M4) since their respective types are $Fz_3Fx_1Fx_2u$ and Fx_3z_3. It yields the complex module (**B** (**C** ($\mathbf{C_2}$ ($\mathbf{B^2}$ M7 (**B** (**C** (**B** M5 M2)) (**B** M3 M1)))) (**B** M6 M4)) whose type is $Fx_3Fx_1Fx_2u$ (Figure 22).

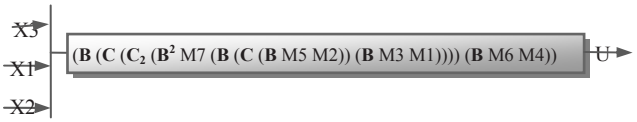

Figure 22 : Last step of the analysis

At this last step the processing chain given in Figure 16 is considered as syntactically correct. The complex module expressed in Figure 22 by means of combinators is the combinatory expression underlies this processing chain. The reduction of combinators by means of β-reduction rules gives the order in which each module can be running on its inputs.

Conclusion

The need for flexible, adaptable, consistent and easy-to-use tools and platforms in a recent and active field such language engineering is indisputable. Some projects with this philosophy in mind have seen the light in the last years. The model we propose has strong formal foundations (Applicative Grammars and combinatory logic). One of the principal advantages of our formalism is to assure a firm compositionality of the different modules in the different processing chains. Another but not least advantage is the possibility to compose an infinity of modules. We will not have the limits on the vocabulary that, for example, a traditional context free grammar or a regular grammar would impose.

A first prototype of this model named SATIM was implemented in C++.

References

Biskri, I., 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.

Coquery, E., Fages, F. 2001. "Programmes logiques avec contraintes typés", In proceedings of JFPLC 2001. Hermès. pp 223-238.

Crispino G., Ben Hazez S., Minel J.L. 1999. *"Architecture logicielle de Context ; plate-forme d'ingénierie linguistique"*, in proceedings of TALN 99.

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V. 2002. "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications". *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*. Philadelphia, July 2002.

Descles, J. P. 1990. *Langages applicatifs, langues naturelles et cognition*, Hermes, Paris.

Downie, J. S., Unsworth, J., Yu, B., Tcheng, D., Rockwell, G., and Ramsay, S. J., 2005, A revolutionary approach to humanities computing: tools development and the D2K datamining framework. In Proceedings of the 17th Joint International Conference of ACH/ALLC. 2005

Curry, B. H., Feys, R. 1958. *Combinatory logic* , Vol. I, North-Holland.

Hindley, J. R., Seldin, J. P. 2008. *Lambda-calculus and Combinators, an Introduction.* Cambridge University Press.

Seffah, A., Meunier, J.G. 1995. "ALADIN : Un atelier orienté objet pour l'analyse et la lecture de Textes assistée par ordinaleur". International Conferencence On Statistics and Texts. Rome 1995.

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.

Warr, A. W. 2007. *Integration, analysis and collaboration. An Update on Workflow and Pipelining in cheminformatics*. Strand Life Sciences.