Modular Unification-Based Parsers

Rémi Zajac and Jan W. Amtrup
Computing Research Laboratory, New Mexico State University
zajac,jamtrup@crl.nmsu.edu

Abstract

We present an implementation of the notion of modularity and composition applied to unification-based grammars. Monolithic unification grammars can be decomposed into sub-grammars with well-defined interfaces. Sub-grammars are applied in a sequential manner at runtime, allowing incremental development and testing of large coverage grammars. The modular approach to grammar development leads us away from the traditional view of parsing a string of input symbols as the recognition of some start symbol, and towards a richer and more flexible view where inputs and outputs share the same structural properties.

1 Introduction

We present a modular unification-based parsing architecture which allows to build a parser as a sequence of sub-components that can be developed, tested and executed one at a time. This modularization reduces the overall complexity of the grammar and enables incremental development of grammars. The motivation comes from practical problems encountered in developing broad coverage grammars where linguists face the same problems as those addressed by software engineering [Zajac 92b] and we seek similar solutions: modular approach to 'lingware' construction, encapsulation, typing, incremental compilation.

Unification-based grammars represent a definite advance in comparison to previous procedural parsing systems such as ATNs but are still notoriously difficult to develop and debug. The complexity stems mainly from the interaction between a large set of rules, where a modification to one rule can cause the parser to fail several steps after the successful application of the rule, making it difficult to trace back the source of the failure. Of course, this problem is shared by procedural parsing systems as well, where tracing back is even more difficult since the state of the system may have changed and may be irrecoverable. In a unification-based system, data is changed monotonically, making it possible, at least in theory, to recover any previous state of the system. However, most systems do not record states of computation, and previous states are only partially recoverable.

Simply inspecting the final result of a parse, although it typically contains all the derivation information, is very hard since the structures are fairly large and include a lot of detailed linguistic information. The other alternative, tracing the parsing process at runtime is also very difficult since most parsing algorithms (e.g., chart-based parsing algorithms), are non-deterministic and, for some parsers, do not proceed in a left-to-right fashion (e.g. island-driven parsers).

Some experimental Prolog environments include debuggers that allow to trace back through any

branches of a computation. This approach is attractive but fairly complex to implement, and does not solve one of the main problems, which is the debugging of a single monolithic grammar.

In this paper, we present a modular parsing architecture that attempts to address some of these problems. The architecture is derived from ideas developed in the Tipster project [Zajac, Casper & Sharples 97, Zajac 98a] as well as several transformational parsing systems, including Q-systems [Colmerauer 71] and tree-transducers such as GRADE [Nakamura, Tsujii & Nagao 84] or ROBRA [Vauquois & Boitet 85]. The grammars that are used in the system are rule-based unification grammars using a context-free skeleton (extended context-free grammars): LFG is an example of this type of grammars. Grammars are combined using a composition operator similar to the regular relation as described in [Kaplan & Kay 94] (see also [Wintner 99] for related ideas on modular context-free grammars), and each grammar specifies its interface as a set of input and output types.

Section 1 presents an overview of the parsing architecture, and how sub-grammars can be combined to build a parsing application.

Section 2 presents the unification-based parser and grammar formalism used in the system along with examples from a parser developed for a Persian-English machine translation system.

Section 3 discusses the modular organization of grammars from a linguistic point of view ands mention some runtime complexity figures of a modular parsing system.

2 Parsing Architecture

The parsing architecture is derived from previous work on NLP architectures within the Tipster framework [Zajac, Casper & Sharples 97, Zajac 98a, Bird & Liberman 99] and combines ideas from early modular NLP systems such as Q-systems [Colmerauer 71] and tree-transducers such as GRADE [Nakamura, Tsujii & Nagao 84] or ROBRA [Vauquois & Boitet 85], which provide the linguist which very flexible ways of decomposing a complex system into small building blocks which can be developed, tested and executed one by one. It uses a uniform central data structure which is shared by all components of the system, much like in blackboard systems [Boitet & Seligman 94], and incorporating ideas on chart-based NLP [Kay 73, Kay 96, Amtrup 95, Amtrup 97, Amtrup & Weber 98, Amtrup 99, Zajac 99a]. The traditional chart structure, which stores linguistic information on edges and where nodes represent a time-point in the input stream, are augmented, following Tipster ideas on 'annotations', with tags which define the kind of content an edge bears, and with spans (pairs of integers) pointing to a segment of the input stream covered by the edge. Spans are used for example in debugging and displaying a chart with edges positioned relative to the input text they cover. Tags identify for example edges built by a tokenizer, morphological analyzer, or a syntactic parser, and define sub-graphs of the whole chart that are input to some component. Linguistic information on edges is encoded using a weaker version of the Typed Feature Structure formalism as presented for example in [Zajac 92a].

Using a centralized data structure has several advantages: a chart gives a precise idea on the state of the system at any point in the computation, and integrating components to build larger systems become easy since all components implement the same interface.

In this architecture, all parsers operate in a bottom-up fashion on the same data structure: the input of a parser is a chart, maybe reduced to a linear sequence of tokens, and its output is also a chart, where active edges have been removed (and possibly edges representing sub-constituents). To operate, the first parser obviously needs a component that can read the input text and convert it to a chart.

In current applications, this is done by a tokenizer and/or a morphological analyzer which produce a chart of word structures. The tokenizer reads the input stream, classifies each token as (potential) words, numbers, symbols, or punctuation characters,[1] and produces a chart where each edge contains the analyzed token. A morphological analyzer component reads the token chart and produces for each input token one or several analyses, each stored in a different edge. A morphological analyzer can be some external program for which a wrapper implementing the chart interface has been developed, or be a pre-built component parameterized by a unification-based morphological grammar [Zajac 98b]. In turn, each parser operating on the chart is a component parameterized with a unification-based grammar.

A working system can easily be assembled from a set of components by writing a resource file, called an application definition file, which describes instantiations of components, the calling sequence of components and various global variables and parameters. Assembling components together is done using a composition operator which behaves much like the Unix pipe. When, in a Unix command, data between programs is transmitted using files (stdin/stdout) and programs are combined using the pipe '|' command, in MEAT, the data transmitted between components is a chart, and syntactically the sequence of component is combined using the ':' composition operator. In effect, the MEAT system is a specialized shell for building NLP systems. The implementation language is C++ but external components can be integrated in the system by writing wrappers (as done for several morphological analyzers previously built or used at CRL).

Each component is an instance of a C++ class which implements a standard interface. The application definition file that parameterizes the system and defines actual applications consist of three sections:

- A variable section defines global variables that typically identify locations of resources or the value of some parameters shared by several components.
- In the parameter section, global parameters are component parameters shared by several components. For example, a global parameter used by almost all components is the file containing the set of TFS definitions defining types and associated feature structures that are built by components and stored on edges of the chart.
- In the component section, each component is described as an instance of a C++ class and a list of parameters for that component (e.g. a grammar and a start symbol for a parser).
- In the application section, an application is simply defined as a sequence of components.

This application definition file is read at runtime by the system and the user provides the name of the application on the command line together with the required parameters (e.g., the input text file for a parser). For example, an application definition file would look like:

```
$ROOT=$HOME/arabic/

$MORPH=$ROOT/src/arabic.samba
$SYN=$ROOT/src/arabic.bolero
```

[1]The system includes a Unicode-base tokenizer that can operate on most scripts using spacing characters, and a large number of codeset converters for most writing systems.

```
tangoModule = $ROOT/runtime/arabic.tangoc

module Tok {
 class =  Tokenizer
 targetTag = TOKEN
 verbose = true
 inputFile = $ifile
 encoding = cp1256
 }

module Ama {
  class = MorphAnalyzer
  grammar = $MORPH
  rule = Word
  type = chart
  sourceTag = TOKEN
  targetTag = MORPH
  }

module Syn {
  class = Parser
  grammar = $SYN
  successTypes = ara.LSign
  inTag = MORPH
  outTag = SYN
  }

application parse = Tok($ifile=$1):Ama:Syn:SaveChart
```

The user would then use this application definition file (called say 'myapp') as a parameter of the meat executable:

```
% meat myapp parse test/alf.txt
```

The system would save the chart in some file that can they be browsed using the chart viewer (also a component of the system).

3 Modular Unification-Based Parsers

The parser component of the MEAT system implements a bottom-up island-driven parsing algorithm [Stock et al. 88], with a few modifications geared towards efficiency. In each rule, it is possible to identify one element of the right-hand side as the island which will be searched before attempting to apply the rule. The default behavior when no island is specified is a left-to-right regular chart-parsing algorithm.

Islands can be used to avoid triggering rules on the first element of the left-hand side when this element can appear much more frequently than the island. For example, the left-hand side of a rule describing the coordination of noun phrases would start with a noun phrase, and be triggered for each noun phrase in the default left-to-right parsing strategy. Specifying the coordinator as the island allows the rule to be triggered only in the presence of a coordination.

Elements of the right-hand side can also be specified as optional, reducing the number of disjunctive rules in the grammar.

It is also possible to specify that when a rule successfully applies, its constituents can be erased from the graph. This is used in the Persian system for example when parsing complex predicates with auxiliaries. This grammar, although written using an extended context-free formalism, is actually a regular deterministic grammar. Therefore, when an auxiliary can be integrated in a larger constituent, we can be sure that this auxiliary will never be needed to form another constituent, and we can delete the corresponding edge from the graph. This does not influence the correctness of the result but results in a reduced number of edges that have to be examined by the parsing algorithm, and reduces the number of useless constituents that will never be integrated in the final result (spurious intermediary constituents). Reducing the number of edges has also a direct benefit in terms of debugging, since the chart becomes less cluttered with useless information.

The system also includes a component that removes all edges that do not belong to a shortest path in the graph, thereby implementing a maximum coverage heuristics. This component can be used instead of the erase facility that is local to a rule, in particular when all intermediary results are needed to ensure completeness.

Each parser is created as an instance of the component Parser such as in the following definition:

```
module Syn {
  class = Parser
  grammar = $HOME/arabic/ara.bolero %$
  inTag = MORPH
  outTag = SYN
  successTypes = ara.LSign
  }
```

This definition specifies that the grammar parameterizing the parser is `ara.bolero`, that the input edges that the parser should consider are all edges tagged as `MORPH` (output of the morphological analyzer), and that the edges built by the parser as tagged as `SYN`. Moreover, the edges to be considered as part of the final result must be subsumed by the type `ara.LSign` and all other `SYN` edges are erased.

A parser's output is not an edge or a set of edges covering the whole input sentence but a chart containing all edges built by the parser that belong to a particular type. The measure of parsing success is therefore not the recognition of a single edge spanning the input, but the production of a graph, preferably connected, where edges belong to a success type. Thus, a parse is not a boolean answer anymore, but a graded result where graph connectivity, graph topology and edge type play a role. This approach proves extremely useful in building robust NLP systems which performance degrades gracefully on 'ungrammatical' input [Zajac 99b].

Since input and output are of the same kind, parsers can be chained together, the output edges of one parser becoming the input to the next parser. In the Persian system for example, there is one[2] sub-grammar used to analyze complex verbal predicates (conjugated forms with auxiliaries). For the next level of parsing, each complex verbal predicate is then viewed as a single word and sub-constituents of a complex predicate are erased from the chart.

A rule is specified in the feature structure notation and each syntactic element follows the general feature structure syntax. Although this makes it sometimes a little bit awkward, it allows to compile rules as feature structures which are themselves compiled as compact arrays of integers[3] and enable

[2] Actually 2, the second being used to parse light verb constructions.

[3] The unification algorithm operates on this data structure. Arrays of integers can also be written or loaded from a

very fast access of the rule at runtime (see for example [Wintner 97]).

A simple rule describing the Persian Passive Compound Imperfect as a Past Participle followed by Passive Auxiliary in the Compound Imperfect tense:

```
PassiveCompoundImperfect = per.Rule[
  lhs: per.Verb[
         trans: #eng,
         exp: #orth,
         lex: #lex,
         infl: [mood: per.Indicative,
                tense: per.CompoundImperfect,
                voice: per.Passive,
                person: #per,
                numberAgr: #num,
                causative: #caus,
                participle: #part,
                negation: #neg]],
  rhs: <:
    per.Verb[
         trans: #eng = Top,
         exp: #orth = Top,
         lex: #lex= Top,
         infl: [participle: #part= per.Pst,
                causative: #caus = Top,
                negation: False]]
    per.Verb[
         exp: "^sdn",
         infl: [tense: per.CompoundImperfect,
                mood: per.Indicative,
                voice: per.Active,
                person: #per = Top,
                numberAgr: #num = Top,
                negation: #neg = Top]]
       :>,
   remove: True
   ];
```

In this rule, unlike grammars written in the LFG or HPSG-style, a feature structure is created anew for the left-hand side: the left-hand side is not simply the head of the construction percolated up. Therefore, all head features that need to be transmitted need to be specified in the rule. This style makes each rule heavier than its LFG counterpart[4] but allows to categorize each constituent as belonging to a different class, e.g. a different bar-level. For example, the following rule builds an N' from a noun. Rules for noun phrases will then consider only N' and not Nouns, and so on. Recategorization of heads is not possible if the head itself becomes the left-hand-side constituent. Note that this style also make grammars inherently more reversible (assuming that generation would start with a syntactic structure) since we would only need to specify the head in this rule (specified here as the island).

file very efficiently, a facility which is used to pre-compile grammars as arrays of integers as well.

[4]It is of course possible to write LFG-style rules in this formalism.

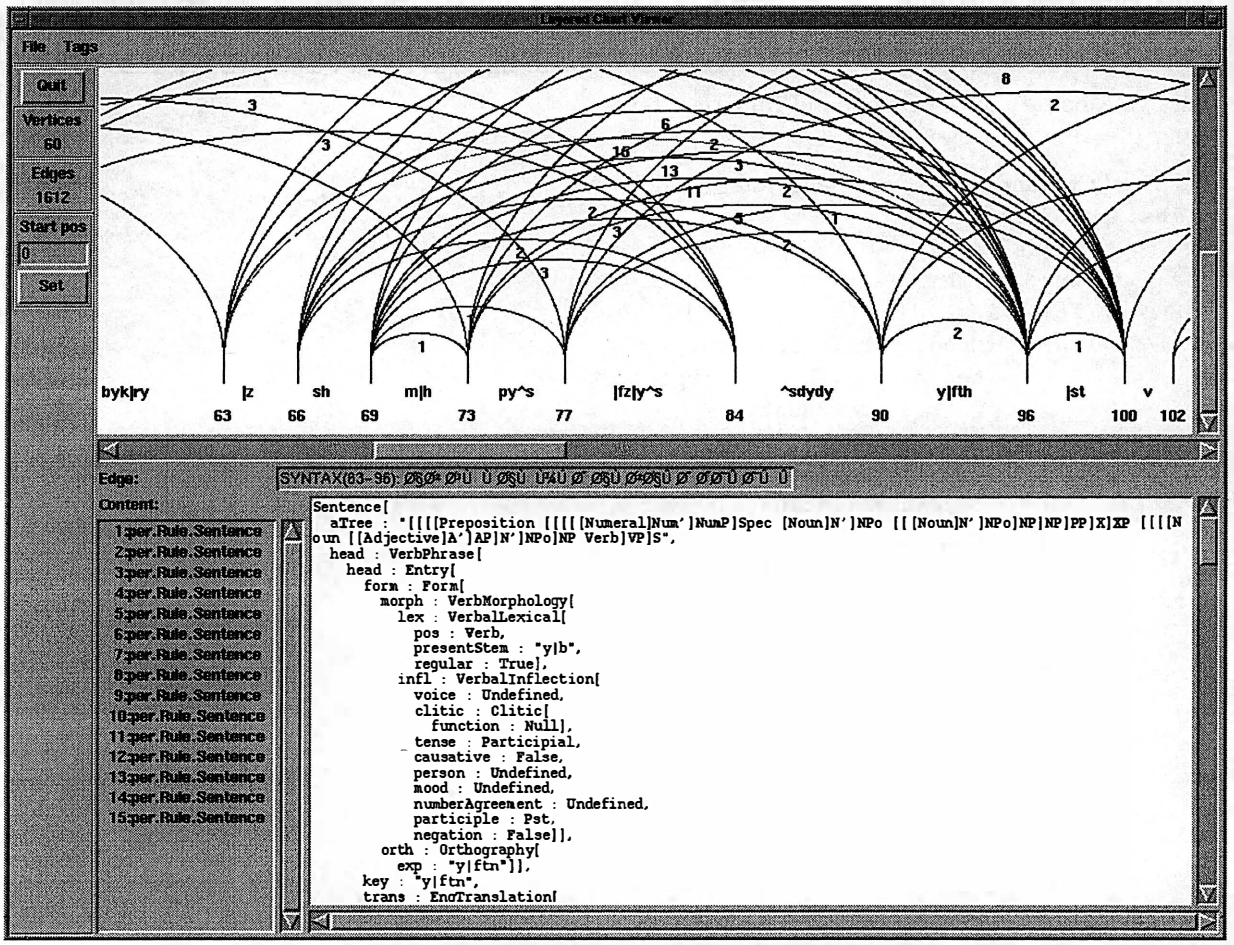

Figure 1: Viewing a complex analysis with the chart viewer.

```
NounBarNoEzafe = per.Rule[
  lhs: per.NounBar[
       head: #head,
       boundary: per.NPtrue],
  rhs: <:
       #head= per.NounOrNounCompound[
              infl: [ezafe: per.EzFalse,
                     indefEncl: False,
                     clitic.function: per.Null]]
       :>
  island: #head
  ];
```

4 Modularizing a Grammar

As already hinted in the previous section, a monolithic grammar can be decomposed in manageable sub-parts that can be developed and tested separately. Two orthogonal ways of classifying constituents can be used to identify sub-grammars:

1. A classification of constituents in levels of increasing complexity, reminiscent of the X-bar theory. Level 0 would represent so-called pre-terminal elements built directly and exclusively from lexical elements. Level 1 would provide the last interface between morphology and syntax, analyzing free morphemes (e.g. clitics), auxiliaries, non-lexicalized compounds, etc., and build new constituents from Level 0 and Level 1 constituents only. Level 1 grammars would typically fall in the class of regular grammars. Level 2 would analyze modifiers of heads, where the modifier would be from a different category (e.g. adverbs modifying adjectives, adjectives modifying nouns, but not nouns as complements of nouns). Level 2 grammars do not build recursive structures and therefore also fall in the regular class (although the form of the rules would be the one of general context-free rules). Finally, Level 3 would include recursive structures (nominal complements, relative clauses, coordination, etc.)

2. A classification according, broadly, to the category of the head: the traditional division between noun phrases, verb phrases, adjectival phrases, etc. This classification is a partial order since for example an adjective can modify a noun, but a noun will not be a subconstituent of an adjectival phrase. It can be used to order the application of sub-grammars. For example, adjectival phrases must be built before noun phrases, adverb phrases before adjectival phrases, and so on. At the higher level, level 3 constituents include about any other constituent, but at lower levels, for example adverb phrases, level 3 involve mostly or solely coordination of constituents of the same category.

In a classification of constituents into adverb phrases, adjective phrases, determiner phrases, noun phrases, verb predicates, verb phrases, and clauses, a complex monolithic grammar could be ideally be decomposed into 28 sub-grammars, each of them describing a well defined set of constructions. The grammar writer could then define a parser for each of these grammars, define classes of input/output using a set of corresponding tags, and test separately each sub-grammar in turn.

Additionally, after most of grammars from level 0, 1 and 2, it is possible to remove sub-constituent edges from the chart since level 0 and level 1 constituents are typically not ambiguous (contrary to level 3, where we find the traditional PP attachment problems and the likes), thereby reducing the number of hypotheses that the parsing algorithm needs to consider. Edge reduction has also the advantage of facilitating the inspection of the chart for debugging purposes.

5 Conclusion

The MEAT parsing architecture has been developed within the Corelli project at CRL. The system has been implemented in C++ and is used to deliver applications running on Unix and Windows platforms. Strings are internally represented using Unicode, and a set of Unicode converters is also included in the system. The system has been used for developing the Shiraz Persian-English machine translation system (see *http://crl.nmsu.edu/shiraz/*) and for the Turkish-English MT system developed within the Expedition project. It has also been used to reimplement previous glossary-based MT systems for Arabic, Japanese, Russian, Spanish and Serbo-Croatian developed in the Temple project and to develop new MT systems (e.g, Korean-English, see *http://crl.nmsu.edu/corelli/*). Finally, it is also used as the machine translation infrastructure for the Expedition project (*http://crl.nmsu.edu/expedition/*).

The two main parsers developed using this architecture are a Persian parser (used in the Shiraz MT system) and a Turkish parser (used in the Expedition Turkish MT system). The Persian grammar has been developed in 4 months and includes a hundred rules. It covers noun phrases, relative clauses, and

basic sentential constructions. The bilingual Persian-English dictionary contains about 42,000 entries (lemmas). The morphological analyzer has been developed using the Samba unification-based system [Zajac 98b]. The Persian MT system has been developed and tested on a corpus of 3000 sentences extracted from on-line news articles. The Turkish grammar (still being developed) includes about ninety rules; it has been developed in 2 months and covers noun phrases, nominalized relative clauses, some nominalized subordinate clauses, basic sentential constructions including free word order. The Turkish morphological analyzer is the two-level system developed by Kemal Oflazer [Oflazer 94]. The bilingual Turkish dictionary contains about 32,000 entries. The MT system is being developed and tested on a corpus of 260 sentences also extracted from on-line news articles.

Performances compare favorably with other parsing systems (see e.g. the comparison between ALE and AMALIA in [Wintner, Gabrilovitch & Francez 97]). Compilation time for a Persian sub-grammar of 80 rules takes about .25 seconds on a Sparc5 and parsing an average sentence (25 words) takes about .22 seconds. This time does not take into account pre-processing of input text (tokenization, morphological analysis and dictionary lookup). Total parsing times below includes the execution of edge removal executed after G2.

	G1: 5 rules	G2: 16 rules	G3: 80 rules	Total: 101 rules
Compilation	0.040s	0.120s	0.210s	0.370s
Parsing 25 words	0.020s	0.040s	0.220s	0.300s
Parsing 80 words	0.050s	0.100s	0.660s	0.820s
Parsing 700 words	0.210s	0.520s	10.100s	10.880s

The MEAT environment is publicly available under the general GNU license agreement.

Acknowledgments

The MEAT system has been implemented by Jan Amtrup and Mike Freider with contributions from many other people at CRL. The Persian and Turkish grammars have been developed by Karine Megerdoomian.

This research has been funded in part by DoD, Maryland Procurement Office, MDA904-96-C-1040.

References

[Amtrup 95] Jan W. Amtrup. 1995. "Chart-based Incremental Transfer in Machine Translation". Proceedings of the *6th International Conference on Theoretical and Methodological Issues in Machine Translation - TIM'95*, 5-7 July 1995, Leuven, Belgium. pp188-195.

[Amtrup 97] Amtrup J. (1997). "Layered Charts for Speech Translation". In Proceedings of the Seventh International Conference on Theoretical and Methodological Issues in Machine Translation, TMI '97, Santa Fe, NM, July 1997.

[Amtrup 99] Amtrup J. (1999). *Incremental Speech Translation.* Lecture Notes in Artificial Intelligence 1735, Springer Verlag, 1999.

[Amtrup & Weber 98] Jan Amtrup & Volker Weber (1998). "Time Mapping with Hypergraphs". In Proceedings of COLING'98, Montreal, Canada.

[Colmerauer 71] Alain Colmerauer (1971). "Les systèmes-Q: un formalisme poyr analyser et synthétiser des phrases sur ordinateur. Groupe TAUM, Université de Montréal.

[Bird & Liberman 99] Steven Bird & Mark Liberman (1999). "Annotation graphs as a framework for multidimensional linguistic data analysis". http://xxx.lanl.gov/abs/cs.CL/9907003.

[Boitet & Seligman 94] Christian Boitet and Mark Seligman. 1994. "The Whiteboard Architecture: a Way to Integrate Heterogeneous Components of NLP Systems". Proceedings of the *15th International Conference on Computational Linguistics – COLING'94*, August 5-9 1994, Kyoto, Japan. pp426-430.

[Kaplan & Kay 94] Ronald Kaplan and Martin Kay. 1994. *Regular Models of Phonological Rule Systems* Computational Linguistics 20(3), pp331-378.

[Kay 73] Martin Kay. 1973. "The MIND system". In R. Rustin (ed.), *Courant Computer Science Symposium 8: Natural Language Processing.* Algorithmic Press, New-York, NY. pp155-188.

[Kay 96] Martin Kay. 1996. "Chart Generation". Proceedings of the *34th Meeting of the Association for Computational Linguistics* ACL'96. pp200-204.

[Nakamura, Tsujii & Nagao 84] Jun-Ichi Nakamura, Jun-Ichi Tsujii, Makoto Nagao (1984). "Grammar Writing System (GRADE) of Mu-Machine Translation Project and its Characteristics" In Proceedings of Coling'84, Stanford University, CA, 2-6 July 1984. pp338-343.

[Oflazer 94] Kemal Oflazer. 1994. "Two-level Description of Turkish Morphology", *Literary and Linguistic Computing* 9/2.

[Stock et al. 88] Oliviero Stock, Rino Falcone and Patrizia Insinnamo (1988). "Island Parsing and Bidirectional Charts". In Proceedings of COLING'88. pp636-641.

[Vauquois & Boitet 85] Bernard Vauquois & Christian Boitet (1985). Automated Translation at Grenoble University. Computational Linguistics 11(1), pp28-36, January-March 1985.

[Wintner 99] Shuly Wintner (1999) "Modularized context-free grammars". In MOL6 - Sixth Meeting on Mathematics of Language, Pages 61-72, Orlando, Florida, July 1999.

[Wintner 97] Shuly Wintner (1997). *An Abstract Machine for Unification Grammars.* PhD Thesis, Technion – Israel Institute of Technology, Haifa, Israel.

[Wintner, Gabrilovitch & Francez 97] Shuly Wintner, Evgeniy Gabrilovitch and Nissim Francez (1997). "AMALIA – A Unified Platform for Parsing and Generation" In Proceedings of RANLP'97, Tzigov Chark, Bulgaria. pp135-142.

[Zajac 92a] Rémi Zajac (1992). *Inheritance and Constraint-based Grammar Formalism.* Computational Linguistics 18(2), pp 159-182.

[Zajac 92b] Rémi Zajac (1992). "Towards Computer-Aided Linguistic Engineering". In Proceedings of Coling'92, Nantes, France. pp828-834.

[Zajac 98a] Rémi Zajac (1998). "Reuse and Integration of NLP Components in the Calypso Architecture". In First International Conference Language Resources and Evaluation. Granada, Spain, May 1998.

[Zajac 98b] Rémi Zajac. 1998. "Feature Structures, Unification and Finite-State Transducers". FSMNLP'98, International Workshop, on Finite State Methods in Natural Language Processing, June 29 - July 1, 1998. Bilkent University, Ankara, Turkey.

[Zajac 99a] Rémi Zajac. 1999. "Graphs and feature structures in a component-based architecture." EPSRC Workshop on NLP Architectures and Language Resources, Baslow, UK, December 1998.

[Zajac 99b] Rḿi Zajac. 1999. "A Multilevel Framework for Incremental Development of MT Systems". Machine Translation Summit VII, National University of Singapore, September 13-17, 1999, Singapore.

[Zajac, Casper & Sharples 97] Rémi Zajac, Mark Casper and Nigel Sharples (1997). "An Open Distributed Architecture for Reuse and Integration of Heterogeneous Components". ANLP'97.

Posters