A Modular Toolkit for Machine Translation Based on Layered Charts

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

Abstract

We present a freely available toolkit for building machine translation systems for a large variety of languages. The toolkit uses standard linguistic data representation based on charts and typed feature structures; A modular open architecture based on standardized interfaces and processing architecture, enabling the addition of external language processing components and the configuration of new applications (plug-and-play); An open library of basic parameterizable language processing components including a morphological finite-state processor, dictionary components, an island chart parser, chart generator, and chart-based transfer engine (for MT systems). It is open-source: the C++ source code is available, and portable: targeted systems are Unix and Windows systems.

1 Introduction

The MEAT[1] machine translation toolkit was developed in order to significantly shorten the development cycle for machine translation prototypes. In addition, systems developed using the toolkit should be robust and their performance (both qualitative and quantitative) should be predictable. Finally, basic components should be easily reconfigured or modified to adapt to new applications or languages.

The toolkit is geared towards multilingual processing and offers a well-founded uniform representation of all processing steps. Based on modern computational linguistic concepts, the aim is to incorporate best practice in language engineering.

The toolkit uses throughout a standard linguistic data representation based on charts to represent processing results and typed feature structures to represent linguistic structures. The toolkit is based on a modular open architecture that uses standardized interfaces for processing components and a single simple processing architecture. The architecture enables the addition of external NLP processing components and the configuration of new applications (plug-and-play).

[1] Multilingual Architecture for Advanced Translation

The system includes an open library of basic parameterizable NLP components that include a morphological finite-state processor, dictionary components, an island chart parser, a generator, and a transfer component. Complex components such as the parser or the morphological analyzer are parameterized by using high-level declarative languages for the linguist. The system has been implemented in C++ and the source code is available. The system is portable and currently exists in a Unix version and a Windows version.

2 Representation

The architecture is derived from previous work on NLP architectures within the Tipster framework (Zajac et al., 1997; Zajac, 1998; Steven Bird, 1999) and combines ideas from early modular NLP systems such as Q-systems (Colmerauer, 1971) and tree-transducers such as GRADE (Nakamura, 1984) or ROBRA (Vauquois and Boitet, 1985), 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 and Seligman, 1994), and incorporating ideas on chart-based NLP (Kay, 1973; Kay, 1996; Amtrup, 1995; Amtrup, 1997; Amtrup and Weber, 1998; Amtrup, 1999; Zajac et al., 1999). All linguistic structures are encoded as Typed Features Structure and the association of linguistic structures to the text is maintained through the use of a Chart. The Chart itself is the main processing data structure.

2.1 Typed Feature Structures

A declarative, efficient and theoretically well-founded formalism to describe linguistic objects is an essential ingredient in any natural language processing system. A uniform data structure that is used by all components of a system offers several advantages over the use of multiple description systems. In particular, it simplifies enormously communication between NLP modules. All linguistic information in the system is encoded using Typed Feature Struc-

tures which is a versatile standard for representing linguistic structures. Typed Feature Structures are an extension of the traditional notion of linguistic features (Kay, 1979; Aït-Kaci, 1986; Pollard and Sag, 1987) and are used in all modern computational linguistic frameworks (LFG, HPSG, etc.). The TFS formalism also unifies object-oriented concepts and theorem proving techniques. TFSs are declarative with a sound logical semantics; they are associated to a small set of logical operators and can benefit of efficient implementations.

In the toolkit, the Typed Feature Structure system uses a version where types define their appropriate features (and type of their values), see e.g. Carpenter (1992). To improve the runtime behavior of the system, no complex constraints are associated to types as for example in the formalism presented in Zajac (1992). Feature structures provide a simple, versatile and uniform way of describing linguistic objects while a type system with appropriateness ensures the validity of feature descriptions and increases efficiency. Descriptions of words, syntactic structures, as well as rules for the various components can uniformly be coded as feature structures. The use of types enforces a type discipline for linguistic data: all legal linguistic structures are specified as a set of type definitions. Therefore, one of the initial task of the linguist building a system using the toolkit is to build and inventory of kinds of linguistic structures built during processing and formalize this inventory as a set of types and type definitions. The type definitions will then be used (1) by various compilers to compile (and type-check) linguistic resources such as dictionaries or grammar rules, and (2) at run-time by the various components accessing and manipulating feature structures to ensure that all feature structures created in the system are legal (i.e., conform to the type definitions). The type definitions themselves are compiled and the binary file is used as a runtime parameter by the TFS C++ library.

The formalism we developed uses a consecutive memory model for feature structures. Feature structures are stored as arrays of memory words rather than having a representation relying on the use of pointers. This is mainly done to reduce the processing needed for input/output operations and also targets at the distributed employment of a formalism. Similar representations are used for implementations of formalisms oriented towards abstract machine operations (Carpenter and Qu, 1995; Wintner and Francez, 1995). The formalism itself is implemented as a set of C++ classes representing types and feature structures. Apart from the usual operations for feature structures (subsumption and unification), the system also provides an API to destructively manipulate feature structures, a property that has to be used with care, but is extremely useful at times. The efficiency of the implementation is satisfactory and currently, we reach 4500 unifications per second in a translation application.

```
MainVerb[
 exp : "sufrir",
 infl : InflMorph[
        number : Singular,
        tense : Past,
        gender:Masculine,
        mood : Participle],
 lex : LexMorph[
        subcat : Transitive],
 trans : <:
        LSign[exp : "suffer",
              lex : LexMorph[
              regular : True]]:>]
```

2.2 Charts

Charts are a standard for representing sets of embedded linguistic structures (Hockett, 1958; Kay, 1973). They are also a versatile computational data structure for parsing (text and speech), generation and transfer (MT). A chart represents partial results independently of processing strategies and processing peculiarities (Kay, 1973; Sheil, 1976; Haruno et al., 1993; Kay, 1999). Formally a directed, acyclic, rooted graph, a chart can be viewed as a generalization of a *well-formed substring table*, capable not only of representing complete constituents ('*inactive edges*'), but also storing partial results ('*active edges*') (Sheil, 1976). The basic functions that operate on a chart are very simple. Since chart-based algorithms are almost always designed to be monotonic[2], a chart parser for example uses two main rules to add edges to the chart:

- The *Hypothesize* rule takes an edge of the chart and consults a grammar to propose new promising hypotheses that should be pursued;
- The *Combination* rule takes two edges, one of them active, the other inactive, and tries to combine them. If this combination succeeds, a new edge is created and eventually inserted into the chart.

The main advantage of formulating a natural language processing task as a chart-based process is the division of describing *what* has to be computed from *how* the individual operations have to be carried out. Kay (1980) calls the specification of a task that does not specify search and processing strategies an *algorithm schema*. In practice, one can experiment with various dimensions of strategies, e.g. top down vs. bottom up, left-to-right vs. right-to-left vs.

[2]See Wirén (1992) for a notable exception.

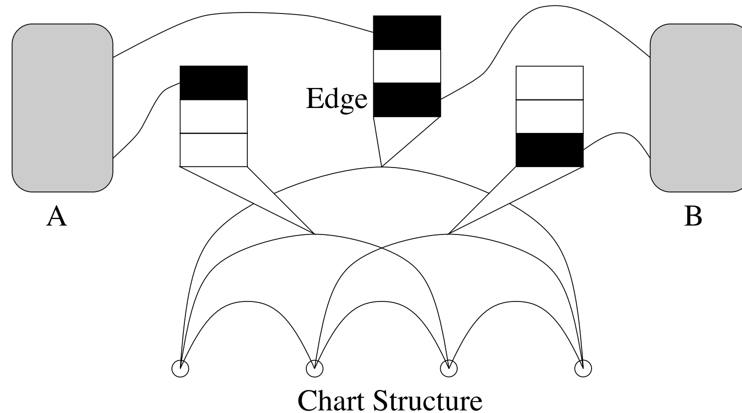

Figure 1: A layered chart.

mixed strategies or depth first search vs. breadth first search.

In our system, charts are *layered*. A layered chart is modular and declarative representation of the data manipulated by multiple processes. 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. The chart is implemented as a C++ class which provides a set of methods to traverse the graph and manipulate edges and their content.

By attaching *tags* to edges that define what kind of content an edge bears, charts can be used to store information for more than one component. In this layered chart[3] each component sees only the fraction of information that it needs to operate on. Therefore, the content of the chart gives a precise view of the current state of operations within the system, and interfaces between two modules become extremely easy to implement, as the exchange of information rests on a common concept, that of a chart edge.

At runtime, the chart is kept in memory and the various components of an application work on the same chart. Each component processes only a subset of layers, typically only two: the input layer and the output layer. For example, a parser will look at morphological edges and produce syntactic edges.

[3]The type of chart we use here is a weaker version of the layered charts defined in Amtrup and Weber (1998), as we do not distribute the chart, and we don't use parallel processing on the component side.

The chart is actually implemented as a lattice (directed rooted acyclic graph) where nodes can be time-aligned but where two time-aligned nodes are not necessarily identical. In the general case, nodes are partially ordered (with respect to time), and not completely ordered as in the traditional chart. This enables, the implementation of processes that create a sequence of edges covering a single input edge:[4]

- Normalization of contraction and elision phenomena: English contraction *don't* expanded as *do not*, or French elision *du* expanded as *de le* for example.
- POS disambiguation: sequences such as French *la porte* are ambiguous between determiner/pronoun and noun/verb. A disambiguation process would eliminate the incorrect sequences determiner+verb and pronoun+noun, leaving only two valid sequences determiner+noun and pronoun+verb, creating 2 additional distinct intermediary nodes in between the two words *la porte*.
- Chart generation, where an input edge results in a sequence of sub-edges covered by the input edge, but unrelated to other edges in the graph.

3 Architecture

The toolkit is architectured around the following three notions:

1. A module performs a complete elementary processing step.
2. An application is defined as a sequence of modules.
3. A module is an instance of a component from the component library.

[4]This also allows to represent directly the output of a speech recognizer, a word lattice.

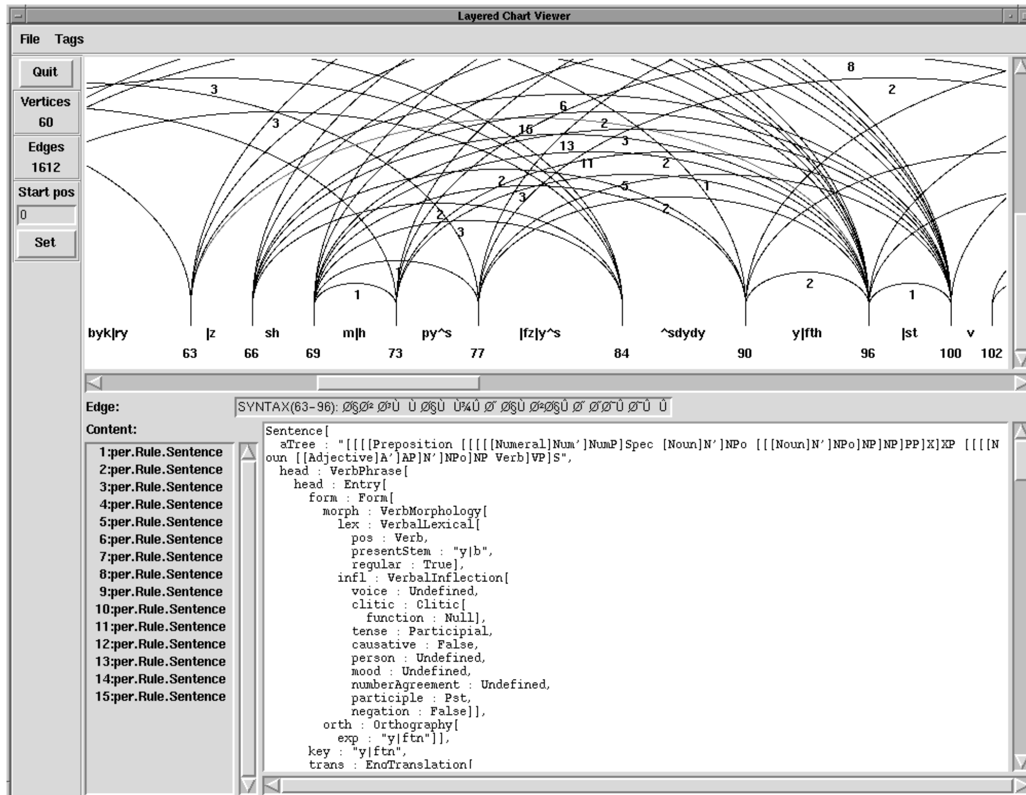

Figure 2: Viewing a complex analysis with the Chart Viewer.

3.1 Applications

An application is basically a sequence of modules. The standard I/O for a module is a layered chart, which is a global parameter for an application (special modules can also deal with files). Another global parameter for an application is the set of type definitions specifying the set of legal linguistic structures that can be stored on chart edges.

A working system (an application) can easily be assembled from a set of components by writing a resource file, called an application definition file. This file uses a simple scripting language to describe 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 M, 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).

An application definition file consists of three sections. (1) variable definitions reduce typing and enhance the transparency of application definition files. (2) Application definitions specify the sequence of modules that compose a given application. (3) Module definitions define named building blocks for applications and the parameters that they receive during a system run.

Variables provide symbolic names for long path names and make it easy to switch configuration values that pertain to several modules. Once defined, the variable name can be used instead of its value throughout the application definition file. We support both variables defined in the application definition as well as environment variables. Variable definitions in an application definition file can refer to other variables for their values. Typically, this is used in contexts like this:

```
$ROOT = /home/user/M
$SYNGRAM = $ROOT/per/SynGram.cbolero
$MORPHGRAM = $ROOT/per/Morph.samba
```

Aside from variables that are defined inside an application definition file and environment variables, we also support command line variables which are passed to the application.

Applications are defined as sequences of Modules that have to be executed to achieve a certain task. Each application is defined by its name together with the names of modules that have to be processed in turn:

```
application lookup =
  Tok($ifile=$1):Morph:Dictionary:ChartSaver
```

This application would first perform tokenization. The variable equation in the definition for the tokenizer specifies that the variable `$ifile` is set to the value of the first command line parameter. Any reference to that variable for this particular execution of the Tokenizer module would use the value given by the user on the command line. This binding is strictly local to the module for which it is defined. After Tokenization, a number of other modules are executed, including the morphological analyzer that we described earlier.

Currently, we restrict the model of operation to a sequence of modules without alternatives. We do not support graphs of modules as a model for an application. Thus, we do not support multi-threading or otherwise concurrently executed modules. Applications can be executed using a shell command or through a graphical interface (see below).

3.2 Modules

Conceptually, a module performs a single linguistic task on the data currently present in the chart. Thus, a module would take some edges of the chart as input data and provide new edges as output. In some cases, however, a module may be executed for its side effects. For instance, an input component might read a file and produce edges.

A sample module definition (performing morphological analysis of Persian text) looks like this:

```
module MorphAnalyzer {
  class = MorphAnalyzer
  grammar = $RES/morph.samba
  rule = Morphology
  type = chart
  sourceTag = TOKEN
  targetTag = MATOKEN
}
```

A module is an instance of a component from the MEAT component library (a set of C++ classes). Every module definition must specify at least one parameter, the name of the component (C++ class) of which the module is an instance (parameter `class`). By parameterizing the class representing the module within the main program, the same component can be used several times within one application. For instance, there could be several parsers within one application.

Additional parameters can be provided according to the specifications of the module in question. In the example above, the morphological grammar and initial rule need to be specified, as well as the tags that define the input and output sub-graphs of the chart.

Parameters can also be defined as global and used outside the scope of a module definition. In this case, they are global and inherited by all modules of an application. However, the local definition of a parameter overrides the global behavior. Thus, if one would define `verbose = false` on the global level, and define it as being `true` for only a subset of the components, then only those components would issue logging messages.

In the current implementation, all modules are linked in the main executable at compile time and the model does not support distributed processing. Although we have experimented with distributed architectures (Zajac et al., 1997) in the past, the overhead can be significant and the architecture must be carefully designed to support the needs for distributed components while minimizing overhead. In particular, a distributed architecture can be designed to support either collaborative research (with remote execution of components) or parallel processing on the same text. The requirements are fairly different and could be difficult to reconciliate within a single model.

3.3 Library

The toolkit includes libraries of approximately 30 processing components. Most of these components perform simple tasks and are parameterized directly in the application file. Some more complex component have external parameters such as a unification-based grammar or a dictionary file. The core library include components that have a general use:

- Utilities: Unicode tokenizer; Store/Load charts (for debugging)
- Dictionary: compiler; indexer; lookup (single words, compounds).
- Morphology: wrappers; parameterized analyzer/generator.
- Parsing: modular bi-directional island parser.
- Generation: linearizer.
- Transfer: lexical transfer; morphological feature transfer.

Each component is a C++ class that implements a pre-defined interface. The core library can easily be extended by creating a new class in the user library and linking to the other libraries at compile time. User-defined components can be used in applications as if there were native components. At

runtime, the MEAT interpreter instantiates modules defined in the application file by creating an instance of the corresponding C++ class with the appropriate parameters as specified in the module definition (in particular, an obligatory parameter is the set of types definitions defining legal feature structures). The module is executed by calling the `run()` method (which is implemented as part of the component interface).

3.4 External Components

It is possible to integrate external software modules via special components that act as wrappers. For example, the current implementation includes a morphological wrapper component that reads a file of tokens in a standard format to build a chart which is then used for further processing. This wrapper has been used to integrate several morphological analyzers (Prolog for a Spanish morphological analyzer, Lisp for a Russian one, Java for a Serbo-Croatian, and C for a Japanese and Korean).

We are currently working on extending this mechanism to provide a more general wrapping mechanism that can work on any kind of input chart, and not only a linear sequence of (possibly ambiguous) tokens. Note that it is also relatively easy to develop C++ components that implement wrappers communicating with some software module with its API if available.

4 Linguistic Knowledge

All linguistic knowledge used by the components of the core library (morphological analyzer and generator, parser, generator, dictionary lookup, transfer) is defined in external resource files that parameterize the runtime components. For example, a unification grammar used by the parser component is stored in a text file that contains the set of rules for that grammar. During the initialization phase at runtime, an instance of the parser component reads the file containing the rules that will drive the parsing algorithm. Since both input and output of the parser are charts, it is possible to to create several parser instances with different grammars and apply them in sequence on a chart (Zajac and Amtrup, 2000).

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[5] and enable very fast access of the rule at runtime (see for example (Wintner, 1997)).

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

```
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
 ];
```

Most of the language resource files are compiled before runtime and components load compact binary files instead of text source files. The toolkit provides compilers for the various formalisms and for dictionaries; dictionaries are compiled as one data file and one or more index files (tries). Since all resources files use typed feature structures as the basic representation formalism, all resource files include a set of type definitions which is used by the compilers to create binary instances of feature structures (linear arrays of integers), and by runtime components to create in memory instances (using the same array layout) or to print feature structure in a text format. The type definitions themselves are stored in a separate text file which must be itself compiled before compilation of any other resource file. The type definition file specifies the set of types used in a given application (type definitions are global to an application and are an obligory parameter to each of the components). A type definition defines super-types or sub-types (inheritance hierarchy), and the set of appropriate features for that type (and the types of their values), not but complex type constraints as in (Zajac, 1992) for example.

5 Development Environment

The development environment consist currently of two tools: the Chart Viewer and the application Runner. The chart built by some application can be saved in a file at any point during processing for further inspection. The chart can then be displayed using the Chart Viewer which allows the selective display of chart layers (by tags), and the selective display of feature structures.

The MEAT Application Runner can be run from the command line by passing to the MEAT interpreter the application file, the name of the application to be executed and the application parameters:

```
% meat -v app lookup test/doc1.txt
```

There is also a graphical tool that allows to execute applications defined in an application file using a graphical interface. This tool basically provides a

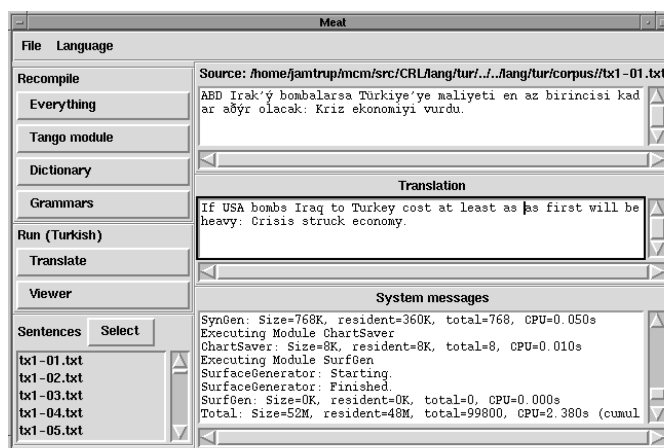

Figure 3: Executing applications from the Runner.

graphical view of the application file and allows execution of applications. (NB: this tool is still under development).

6 Conclusion

This toolkit has been used to develop a Persian-English MT system; to port previously developed glossary-based MT systems and to develop a Turkish-English MT system; and as the Machine Translation infrastructure of an elicitation-based MT system. The architecture and the core library is also used in new projects on multilingual information extraction and multilingual question-answering systems. Documentation, sources and binaries (Unix and Windows) available at `http://crl.nmsu.edu/meat`.

The toolkit is still under development as new components are added to the core library and previous components are enhanced or corrected. In the near future, we plan to enhance the library with new components for machine translation, including better transfer and generation components.

Acknowledgments

The MEAT system has been implemented by Jan Amtrup and Mike Freider with contributions from many other people at CRL.

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

References

Hassan Aït-Kaci. 1986. An Algebraic Semantics Approach to the Effective Resolution of Type Equations. *Theoretical Computer Science*, 54:293–351.

Jan W. Amtrup and Volker Weber. 1998. Time Mapping with Hypergraphs. In *Proc. of the 17 th COLING*, Montreal, Canada.

Jan W. Amtrup. 1995. Chart-based Incremental Transfer in Machine Translation. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation, TMI '95*, pages 188–195, Leuven, Belgium, July.

Jan W. Amtrup. 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.

Jan W. Amtrup. 1999. *Incremental Speech Translation*. Number 1735 in Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New York.

Christian Boitet and Mark Seligman. 1994. The "Whiteboard" Architecture: A Way to Integrate Heterogeneous Components of NLP systems. In *COLING-94: The 15th International Conference on Computational Linguistics*, Kyoto, Japan.

Bob Carpenter and Yan Qu. 1995. An Abstract Machine for Attribute-Value Logics. In *Proceedings of the 4th International Workshop on Parsing Technologies (IWPT95)*, pages 59–70, Prague. Charles University.

Bob Carpenter. 1992. *The Logic of Typed Feature Structures*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

Alain Colmerauer. 1971. Les systemes-q: un formalisme pour analyser et synthetiser des phrases sur ordinateur. Technical report, Groupe TAUM, Universite de Montreal.

Masahiko Haruno, Yasuharu Den, Yuji Mastumoto, and Makato Nagao. 1993. Bidirectional chart generation of natural language texts. In *Proc. of AAAI-93*, pages 350–356.

C. F. Hockett. 1958. *A course in modern linguistics*. Macmillan, New-York.

Martin Kay. 1973. The MIND System. In R. Rustin, editor, *Natural Language Processing*, pages 155–188. Algorithmic Press, New York.

Martin Kay. 1979. Functional grammar. In C. Chiarelloet *et al.*, editor, *Proc. 5th Annual Meeting of the Berekeley Linguistic Society*, pages 142–158, Berkeley, CA.

Martin Kay. 1980. Algorithmic Schemata and Data Structures in Syntactic Processing. Technical Report CSL-80-12, Xerox Palo Alto Research Center, Palo Alto, CA.

Martin Kay. 1996. Chart generation. In *Proc. of the 34 nd ACL*, pages 200–204, Santa Cruz, CA, June.

Martin Kay. 1999. Chart Translation. In *Machine Translation Summit VII*, pages 9–14, Singapore.

Makoto Nagao Nakamura, J. Juni-Ichi Tsujii. 1984. Grammar Writing System (GRADE) of Mu-Machine Translation Projects and its Characteristics. In *Proc. of the 10 th COLING*, Stanford, CA.

Carl Pollard and Ivan A. Sag. 1987. *Information-based Syntax and Semantics. Vol 1: Fundamentals.* CSLI Lecture Notes 13, Stanford, CA.

B. A. Sheil. 1976. Observations on Context-Free Parsing. *Statistical Methods in Linguistics*, 6:71–109.

Mark Liberman Steven Bird. 1999. A Formal Framework for Linguistic Annotation. Technical Report MS-CIS-99-01, Dept of Computer and Information Science, University of Pennsylvania.

Bernard Vauquois and Christian Boitet. 1985. Automated Translation at Grenoble University . *Computational Linguistics*, 11(1):28–36.

Shuly Wintner and Nissim Francez. 1995. Abstract Machine for Typed Feature Structures. In *Proceedings of the 5th Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Spain.

Shuly Wintner. 1997. *An Abstract Machine for Unification Grammars.* Ph.D. thesis, Technion - Israel Institute of Technology, Haifa, Israel, January.

Mats Wirén. 1992. *Studies in Incremental Natural-Language Analysis.* Ph.D. thesis, Linköping University, Linköping, Sweden.

Rémi Zajac and Jan W. Amtrup. 2000. Modular Unification-Based Parsers. In *Proc. Sixth International Workshop on Parsing Technologies*, trento, Italy, February.

Rémi Zajac, Marc Casper, and Nigel Sharples. 1997. An Open Distributed Architecture for Reuse and Integration of Heterogeneous NLP Components. In *Proc. of the 5 th Conference on Applied Natural Language Processing*, Washington, D.C.

Rémi Zajac, Malek Boualem, and Jan W. Amtrup. 1999. Specification and Implementation of Input Methods Using Finite-State Transducers. In *Fourteenth International Unicode Conference*, Boston, MA, March.

Rémi Zajac. 1992. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182.

Rémi Zajac. 1998. Annotation Management for Large-Scale NLP. In *ESSLLI-98 Workshop on Recent Advances in Corpus Annotation*, Saarbruken, Germany.