# Toward a New Language Engineering

**Ismail Biskri[1], Jean-Guy Meunier[2], Adam Joly[1] and Marc-André Rochette[1]**

[1] LAMIA, Département de Mathématiques et Informatique, Université du Québec à Trois-Rivières
C.P. 500, Trois-Rivières (QC) G9A 5H7, Canada
{ismail.biskri; adam.joly; marc-andre.rochette}@uqtr.ca
[2] LANCI, Département de Philosophie, Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville, Montréal (QC) H3C 3P8, Canada
meunier.jean-guy@uqam.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 or a parallel combination of such modules. Thus, 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 tasks underlying modules that an engineer wants to apply to the text. Therefore, in this perspective, a fundamental question arises: given a set of modules, what are the eligible chains of all combinations of the given modules? This is what we will discuss about in our paper.

## 1. Introduction

Language engineering is a young and interdisciplinary science. This term that we could consider as a neologism covers everything related today to the natural language processing and more specifically to the knowledge extraction. Besides, the main goal of language engineering is to help humans to access to knowledge contained in texts. If we were to define language engineering, we would say that it corresponds to the study and the description of the concepts, the approaches, the methods and the techniques that allow data extraction and knowledge modeling and acquisition. This definition, even though closed to knowledge engineering (more focused on researches in artificial intelligence (but not limited to it)), is no less valid for language engineering which also includes the linguistic due to the observed connections between linguistic and knowledge engineering. It has been observed that knowledge acquisition from text was due to be assisted by analysis tools for corpus which can be syntactic or semantic analyzers, marker tracking tools supported by contextual exploration, statistical analyzers, etc. The application fields for language engineering are numerous and in constant evolution, more particularly since the development of communication networks (Web) and office tools that are able to support large quantity of contextual data. Therefore, throughout its development up to now, language engineering has gone through many steps or generations of tools, each of these generations being the consequence of a particular technical need. At the beginning, about 40 years ago, applications were often focusing on a single very specific functionality. Since the 90's, in consequence of more complex approaches required for text analysis and a will of technological transfer to the industry, we have noticed an interest for functions and operations assembling in complex processing chains. Most of the tools proposed at that time have been offering various functionalities. Despite some successes with industries as well as with scientists, many important limits has been identified: on the one hand the technologies offer a *closed and limited* set of functionalities and on the other they are designed as autonomous entities that are hardly integrable into even more complex processing chains. Moreover, a researcher having particular analysis objectives will find himself in the impossibility to use this type of technology, due to a lack of adaptability. Despite the high level of computational modeling offered by paradigms such as object-oriented programming, 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 the creation of software platforms for language engineering which integrate statistical analysis, such as Aladin (Seffah & al, 1995), T2k and Knime (Warr, 2007), or linguistic analysis, such as Context (Crispino & al, 1999) and Gate (Cunningham et Al., 2002)). From these new platforms emerge new interests on processing chains about their coherence, their flexibility, their adaptability, etc.

## 2. General Framework

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 (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.). 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.

Moreover, a processing chain made of modules will have to allow their composition. 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?

(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 chosen general theoretical framework is the Applicative Grammars (Desclés, 1990; Shaumyan, 1998). Applicative Grammars have won one's spurs in the syntactic, semantic and cognitive analysis of languages. They have a dichotomous view on the language units. Some of these linguistic units work as operators and other as operands. This is translated by an assignment of applicative categories to the linguistics units in a way to reflect their nature. The admissible (syntactically correct) sentences are those for which the combination of the categorical types assigned to units can be simplified into a base type. The same general model has been applied to other data encoding types, particularly iconic forms (Meunier, 1996). In addition, in our research, the processing chains become applicative "combinations" of typed functions. This vision 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 then 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.

One of the principal advantages of this 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.

In our paper we will present our theoretical model of logical representation of the processing chains, based on combinatory logic, along with many cases of modules configurations.

## 3. Combinatory Logic

The origins of the 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 present 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 |
| **Φ** | Distribution | **Φ** x y z u → x (y u) (z u) |
| **W** | Duplication | **W** x y → x y y |

The composition combinator **B** combines together two operators x and y 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 three operators x, y and z and the operand u, the distribution combinator **Φ** distributes the operand with the two precedent operators. Finally, given the binary operators x, and the operand y, the combinator **W** duplicates y so that the operator x will have its two 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 "**Φ 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**, such as:

(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

The resulting expression without combinator is called a normal form, which is, according to Church-rosser theorem, unique.

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 $\chi$, 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 last case, an index value of n postpones the action of a combinator $\chi$ of n steps, such as $\chi_0 = \chi$ and $\chi_n = \mathbf{B^n}\ \chi$. If we consider the combinatory expression "$\mathbf{C_2}$ a b c d e", the action of the complex combinator would be given by "$\mathbf{B^2}$ C a b c d e" → … → "a b c e d".

## 4. Construction of a Processing Chain

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 is a time and money saver, especially when the size of the programs are substantial.

The role of a metaprogram is then to act as a controller over the programs, by specifying the interactions between programs and their flow of execution.

Our model refers programs as modules and concerns systems for which the modules are processed serially only, so-called processing chains. Like we said, 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, process one specific action and gives an output as 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 done internally.

The modules must also have the capacity to communicate together with the help of a protocol and a controller must supervise the flow of communication. A module must provide data about itself (its name, its description and the list of its inputs and outputs) so the communication can be effective (possible).

A module can have none to many inputs, but can have none or only one output. In order to establish the communication between two modules, the controller must verify that the domain of the output of the first module must be the same that the one from the second module. The grammar we use accept four primitive types (character, string, integer and real number) and a list structure, from which we can create an infinity array of domains. A list can also be use to simulate many outputs within a single one that can be distributed to many other modules. For example, a valid domain could be Integer, List(Integer) or Character, Integer, Real, List(List(Integer), Real).

A processing chain is a layout of modules. It is governed by two mains rules:

(i) The chain must contain at least one module;
(ii) The chain is syntactically correct.

A single module is considered as a valid chain. "Syntactically correct" means that every output is connected to an input of the same domain. We do not take into consideration the semantic aspects of the chain: this is the responsibility of the language engineer to assure that the chosen modules serve the goals of the processing chain.


Figure 1: An example of a processing chain

A processing chain must be executed in a particular order. This execution will be supervised by a modules controller, which has the role to determine this order. This is in fact the heart of our approach aspect of the system. Only one module will be executed at a time and it must be triggered by the controller. Each processing chain has its own controller and a controller can act on other controllers. It means that, using the principle of abstraction, a processing chain can be considered as a (super or meta) module by itself and be embedded as a module in a super processing chain, for which the inputs will be those of the sub-module(s) at the beginning of the chain and the outputs those of the last one (see figure 2).

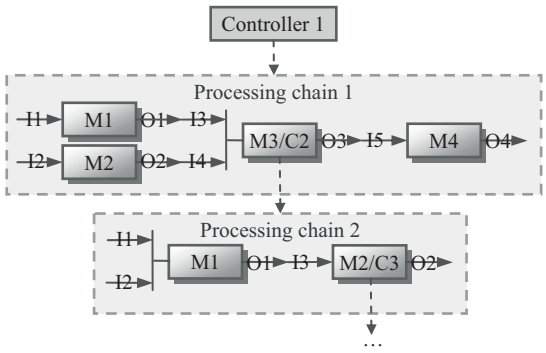


Figure 2: Processing chains and module controllers

Of course, such schemes will influence the execution order, as we will have to execute the entire sub chain when we encounter one. In figure 2, for example, the module controller of the base processing chain will first execute module M1, then M2. This will produce the required inputs for module M3, which hides itself a module controller C2

on a second processing chain that will use the previous outputs as inputs, etc.

In order to assure the execution and determine the order, the controller will need to have access to a formal representation of the processing chain. In our model, this representation will be a combinatory expression. It will be built using some elements of the combinatory logic that we have presented earlier.

Combinatory logic fills two major goals: (i) it gives a interoperable and formal representation of the solution and (ii) it gives the direct execution order, as we will see later. In order to build a processing chain, we need specific datas: (i) the list of the modules and (ii) the list of their inputs and outputs. Moreover, combinators of the combinatory logic provide operators to support the different types of interactions between modules:

- The combinator **B** is used to express the composition of two modules that are connected together with an input and an output of the same domain.
- The combinator **C** is used for ordering, that is to assure that all combinators and modules of the expression appear together to the left, then all inputs to the right.
- Finally, the combinator Φ is used to distribute the same input to two or more different modules

Let us now present many different cases of processing chains and explain how we represent them in respect with combinatory logic.

Like we said it before, a basic processing chain is constructed with only one module. In this case, the combinatory expressions representing the chains are quite simple and do not contain any combinator: they are functional expressions.


Figure 3: Simple module case

No combinator is needed and the output O1 is obtained by the application of the module 1 to the input I1: "O1 = M1 I1". If there are many inputs, we simply add them at the end of the expression.

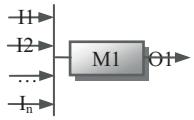


Figure 4: Simple module with n inputs case

The generalization of the base case is then "O1 = M1 I1 I2 … $I_n$".

**Serial processing chains.** A serial processing chain is composed of many modules connected together. These relations of composition between the modules are represented by the combinator **B** in the combinatory expression.

In the most general case, we consider two modules for which each of them has one input and one output. The output of the first module is connected to the input of the second module and, of course, the connectors are domain-compatibles.

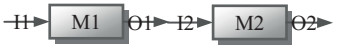


Figure 5: Two connected modules case

The logical representation is built from "right to left", that is to say that we must first consider the module the furthest to the right, then the previous one, etc.

In our case, this means the first module to be considered is M2, then M1. Basically, we pose the hypothesis that "O2 = M2 I2" and "O1 = M1 I1". We also know that I2 is connected on O1, so we replace I2 by "M1 I1", such as "O2 = M2 (M1 I1)". To combine the modules together in combinatory logic, we use the composition combinatory **B**: "O2 = **B** M2 M1 I1". The processing chain obtained is then (**B** M2 M1). This processing chain needs one input: I1.

If we add one more module to the chain, the list of the modules becomes "O1 = M1 I1", "O2 = M2 I2" and "O3 = M3 I3", I2 being connected with O1 and I3 with O2.


Figure 6: Three connected modules case

The output O3 gives the result of the processing chain. The following steps are required in order to build the representation:

O3 = M3 I3

| | |
|---|---|
| 1. O3 = M3 (**B** M2 M1 I1) | (I3 = O2) |
| 2. O3 = $\mathbf{B}^3$ M3 **B** M2 M1 I1 | (Introduction of $\mathbf{B}^3$) |
| 3. O3 = **C** $\mathbf{B}^3$ **B** M3M2 M1 I1 | (Introduction of **C**) |

At step 1 and 2, I3 is substituted by "**B** M2 M1 I1" according to what we obtained with the previous case. Then, we introduce the first combinator $\mathbf{B}^3$, using the rule "$\mathbf{B}^3$ x y z u v → x (y z u v)", for which x stands for M3, y for **B,** z for M2, u for M1 and v for I1. For the last step, another combinator, **C**, is introduced, because we need to have all the combinators to the left, the modules in the middle and the inputs at the end.

If we have four serial modules with exactly one input and one output each, the combinatory expression would be "O4 = **C** $\mathbf{B}^4$ **(C** $\mathbf{B}^3$ **B)** M4 M3 M2 M1 I1.

If we have five serial modules with exactly one input and one output each, the combinatory expression would be "O4 = **C** $\mathbf{B}^5$ **(C** $\mathbf{B}^4$ **(C** $\mathbf{B}^3$ **B))** M4 M3 M2 M1 I1", etc.

Here we can observe a relation of recursion between the modules and the combinators: the power of **B** is directly induced by the number of modules in the processing chain. We can express and eventually save any serial processing chain with a combinatory expression.

**Parallel processing chains.** When a processing chain contains at least one module with more than one input, we call it a parallel processing chain.

In the first example, a module with one input and one output ("O1 = M1 I1") is connected to a second module with two inputs and one output ("O2 = M2 I2 I3") via the input I2.

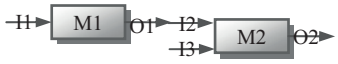


Figure 7: Module connected on the first input

To represent this chain, we only need to substitute "I2" by "M1 I1", then to compose both modules with the combinator **B** like this: "O2 = **B** M2 M1 I1 I3". Then the processing chain is (**B** M2 M1). Of course, this processing chain needs two inputs I1 and I3.

Let us take another example where a module M3 has two inputs and each of them is connected to a module, like in figure 8.

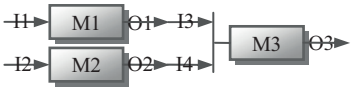


Figure 8: Two modules connected to a third module

By substituting "I3" by "M1 I1" and "I4" by "M2 I2" in the expression "O3 = M3 I3 I4", we obtain "O3 = M3 (M1 I1) (M2 I2)", then by applying combinators **B**, we find "O3 = **B** M3 M1 I1 (M2 I2)". Afterward, "I1" must be switched with "M2 I1", so we apply the combinator **C** at a distance of 2. The combinatory expression is then "O3 = $\mathbf{C_2}$ **B** M3 M1 (M2 I2) I1". Finally, we remove parentheses when we apply the combinator $\mathbf{B_3}$. O3 is then expressed by the processing chain is $\mathbf{B_3}$ $\mathbf{C_2}$ **B** M3 M1 M2 I2 I1.

In case where three modules are connected to a fourth one the processing chain applied to its inputs would be expressed by this combinatory expression: $\mathbf{B_7}$ $\mathbf{C_6}$ $\mathbf{C_6}$ $\mathbf{B_3}$ $\mathbf{C_2}$ **B** M4 M1 M2 M3 I3 I2 I1.

In case where four modules are connected to a fifth one: $\mathbf{B_{12}}$ $\mathbf{C_{11}}$ $\mathbf{C_{11}}$ $\mathbf{C_{11}}$ $\mathbf{B_7}$ $\mathbf{C_6}$ $\mathbf{C_6}$ $\mathbf{B_3}$ $\mathbf{C_2}$ **B** M5 M1 M2 M3 M4 I4 I3 I2 I1, etc.

Once again, it is possible to observe another relation of recursion. The distance of **B** and **C** are induced by the number of modules. There is a relationship established between the value of the distance and the number of modules.

We will now present a special case of parallel processing chain. This is the base case of a parallel processing chain with only one input.

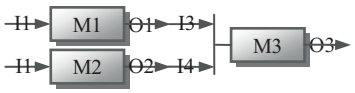


Figure 9: Two modules connected to a third module

O3 in this case is equal to $\mathbf{B_3}$ $\mathbf{C_2}$ **B** M3 M1 M2 I1 I1. In fact, we have the case of a processing chain where the two parallel modules have the same input. However, the input I1 in the combinatorial expression must not be repeated, especially when I1 is the result of the application of a certain module to its own input. This is why we introduce the combinator **W** in the combinatory expression so we obtain: $\mathbf{B_5}$ **W** $\mathbf{B_3}$ $\mathbf{C_2}$ **B** M3 M1 M2 I1. However, this expression is equivalent to another simpler one: **Φ** M3 M1 M2 I1.

**A complex 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 a combinatory expression of a somewhat complex processing chain can look like.

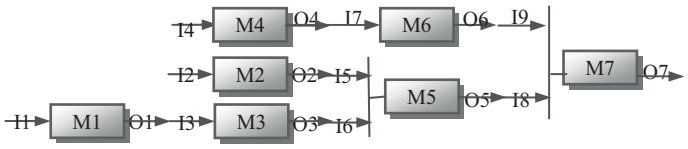


Figure 10: A complex processing chain

The output O7 of the processing chain in figure 10 is equal to "M7 I9 I8". I9 is the first input of M7 but is also the output of a subchain of two connected modules (figure 5). I9 is the ouptput of the recursively constructed module to "**B** M6 M4" whose input is I4. I8 is the output of a subchain where two modules are connected to a third one (figure 8), however I6, which is the second input of M5, is the output of a subchain of two connected modules M3 and M1. Thus we must recursively construct, first the subchain with M3 and M1 ("**B** M3 M1" whose operand is I1) and second the subchain with M5, M2 and "**B** M3 M1" ($\mathbf{B_3}$ $\mathbf{C_2}$ **B** M5 M2 (**B** M3 M1)) whose first operand is I2 and the second is I1. At last the whole constructed chain is represented by :

$\mathbf{B_3}$ $\mathbf{C_2}$ **B** M7 (**B** M6 M4) ($\mathbf{B_3}$ $\mathbf{C_2}$ **B** M5 M2 (**B** M3 M1))) I4 I2 I1

## 5. SATIM

Computationally, the chosen architecture postulates three levels of interactions with a language engineer. The first level is a **workshop** in which we find various modules, procedures or functions, in the computational sense of the term, to which applicative categories are assigned. It is also possible to add or delete modules. The second level is a **laboratory** in which the engineer builds his processing chain as he adjusts it according to his objective with the help of tests. A processing chain is thus a software containing a coherent and well organized subset of modules, procedures or functions. Once a processing chain is approved, it is considered at a third level as an autonomous **application**. SATIM comprises the following points:

(1) A set of primitive operations (base modules). Since the beginning of the 90's, the diversity of research conducted on language engineering are accompanied by a certain convergence of view (at least for the numerical approaches) on some fundamental points, as a matter of fact some primitive functions like the construction of the lexicon, the lemmatization, the segmentation, the application of particular metrics to the lexicon, etc.
(2) A set of applicative categories assigned to primitive operations. These categories indicate the type of the inputs and the outputs of a (primitive or composed) operation. From a mathematical perspective, the applicative categories connect elements of a domain (the input) to elements of another domain (the output).
(3) A set of rules to verify on the one hand the coherence of the processing chains and on the other to build their interpretation.

Following the precedent principles, we have implemented very recently SATIM (***S**ystème d'**A**nalyse et de **T**raitement de l'**I**nformation **M**ultidimensionnelle*, or Multidimensional Data Analysis and Processing System), a modular platform written in C++ used to build such processing chains in a visual environment.

For now SATIM still at prototype stage that allows us, in the first place, to assess the feasibility of the approach. Secondly, this will become the full-size project within which we aspire to design tools for language engineering and other tools for natural language processing in general.

Moreover, the formalism and the principles at the heart of SATIM are aimed to make the communication between programs a lot easier.

## 6. 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) and uses metaprogramming so we can build systems by simply adding (possibly interchangeable) modules to a controlled processing chain. A module is represented by an expression from the combinatory logic. Combinators are used as operators to represent different types of collaboration between the modules: combinators **B** and **Φ** are used respectively for the composition of modules and the distribution of inputs, while the purpose of combinator **C** is to reorder, if necessary, the expression so that the combinators and the modules and the inputs are sides apart. We have presented many different cases and have shown how to represent them.

We implemented a prototype of this model named SATIM that allows an engineer to build applications using independent modules. Like we said, we are planning to enrich the component base of NLP functionalities and use them in our future researches. Our hope is that such a platform can help research teams to collaborate together by sharing components while guaranteeing the respect of their copyright.

## References


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.

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.

Meunier, J.G. 1996. "Théorie cognitive:son impact sur le traitement de l'information textuelle". in V. Rialle et D. *Fisette Penser L'esprit ,Des sciences de la cognition a une philosophie cognitive*. Presses de Université de Grenoble. 1996 289-305

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.