The Karamel System and Semitic Languages: Structured Multi-Tiered Morphology

François Barthélemy
CNAM-Cedric, Paris, France
INRIA-Alpage, Rocquencourt, France
francois.barthelemy@cnam.fr

Abstract

Karamel is a system for finite-state morphology which is multi-tape and uses a typed Cartesian product to relate tapes in a structured way. It implements statically compiled feature structures. Its language allows the use of regular expressions and Generalized Restriction rules to define multi-tape transducers. Both simultaneous and successive application of local constraints are possible. This system is interesting for describing rich and structured morphologies such as the morphology of Semitic languages.

1 Introduction

Karamel is a system for defining and executing multi-tape finite-state transducers where relationships between tapes are expressed using a tree structure. This structure is obtained through embedded units, which are used to analyze a tuple of strings recognized by the transducer. For instance, the units considered in an example may be affix, form and sentence.

The system includes a language and an Integrated Development Environment. The language uses extended regular expressions, computations and contextual rules. The environment provides a graphical interface to write and execute finite-state descriptions.

Karamel has many applications. For Natural Language Processing, it may be used for morphological analysis, transliteration, parsing, etc. This paper is dedicated to the application of Karamel to the morphological analysis of Semitic languages, for which both multiple tapes and complex structures are useful.

Some descriptions of the morphology of Semitic Languages use several tiers. For instance, (McCarthy, 1981) uses four tiers, one for prefixes, one for the root, one for the template (consonant-vowel pattern) and the last one for the vocalization.

Such a multi-tiered description may be implemented using a cascade of 2-tape machines (Beesley, 1998) or using a multi-tape transducer where each tier is described by a tape and the surface form by an additional tape. This is the approach of G. A. Kiraz for the Syriac language (Kiraz, 2000). Karamel is designed for the later solution.

The multi-tape feature is also interesting for describing related dialects, whenever a great part of the analysis may be shared. A separate tape is dedicated to the surface form in each dialect.

The Semitic Morphology is strongly structured by the roots. The basis of an analysis is the identification of the root. Furthermore, several layers of affixes may be distinguished around the core containing the roots: paradigmatic prefixes; affixes encoding the person, gender and number; clitics such as pronouns. This structure is conveniently defined using Karamel's units.

In the following section of the paper, we present Karamel's language, its theoretical background and its syntax. Section 3 describe the other aspects of the Karamel System: its development environment, its current state and future evolutions. Then comes an example of Semitic morphology written in Karamel, the description of Akkadian verbal flexion. The last section compares Karamel to some other systems.

2 The Karamel language

The language is based on a subclass of rational n-ary relations called *multi-grain relations* which is closed under intersection and difference (Barthélemy, 2007b). They are defined using rational expressions extended with a typed Cartesian product. This operator implements the notion of unit used to give a tree-structure to tuples of the

Proceedings of the EACL 2009 Workshop on Computational Approaches to Semitic Languages, pages 10–18,
Athens, Greece, 31 March, 2009.

relations. A unit is an inner-node of the structure.

A project is a set of finite-state machine defined over the same universe: the same alphabet, the same tape set, the same units. A project begins with declarations defining this universe. It continues with an ordered sequence of machine definitions.

The declaration section contains several clauses including classes, tapes and unit definitions. A class of symbols is a finite set of symbols. Here are some examples of class definitions:

```
class short_v is a, e, i, u;
class long_v is aa, ee, ii, uu;
class vow is a, e, i, u, long_v;
```

A symbol may belong to several classes. In the definition of a class, another class name may appear and is just an abbreviation for all its members. The class names are also used in regular expressions to denote the disjunction of their members.

The symbols written using several letters and/or digits, when there is a risk of confusion with a string of symbols, must be written enclosed by < and >. For instance, the long a is written `aa` in the class definition (`long_v`) but in a regular expression, it must be written `<aa>` because `aa` denotes a string of two short vowels. The bracketing with < and > is also used for special characters such as the space `< >`, punctuation marks (e.g. `<,>`) and the empty string `<>`.

A tape is declared with its name and the alphabet of the symbols which may appear on it.

```
tape dig: <digit>,
     fr, en: <letter>|< >|<->;
```

The alphabet is defined using a regular expression made with symbols, classes and length-preserving operators such as the disjunction and the difference.

A Karamel unit is a typed Cartesian product. The type consists in i) a number of components and ii) the tapes contained by each component. In the declaration, there is also a default value for each component.

```
unit seg is {d: dig = <digit>*;
             f: fr = <letter>*;
             e: en = <letter>*}
unit num is
        {c: dig, fr, en={seg}+}
```

The unit `seg` (for *segment*) contains three components, each using a single tape. The unit `num` (for *number*) has one component which contains three tapes (`dig, fr` and `en`).

The default value is a non-empty sequence of units of type `seg`. Cartesian products are written in regular expressions as tuples with the type name followed by the components: `{seg: 2(0?),vingt,twenty}`. Component names may be used instead of their position `{seg:e=twenty,f=vingt,d=2(0?)}`. When a component is omitted, the default value is implied. The notation `{seg}` (cf. the default value of the component `c` in `num`) is a unit `seg` with default values in all the components. Units may be embedded:

```
{num:{seg:2,vingt,twenty}
     {seg:2,-deux,-two}}
```

This example is a structured representation of the triplet `(22,vingt-deux,twenty-two)`.

In Karamel, there are three ways of defining a finite-state transducer: by a regular expression, by a computation or by a contextual rule. Regular expressions use symbols, classes of symbols, rational operations and standard extensions (for instance, optionality written ?). Furthermore, intersection, difference and negation are also available although these operations are usually not defined on transducers.

Regular expressions are defined using the `regexp` construction:

```
regexp zero is
  {seg: 0,zéro,(zero|naught)};
  {seg: <digit>*-0};
end
```

A *regexp* contains a non empty sequence of regular expressions ended with a semicolon. It denotes the disjunction of these expressions.

The second way of defining a machine is by applying operators to already defined machines. All the operators used in regular expressions may appear in such a computation, but there are some other operators as well. The first one is the *projection* which removes one or several tapes from its operand. The second one is the *external product* which combines a n-ary relation with a language on a given tape. It is used to *apply* a transducer to a given input which is not split in units yet. All possible partitioning of the input in units is first computed, and then it is intersected with one tape of the transducer. The reverse operation is the external projection which extracts a language from a

relation on a given tape by first applying the simple projection and then removing unit boundaries. These two operations are used to transduce a given possibly non-deterministic input into an output.

```
let segments=
     union(star(non_zero),zero);
```

The *let* is followed by an expression in prefixed notation with the operators written with letters. The literals are the names of previously defined machine. In our example, the expression uses the machines `zero` defined by the previous `regexp` and `non_zero` (not given here).

The last way for defining a machine consists in the Generalized Restriction Rules defined in (Yli-Jyrä and Koskenniemi, 2004). Roughly speaking, these rules are a modernized version of classical Two-Level Rules such as Context Restriction and Surface Coercion rules (Koskenniemi, 1983). They also are comparable to the rewrite rules of Xerox Finite-State Tools (Beesley and Karttunen, 2003), the difference being that rewrite rules are applied in cascades whereas GR rules may be simultaneous.

Contextual rules are defined using three regular expressions:

```
gr_rule rzero is
  {num}
constraint
   {num:seg={seg}*{seg:#0}{seg}*}
=> {num:seg={seg:#0,zéro}}
end
```

The first expression defines a universe. All the expressions in the universe which match the pattern on the left of the arrow must also match the pattern on the right. The sharp symbol is an auxiliary symbol used to make sure that the 0 on both sides is the same occurrence of this symbol. It identifies the *center* of the contextual rule. For more details about the semantics of Generalized Restriction rules, see (Yli-Jyrä and Koskenniemi, 2004).

Karamel implements non-recursive feature structures. Feature values are ordinary symbols and feature structures are typed. The types must be declared in the declaration section of the description. Feature Structures may appear anywhere in regular expressions. They are usually put on one or several specific tapes. They are statically compiled. Feature Structures are to be used with caution, because they allow the expression of long-distance dependencies which are costly and may lead to a combinatorial explosion. The feature structure compilation techniques come from (Barthélemy, 2007a).

A type is defined as follows:

```
fstruct Name is
    [gen=gender,num=1|2|3]
```

where `gender` is a class and 1, 2, 3 are symbols. Each feature is defined with its name and its domain of values, which is a finite set of symbols defined by a regular expression. A feature structure of this type is written as follows: `[Name:gen=masc,num=2]`. As usual, it is possible to specify only part of the features and their order is not important. The type name at the beginning is mandatory. Feature structures are compiled using auxiliary symbols which are not known by the user. The type name denotes a class of symbols containing all the symbols which may be used to compile a structure of this type, including auxiliary symbols.

Regular expressions and contextual rules may use variables which take their values in finite set of symbols. An expression with such a variable stands for the disjunction of all the possible valuation of the variables. Variables are especially useful to express feature structure unification.

The language offers macros called *abbreviations*. An abbreviation is a notation for an already declared unit where part of the components is defined in the declaration and another part is defined in the call. Here is an example.

```
abbrev teen is {d: dig = <digit>;
                f: fr =<letter>*;
               e: en = <letter>*}
  for {seg: 1 @d, @f,@e teen}
```

In an expression, `{teen: 6, seize,six}` is expanded in `{seg: 16,seize,sixteen}` before the compilation.

3 The Karamel System

The core of the system consists in a compiler written in Python which compiles Karamel descriptions into finite-state automata which are interpreted as transducers by the system. The compiler uses the *FSM* and *Lextools* toolkits by AT&T Research. A Karamel regular expression is first compiled in the Lextools format, then the Lextools compiler is called to compile it in FSM binary format. Some Karamel operations over transducers

such as the intersection, the union, the concatenation are directly implemented by the corresponding FSM operation on automata. Some other operations such as the two projections and the external product are performed by a script calling a sequence of FSM computations.

The development environment uses a Graphical User Interface written in HTML, CSS and Javascript. There are three main subparts: project management (creation, deletion, import, export, duplication); project development: creation, deletion, renaming, reordering, checking, compilation of a machine; machine execution, with optional introduction of run-time input, filtering of the result, projection on one or several tapes.

A dependency graph is maintained in order to ensure that a change in one machine is taken into account in all the machines which depend on it. For instance if there is the following definition: `let m3=union(m1,m2);`, any change in m_1 implies a recompilation of m_3. This recompilation is not necessarily immediate. A status is associated to each machine. The change in m_1 results in a change in the statuses of m_1 and m_3.

At the moment, the execution of a machine is possible only through the GUI, using a browser. The development of a C++ or Python function to interpret the FSM machine with the Karamel semantics is not a difficult task, but it is still to be done. Another weakness of the current version of the system is the type-checking which is not fully implemented. The type system is simple and the language is strongly typed, so every type error should be found at compile time. It is not the case yet.

Karamel will be distributed to a few kind beta-testers in a near future. We plan to add some test facilities to the environment. At medium term, a migration from FSM to openFST (Allauzen et al., 2007) and a distribution under the LGPL license are envisaged.

So far, Karamel has been used for morphology. A morphological analyzer for the Akkadian verb is presented in the next section. It is a medium size grammar. Another project describes the French morphology. It is the translation in Karamel of a grammar developed for the MMORPH system (Petitpierre and Russel, 1995). The grammar has a large coverage. It has been tested with a toy lexicon only. The other domain of application explored so far is the transliteration domain. There is a multilingual description of numbers that relates the notation with digits to a written form in several languages (French, English, Finnish). A tape is devoted to each language, a tape to the digits and several tapes for feature structures, some of which are language specific. Another project transliterates Egyptian hieroglyphs into the Latin alphabet, using an intermediate representation on a third tape.

4 An example: the Akkadian verb

In this section, we present an application of Karamel to Semitic morphology, namely a description of Akkadian verbal forms.

Akkadian is the language of the ancient Mesopotamia. It was written in cuneiform, from around 2500 B.C. up to the first century B.C. It is the main representative of the eastern branch of Semitic languages. It is divided in seven main dialects with many local variations. Its verbal morphology is a typical semitic one, with a great number of triliteral roots, some stems with usual flexion (prefixation, reduplication, infixation, vocalization). There are two infixes, `t` and `tn`. Their presence is almost orthogonal with the presence of a stem prefix and the reduplication of the second radical.

The description of the morphology in Karamel is based on a two-level structure. The first level separates verbal forms in three layers:

- a core, which contains the root, its vocalization, and also the prefixes which depend on the stem and/or aspect, infixes and gemination.
- personal affixes (prefixes and suffixes), which encode the person, the number, the gender and the case (when relevant).
- the clitics: enclitic pronoun and proclitic particles.

In the following, these units will be called *big grains*.

The second level is used for the core only, which is divided in smaller parts using the two following criteria: firstly, a unit must be significant in the analysis; secondly, it is determined by a set of features in such a way that no smaller part is uniquely determined by a subset of these features and no greater part is determined by the same set of features. Such a component is invariable for a given

value of its features, except some surface transformations.

Following the proposition of (Malbran-Labat, 2001), three kinds of vowels are distinguished. The first kind of vowel depends on the stem and the aspect. They are called *aspectual vowels*. The second kind, called *lexical vowel* depends on the stem, the aspect and a lexical category attached to the root. The third kind of vowels, the *support vowels* are not related to morphological features. They are necessary to pronounce and/or write[1] the form. The first two kinds of vowels are systematically preserved in weak forms whereas support vowels disappear in weak consonant neighborhood. Support vowel are member of the small grain containing the preceding consonant whereas lexical and aspectual vowels constitute small grains.

The different subparts of the core and their features are given in figure 1. They will be called *small grains*.

The figure 2 gives some extracts of the project. It begins with the declaration section. There is a class of all letters, subclasses of consonants, weak consonants, strong consonants, vowels, long vowels, short vowels. There is also a class for each feature domain. Several types of feature structures are defined: one for each kind of big grain (core, personal affix, pronoun, enclitic particle); a unique type for all the kinds of small grains.

The description has five tapes. The first tape contains the feature structures associated with big grains, the second tape contains the feature structures covering small grains. The third tape contains a canonical form of each grain. It correspond to the *lexical form* of traditional Two-Level grammars. The last two tapes contain the surface forms respectively in the Babylonian and the Assyrian dialects, which are slightly different, mostly in their vocalization.

Here is an example of structured analysis of the form *iptarasū*.

pers pref	core					pers suff
	rad 1	stem infix	rad 2	lex vowel	rad 3	
i	p	ta	r	a	s	ū

The tape scheme does not correspond to a multi-tiered analysis. There are several reasons for this. The first one comes from the Akkadian language. The stems and aspects are not described by patterns but divided in smaller analysis units, in particular stem analysis uses the two orthogonal dimensions called here *stem1* and *stem2*: the first one notes stem gemination and prefixation and the later, infixation. A stem is a pair (stem1,stem2). The vocalization does not require patterns of two vowels separated by the middle radical, but in most cases a pattern of only one vowel.

Another reason comes from the Karamel language: the information usually encoded in tiers appears in the unit types. For instance the information about the root tier appears in small grains of type *radical*. Similarly, the vocalization appears in the small grains of types *aspect vowel* and *lexical vowel*. The rich tree structure is sufficient to express clearly the analysis.

The morphotactics of the language is described as the sum of local constraints. It involves only the first three tapes. The elementary units, namely small grains and all the big grains but the core, are described separately. For instance, the machine `aspect_infix` (cf. figure 2) distinguishes two cases: if the feature `aspect` has `perfect` as value, then there is a small grain of type `ifx_parf` containing the infix `ta`; if the feature `aspect` has another value, then there is no grain of type `ifx_parf` in the core. The two cases are given using two different regular expressions. For more complex small grains, more cases are to be described, up to 13 for the lexical vowels which have different colors and length depending on 4 features.

Two finite-state machines describe the order of respectively small and big grains. The one for small grains called `core_morphotactics` is sketched in the figure.

The lexicon is given using a macro called `lexent`. A `lexent` (for *lexical entry*) tuple looks like a usual lexicon entry, with only lexical information, although it is a regular expression denoting a complete core, with its prefix, infixes, vowels, etc. The `lexicon` finite state machine may be directly intersected with the `sg_order` machine previously defined and all the other constraints in order to obtain a machine describing all the possible cores build on the roots given in the lexicon.

The computation of the two surface forms for

[1]The cuneiform writing is syllabic. It is impossible to write a consonant without a vowel immediately before or after it.

subpart	stem1	stem2	aspect	class	root	example
aspect prefix	X	X	X			**mu**parrisu
stem prefix	X					**šu**prusu
radical					X	ip**r**us
core infix		X				ip**ta**ras
stem1 gemination	X					upa**rr**as
aspect gemination	X	X	X			ipa**rr**as
aspect vowel	X	X	X			uparr**i**s
lexical vowel	X	X	X	X		ipr**u**s

Figure 1: Subparts and features

the two dialects is performed by a set of constraints written using regular expressions and contextual rules. They relate the lexical form and one or both surface forms. The features are used in some of them.

Rules are used for phenomena which may occur several times in a given form. For instance, the deletion of support vowels before another vowel may appear in several places: before lexical and aspectual vowels, but also when a weak consonant disappears or changes to a vowel.

In many cases however, surface transformation occur only in one given place of a form and the use of a rule is not necessary. The tree structure helps in characterizing this place. The example given in the figure is the coloration of the first vowel in some stems (II and III).

The grammar presently covers strong forms, 1-weak verbs and part of 2-weak and 3-weak verbs. Verbs with two or three weak consonants[2] and quadriliteral roots are not covered at all. The description uses 27 `regexp` clauses, 22 `let` and 6 rules.

4.1 Comparisons with other systems

There are many systems for writing finite-state machines. In this section we compare Karamel with some of them which are specialized in morphological descriptions.

The most popular system is probably the Xerox Finite State Tool (Beesley and Karttunen, 2003). It has been used, among others, for the description of Arabic morphology (Beesley, 1998). The interdigitation is handled using a compile-replace process using the replace operator (Karttunen and Beesley, 2000) (Karttunen, 1995).

The computational model is a sequential one, where two-tape transducers are merged using the *composition* operation. The descriptions are oriented, with an input and an output, but the transduction has not to be deterministic and the machines are invertible. The strings are not structured, but some structure may be marked using auxiliary symbols inserted when necessary by the user.

In order to fulfill the constraints that there are only two tapes, grammars often put heterogeneous data on a tape. For instance, the features and the lexical representations are put together on the input tape. Intermediate forms in the cascade often contain a mix of lexical and surface representations.

There are no feature structures in XFST, but features written as ordinary symbols. The scope and the unifications are written by the user.

Karamel is more declarative than XFST. Information of different natures are put on different tapes. Abstract feature structures are available. Their compilation and the unifications are automated. On the other hand, XFST is more efficient. The structure information is put only where necessary.

XFST is distributed under a commercial license.

The system MAGEAD is another system of finite-state morphology developed for Arabic dialects (Habash and Rambow, 2006). It follows the multi-tape approach proposed by George Anton Kiraz for the Syriac language (Kiraz, 2000). It has a rule-based language following the principles of (Grimley-Evans et al., 1996) in which a notion of *partition* splits forms in a sequence of units comparable to Karamel's units. But in this approach, there is only one kind of unit which relates all the tapes. Like Karamel, MAGEAD is a layer above Lextools and FSM. The system is not distributed and its description in published papers is not very detailed.

[2] There is a Akkadian verb with 3 weak consonants as root.

Declarations

```
class vowel is a, e, i, u, aa, ee, ii, uu;
class cons is b, d, g, h, ...
class num is sing, dual, plur;
class aspect is present, preterit, perfect, ...
...
fstruct fspers is [asp=aspect,pers=pers,num=num,gen=gen]
fstruct fscore is [stem1=stem1,stem2=stem2,asp=aspect,lex=lex]
...
tape lex: letter, bab: letter, assy: letter, sg: fssg,
     bg :  fspers|fscore|fsclit;
unit sgrain is {sg: sg = [fssg]; lex: lex = <letter>*,
                 bab: bab =<letter>*, assy: assy = <letter>*}
unit core is {bg: bg = [fscore];
               smallg: sg, lex, bab, assy = {sgrain}* }
...
abbrev sgi is {r1: bg = [fscore]; r2: sg = [fssg];
                r3: lex = <letter>*}
  for {core: @r1, {sgrain}* {sgrain: @r2, @r3} {sgrain}* }
abbrev lexent is {cfs: bg = [fscore]; fst: lex = <cons>;
                  snd: lex = <cons>; thd: lex = <cons>}
  for {core: @cfs, {sgrain: [fssg:typ=typ-rad]}*
        {sgrain: [fssg:typ=rad], @fst} {sgrain: [fssg:typ=typ-rad]}*
        {sgrain: [fssg:typ=rad], @snd} {sgrain: [fssg:typ=typ-rad]}*
        {sgrain: [fssg:typ=rad], @thd} }
```

Small grains morphotactics

```
regexp aspect_infix is
  {sgi: [fscore:asp=perfect],[fssg:typ=ifx_parf], ta};
  {core: [fscore:asp=aspect-perfect],
               {sgrain: [fssg:typ=typ-ifx_parf]}* };
end
...
regexp small_grain_order is
  {core: smallg=
    {sgrain: [fssg:typ=asp_pref]}? {sgrain: [fssg:typ=rad]}
    {sgrain: [fssg:typ=ifx_parf]}? {sgrain: [fssg:typ=ifx_parf]}?
    ...
let core_morphotactics=intersect(aspect_infix,stem_gemination,
                                 ...,small_grain_order);
regexp lexicon is
  {lexent: [fscore:lex=a_u,stem1=I|II|IV],p,r,s};
  {lexent: [fscore:lex=a,stem1=I|III],ṣ,b,t};
  ...
let actual_cores=intersect(core_morphotactics,lexicon);
```

Figure 2: extracts from the Akkadian project

Surface transformations

```
gr_rule delete_support_vowels is
   {core}
constraint
   {core: smallg= {sgrain}*
                  #{sgrain: lex=<letter>+<vowel>,bab=<letter><cons>}
                  {sgrain}* }
=>
   {core: smallg= {sgrain}* #{sgrain}
        {sgrain: bab=<>}? {sgrain: lex=<vowel>} {sgrain}*}
end
regexp color_u is
  {core: [fscore:stem1=II|III],
          {sgrain:lex=<cons>?<vowel>,bab=<cons>?u}{sgrain}*};
  {core: [fscore:stem1=I|IV],
          {sgrain:lex=<cons>?<vowel>,bab=<cons>?(<vowel>-u)}
          {sgrain}*};
end
```

Figure 3: extracts from the Akkadian project

The main difference is that MAGEAD has only one level of structure using only one type of Cartesian Products. Another difference is that the two systems use different kinds of contextual rules. The rules differ both by their syntax and their semantics. Furthermore, contextual rules are the main syntactic construction in MAGEAD whereas Karamel uses also regular expressions.

MMORPH is another system of partition-based morphology based on the work by Pulman and Hepple (Pulman and Hepple, 1993). There are two parts in a description: first, the morphotactics is described using a Unification Grammar where the terminals are lexical affixes and the non-terminals are feature structures; transformation rules relate the lexical and surface levels. The features are flat. Feature structures are evaluated dynamically by a unification machine.

Karamel statically compiles Feature Structures and their unification into finite-state transducers. This is efficient and part of the structures are shared. On the other hand, the grammar of feature structures must be regular and there is a risk of combinatorial explosion. MMORPH uses two kinds of units: one relates affixes to Feature Structures, the other relates small parts of lexical and surface forms (typically, substrings of length 0 to 2). Karamel uses a richer and more flexible structuration. Furthermore, the number of tapes is fixed in MMORPH and user defined in Karamel. MMORPH is distributed under the GPL license. It is not maintained any more.

5 Conclusion

In this paper, we have emphasized the application of Karamel to morphological descriptions. The multiplicity of tapes is useful at all the levels of the analysis. The abstract representation typically uses feature structures. Several tapes are to be used if different kinds of feature structures have different spans with respect to the surface form. At the intermediate level, several tapes may be used by a multi-tiered analysis. It is not the case in our example, but Karamel is compatible with an approach where each tier is put on a different tape (Kiraz, 2000). The surface level may also use several tapes. In our example, two tapes are used for two different dialects. It is also possible to use several tapes for several writings of the surface forms, for instance, a standard written form, a phonetic representation using the International Phonetic Alphabet (IPA) and a transcription in Latin alphabet.

The other main feature of Karamel is to use embedded units to relate the different tapes. This is useful to define the scope of feature structure and to distinguish several parts in the forms.

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. Openfst: A general and efficient weighted finite-state transducer library. In *Implementation and Application of Automata, 12th International Conference, CIAA*, volume 4783 of *LNC*, pages 11–23, Prague, Czech Republic.

François Barthélemy. 2007a. Finite-state compilation of feature structures for two-level morphology. In *International Workshop on Finite State Methods in Natural Language Processing (FSMNLP)*, Potsdam, Germany.

François Barthélemy. 2007b. Multi-grain relations. In *Implementation and Application of Automata, 12th International Conference (CIAA)*, pages 243–252, Prague, Czech Republic.

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications.

Kenneth R. Beesley. 1998. Arabic morphology using only finite-state operations.

Edmund Grimley-Evans, George Kiraz, and Stephen Pulman. 1996. Compiling a partition-based two-level formalism. In *COLING*, pages 454–459, Copenhagen, Denmark.

Nizar Habash and Owen Rambow. 2006. Magead: a morphological analyzer and generator for the Arabic dialects. In *ACL: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 681–688.

Lauri Karttunen and Kenneth R. Beesley. 2000. Finite-state non-concatenative morphotactics. In *Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, pages 1–12, Luxembourg (Luxembourg).

Lauri Karttunen. 1995. The replace operator. In *ACL-95*, pages 16–23, Boston, Massachusetts. Association for Computational Linguistics.

George Anton Kiraz. 2000. Multitiered nonlinear morphology using multitape finite automata: a case study on Syriac and Arabic. *Comput. Linguist.*, 26(1):77–105.

Kimmo Koskenniemi. 1983. Two-level model for morphological analysis. In *IJCAI-83*, pages 683–685, Karlsruhe, Germany.

Florence Malbran-Labat. 2001. *Manuel de langue akkadienne*. Publications de l'institut Orientaliste de Louvain (50), Peeters.

John J. McCarthy. 1981. A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry*, 12:373–418.

D. Petitpierre and G. Russel. 1995. Mmorph: the multex morphology program. Technical report, ISSCO technical report, Geneva, Switzerland.

S. Pulman and M. Hepple. 1993. A feature-based formalism for two-level phonology. *Computer Speech and Language*, 7.

Anssi Mikael Yli-Jyrä and Kimmo Koskenniemi. 2004. Compiling contextual restrictions on strings into finite-state automata. In *Proceedings of the Eindhoven FASTAR Days 2004 (September 3–4)*, Eindhoven, The Netherlands, December.