# Revisiting multi-tape automata for Semitic morphological analysis and generation

**Mans Hulden**
University of Arizona
Department of Linguistics
mhulden@email.arizona.edu

## Abstract

Various methods have been devised to produce morphological analyzers and generators for Semitic languages, ranging from methods based on widely used finite-state technologies to very specific solutions designed for a specific language or problem. Since the earliest proposals of how to adopt the elsewhere successful finite-state methods to root-and-pattern morphologies, the solution of encoding Semitic grammars using multi-tape automata has resurfaced on a regular basis. Multi-tape automata, however, require specific algorithms and reimplementation of finite-state operators across the board, and hence such technology has not been readily available to linguists. This paper, using an actual Arabic grammar as a case study, describes an approach to encoding multi-tape automata on a single tape that can be implemented using any standard finite-automaton toolkit.

## 1 Introduction

### 1.1 Root-and-pattern morphology and finite-state systems

The special problems and challenges embodied by Semitic languages have been recognized from the early days of applying finite-state methods to natural language morphological analysis. The language model which finite-state methods have been most successful in describing—a model where morphemes concatenate in mostly strict linear order—does not translate congenially to the type of root-and-pattern morphology found in e.g. Arabic and Hebrew (Kataja and Koskenniemi, 1988; Lavie et al., 1988).

In Arabic, as in most Semitic languages, verbs have for a long time been analyzed as consisting of three elements: a (most often) triconsonantal root, such as *ktb* (ك ت ب), a vowel pattern containing grammatical information such as voice (e.g. the vowel *a*) and a derivational template, such as `CVCVC` indicating the class of the verb, all of which are interdigitated to build a stem, such as *katab* (كَتَب).[1] This stem is in turn subject to more familiar morphological constructions including prefixation and suffixation, yielding information such as number, person, etc, such as *kataba* (كتبَ), the third person singular masculine perfect form.

The difficulty of capturing this interdigitation process is not an inherent shortcoming of finite-state automata or transducers per se, but rather a result of the methods that are commonly used to construct automata. Regular expressions that contain operations such as concatenation, union, intersection, as well as morphotactic descriptions through right-linear grammars offer an unwieldy functionality when it comes to interleaving strings with one another in a regulated way. But, one could argue, since large scale morphological analyzers as finite-state automata/transducers have indeed been built (see e.g. Beesley (1996, 1998b,a)), the question of how to do it becomes one of construction, not feasibility.

### 1.2 Multitape automata

One early approach, suggested by Kay (1987) and later pursued in different variants by Kiraz (1994, 2000) among others, was to, instead of modeling morphology along the more traditional finite-state transducer, modeling it with a $n$-tape automaton, where tapes would carry precisely this interleaving

[1] Following autosegmental analyses, this paper assumes the model where the vocalization is not merged with the pattern, i.e. we do not list separate patterns for vocalizations such as `CaCaC` as is assumed more traditionally. Which analysis to choose largely a matter of convenience, and the methods in this paper apply to either one.

*Proceedings of the EACL 2009 Workshop on Computational Approaches to Semitic Languages*, pages 19–26,
Athens, Greece, 31 March, 2009. 

that is called for in Semitic interdigitation. However, large-scale multitape solutions containing the magnitude of information in standard Arabic dictionaries such as Wehr (1979) have not been reported.

To our knowledge, two large-scale morphological analyzers for Arabic that strive for reasonable completeness have been been built: one by Xerox and one by Tim Buckwalter (Buckwalter, 2004). The Xerox analyzer relies on complex extensions to the finite-state calculus of one and two-tape automata (transducers) as documented in Beesley and Karttunen (2003), while Buckwalter's system is a procedural approach written in Perl which decomposes a word and simultaneously consults lexica for constraining the possible decompositions. Also, in a similar vein to Xerox's Arabic analyzer, Yona and Wintner (2008) report on a large-scale system for Hebrew built on transducer technology. Most importantly, none of these very large systems are built around multi-tape automata even though such a construction from a linguistic perspective would appear to be a fitting choice when dealing with root-and-pattern morphology.

### 1.3 n-tape space complexity

There is a fundamental space complexity problem with multi-tape automata, which is that when the number of tapes grows, the required joint symbol alphabet grows with exponential rapidity unless special mechanisms are devised to curtail this growth. This explosion in the number of transitions in an n-tape automaton can in many cases be more severe than the growth in the number of states of a complex grammar.

To take a simple, though admittedly slightly artificial example: suppose we have a 5-tape automaton, each tape consisting of the same alphabet of, say 22 symbols $\{s_1, \ldots, s_{22}\}$. Now, assume we want to restrict the co-occurrence of $s_1$ on any combination of tapes, meaning $s_1$ can only occur once on one tape in the same position, i.e. we would be accepting any strings containing a symbol such as $s_1$:$s_2$:$s_2$:$s_2$:$s_2$ or $s_2$:$s_2$:$s_2$:$s_2$:$s_3$ but not, $s_1$:$s_2$:$s_3$:$s_4$:$s_1$. Without further treatment of the alphabet behavior, this yields a multi-tape automaton which has a single state, but 5,056,506 transitions—each transition naturally representing a legal combination of symbols on the five tapes. This kind of transition blow-up is not completely inevitable: of course one can devise many tricks to avoid it, such as adding certain semantics to the transition notation—in our example by perhaps having a special type of 'failure' transition which leads to non-acceptance. For the above example this would cut down the number of transitions from 5,056,506 to 97,126. The drawback with such methods is that any changes will tend to affect the entire finite-state system one is working with, requiring adaptations in almost every underlying algorithm to construct automata. One is then unable to leverage the power of existing software designed for finite-state morphological analysis, but needs to build special-purpose software for whatever multi-tape implementation one has in mind.[2]

### 1.4 The appeal of the multi-tape solution

The reason multi-tape descriptions of natural language morphology are appealing lies not only in that such solutions seem to be able to handle Semitic verbal interdigitation, but also in that a multi-tape solution allows for a natural *alignment* of information regarding segments and their grammatical features, something which is often missing in finite-state-based solutions to morphological analysis. In the now-classical way of constructing morphological analyzers, we have a transducer that maps a string representing an unanalyzed word form, such as *kataba* (كَتَبَ) to a string representing an analyzed one, e.g. `ktb +FormI +Perfect +Act +3P +Masc +Sg`. Such transductions seldom provide grammatical component-wise alignment information telling which parts of the unanalyzed words contribute to which parts of the grammatical information. Particularly if morphemes signifying a grammatical category are discontinuous, this information is difficult to provide naturally in a finite-automaton based system without many tapes. A multi-tape solution, on the other hand,

[2] Two anonymous reviewers point out the work by Habash et al. (2005) and Habash and Rambow (2006) who report an effort to analyze Arabic with such a multitape system based on work by Kiraz (2000, 2001) that relies on custom algorithms devised for a multitape alphabet. Although Habash and Rambow do not discuss the space requirements in their system, it is to be suspected that the number of transitions grows quickly using such an method by virtue of the argument given above. These approaches also use a small number of tapes (between 3 and 5), and, since the number of transitions can increase exponentially with the number of tapes used, such systems do not on the face of it appear to scale well to more than a handful of tapes without special precautions.

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_{input}$ | k | a | t | a | b | a |
| $T_{root}$ | k | | t | | b | |
| $T_{form}$ | **Form I** | | | | | |
| $T_{ptrn}$ | C | V | C | V | C | |
| $T_{paff}$ | | | | | | a |
| $T_{affp}$ | | | | | | +3P<br>+Masc<br>+Sg |
| $T_{voc}$ | | a | | a | | |
| $T_{vocp}$ | | | | | | +Act |
| ... | | | | | | |

Table 1: *A possible alignment of 8 tapes to capture Arabic verbal morphology.*

can provide this information by virtue of its construction. The above example could in an 8-tape automaton encoding be captured as illustrated in table 1, assuming here that $T_{input}$ is the input tape, the content of which is provided, and the subsequent tapes are output tapes where the parse appears.

In table 1, we see that the radicals on the *root* tape are aligned with the input, as is the pattern on the *pattern* tape, the suffix *-a* on the suffix tape, which again is aligned with the parse for the suffix on the affix parse tape ($affp$), and finally the vocalization *a* is aligned with the input and the pattern. This is very much in tune with both the type of analyses linguists seem to prefer (McCarthy, 1981), and more traditional analyses and lexicography of root-and-pattern languages such as Arabic.

In what follows, we will present an alternate encoding for multi-tape automata together with an implementation of an analyzer for Arabic verbal morphology. The encoding simulates a multi-tape automaton using a simple one-tape finite-state machine and can be implemented using standard toolkits and algorithms given in the literature. The encoding also avoids the abovementioned blow-up problems related to symbol combinations on multiple tapes.

## 2 Notation

We assume the reader is familiar with the basic notation regarding finite automata and regular expressions. We will use the standard operators of Kleene closure ($L^*$), union ($L_1 \cup L_2$), intersection ($L_1 \cap L_2$), and assume concatenation whenever there is no overt operator specified ($L_1 L_2$). We use the symbol $\Sigma$ to specify the alphabet, and the shorthand $\backslash a$ to denote any symbol in the alphabet except $a$. Slight additional notation will be introduced in the course of elaborating the model.

## 3 Encoding

In our implementation, we have decided to encode the multi-tape automaton functionality as consisting of a single string read by a single-tape automaton, where the multiple tapes are all evenly interleaved. The first symbol corresponds to the first symbol on tape 1, the second to the first on tape 2, etc.:

| | | | |
|---|---|---|---|
| $T_1$ | ↓ | ↓ | ↓ |
| ... | | | |
| $T_{n-1}$ | ↓ | ↓ | ↓ |
| $T_n$ | ↗ | ↗ | ↓ |
| ... | | | |

For instance, the two-tape correspondence:

| | | |
|---|---|---|
| $T_1$ | a | |
| $T_2$ | b | c |

would be encoded as the string $ab\varepsilon c$, $\varepsilon$ being a special symbol used to pad the blanks on a tape to keep all tapes synchronized.

This means that, for example, for an 8-tape representation, every 8th symbol from the beginning is a symbol representing tape 1.

Although this is the final encoding we wish to produce, we have added one extra temporary feature to facilitate the construction: every symbol on any 'tape' is always preceded by a symbol indicating the tape number drawn from an alphabet $T_1, \ldots, T_n$. These symbols are removed eventually. That means that during the construction, the above two-tape example would be represented by the string $T_1 a T_2 b T_1 \varepsilon T_2 c$. This simple redundancy mechanism will ease the writing of grammars and actually limit the size of intermediate automata during construction.

## 4 Construction

### 4.1 Overview

We construct a finite-state n-tape simulation grammar in two steps. Firstly we populate each 'tape' with all grammatically possible strings. That means that, for our Arabic example, the root tape

should contain all possible roots we wish to accept, the template tape all the possible templates, etc. We call this language the Base. The second step is to constrain the co-occurrence of symbols on the individual tapes, i.e. a consonant on the root tape must be matched by a consonant of the input tape as well as the symbol `C` on the pattern tape, etc. Our grammar then consists of all the permitted combinations of tape symbols allowed by a) the `Base` and b) the `Rules`. The resulting language is simply their intersection, viz.:

$$\text{Base} \cap \text{Rules}$$

### 4.2 Populating the tapes

We have three auxiliary functions, `TapeL(X,Y)`, `TapeM(X,Y)`, and `TapeA(X,Y)`, where the argument `X` is the tape number, and `Y` the language we with to insert on tape `X`.[3] `TapeL(X,Y)` creates strings where every symbol from the language `Y` is preceded by the tape indicator $T_X$ and where the entire tape is left-aligned, meaning there are no initial blanks on that tape. `TapeM` is the same function, except words on that tape can be preceded by blanks and succeeded by blanks. `TapeA` allows for any alignment of blanks within words or to the left or right. Hence, to illustrate this behavior, `TapeL(4,C V C V C)` will produce strings like:

$$\texttt{X}T_4\mathbf{C}\texttt{X}T_4\mathbf{V}\texttt{X}T_4\mathbf{C}\texttt{X}T_4\mathbf{V}\texttt{X}T_4\mathbf{C}\texttt{Y}$$

where `X` is any sequence of symbols not containing the symbol $T_4$, and `Y` any sequence possibly containing $T_4$ but where $T_4$ is always followed by $\varepsilon$, i.e. we pad all tapes at the end to allow for synchronized strings on other tapes containing more material to the right.

Now, if, as in our grammar, tape 4 is the template tape, we would populate that tape by declaring the language:

`TapeM(4,Templates)`

assuming `Templates` is the language that accepts all legal template strings, e.g. `CVCVC`, `CVCCVC`, etc.

Hence, our complete `Base` language (continuing with the 8-tape example) is:

$$\begin{array}{ll}
\texttt{TapeL(1,Inputs)} & \cap \\
\texttt{TapeA(2,Roots)} & \cap \\
\texttt{TapeL(3,Forms)} & \cap \\
\texttt{TapeM(4,Templates)} & \cap \\
\texttt{TapeA(5,Affixes)} & \cap \\
\texttt{TapeM(6,Parses)} & \cap \\
\texttt{TapeA(7,Voc)} & \cap \\
\texttt{TapeL(8,VocParses)} & \cap \\
(T_1\Sigma T_2\Sigma T_3\Sigma T_4\Sigma T_5\Sigma T_6\Sigma T_7\Sigma T_8\Sigma)^* &
\end{array}$$

This will produce the language where all strings are multiples of 16 in length. Every other symbol is the $T_X$ tape marker symbol and every other symbol is the actual symbol on that tape (allowing for the special symbol $\varepsilon$ also to represent blanks on a tape). Naturally, we will want to define `Inputs` occurring on tape 1 as any string containing any combination of symbols since it represents all possible input words we wish to parse. Similarly, tape 2 will contain all possible roots, etc. This `Base` language is subsequently constrained so that symbols on different tapes align correctly and are only allowed if they represent a legal parse of the word on the input tape (tape 1).

### 4.3 Constructing the rules

When constructing the rules that constrain the co-occurrence of symbols on the various tapes we shall primarily take advantage of the $\Rightarrow$ operator first introduced for two-level grammars by Koskenniemi (1983).[4] The semantics is as follows. A statement:

$$X \Rightarrow L_1 \ \_ \ R_1, \ldots, L_n \ \_ \ R_n$$

where $X$ and $L_i$, $R_i$ are all regular languages defines the regular language where every instance of a substring drawn from the language $X$ must be surrounded by some pair $L_i$ and $R_i$ to the left and right, respectively.[5]

Indeed, all of our rules will consist exclusively of $\Rightarrow$ statements.

To take an example: in order to constrain the template we need two rules that effectively say that every $C$ and $V$ symbol occurring in the template

[3] See the appendix for exact definitions of these functions.

[4] There is a slight, but subtle difference in notation, though: the original two-level $\Rightarrow$ operator constrained single symbols only (such as $a{:}b$, which was considered at compile-time a single symbol); here, the argument $X$ refers to any arbitrary language.

[5] Many finite-state toolkits contain this as a separate operator. See Yli-Jyrä and Koskenniemi (2004) and Hulden (2008) for how such statements can be converted into regular expressions and finite automata.

tape must be matched by 1) a consonant on the root tape and 2) a vowel on the input tape. Because of our single-tape encoding the first rule translates to the idea that every $T_4$ $C$ sequence must be directly preceded by $T_2$ followed by some consonant followed by $T_3$ and any symbol at all:

$$T_4 \ C \ \Rightarrow \ T_2 \ \texttt{Cons} \ T_3 \ \Sigma \ \_ \qquad (1)$$

and the second one translates to:

$$T_4 \ V \ \Rightarrow \ T_1 \ \texttt{Vow} \ T_2 \ \Sigma \ T_3 \ \Sigma \ \_ \qquad (2)$$

assuming that `Vow` is the language that contains any vowel and `Cons` the language that contains any consonant.

Similarly, we want to constrain the `Forms` parse tape that contains symbols such as `Form_I`, `Form_II` etc., so that if, for example, `Form_I` occurs on that tape, the pattern `CVCVC` must occur on the pattern tape.[6]

$$T_3 \ \texttt{Form\_I} \Rightarrow \ \_ \ \texttt{TapeM(4,C V C V C)} \qquad (3)$$

and likewise for all the other forms. It should be noted that most constraints are very strictly local to within a few symbols, depending slightly on the ordering and function of the tapes. In (1), for instance, which constrains a symbol on tape 4 with a consonant on tape 2, there are only 2 intervening symbols, namely that of tape 3. The ordering of the tapes thus has some bearing on both how simple the rules are to write, and the size of the resulting automaton. Naturally, tapes that constrain each other are ideally placed in adjacent positions whenever possible.

Of course, some long-distance constraints will be inevitable. For example, Form II is generally described as a `CVCCVC` pattern, where the extra consonant is a geminate, as in the stem `kattab`, where the *t* of the root associates with both `C`'s in the pattern. To distinguish this `C` behavior from that of Form X which is also commonly described with two adjacent `C` symbols where, however, there is no such association (as in the stem *staktab*) we need to introduce another symbol. This symbol $\texttt{C}_2$ occurs in Form II, which becomes $\texttt{CVCC}_2\texttt{VC}$. We then introduce a constraint to the effect that any $\texttt{C}_2$-symbol must be matched on the input by a consonant, which is identical to the previous consonant on the input tape.[7] These long-distance dependencies can be avoided to some extent by grammar engineering, but so long as they do not cause a combinatorial explosion in the number of states of the resulting grammar automaton, we have decided to include them for the sake of clarity.

To give an overview of some of the subsequent constraints that are still necessary, we include here a few descriptions and examples (where the starred (***) tape snippets exemplify illegal configurations):

- Every root consonant has a matching consonant on the input tape

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_1$ | k | a | t | a | b | a |
| $T_2$ | k | | t | | b | |
| $T_1$ | k | a | t | a | b | a |
| $T_2$*** | d | | r | | s | |

- A vowel in the input which is matched by a V in the pattern, must have a corresponding vocalization vowel

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_1$ | k | a | t | a | b | a |
| $T_4$ | C | V | C | V | C | |
| $T_7$ | | a | | a | | |
| $T_1$ | k | a | t | a | b | a |
| $T_4$ | C | V | C | V | C | |
| $T_7$*** | | u | | i | | |

- A position where there is a symbol in the input either has a symbol in the pattern tape or a symbol in the affix tape (but not both)

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_1$ | k | a | t | a | b | a |
| $T_4$ | C | V | C | V | C | |
| $T_5$ | | | | | | a |
| $T_1$ | k | a | t | a | b | a |
| $T_4$ | C | V | C | V | C | |
| $T_5$*** | | | | | | |

[6] To be more exact, to be able to match and parse both fully vocalized words such as *wadarasat* (وَدَرَسَتْ), and unvocalized ones, such as *wdrst* (ودرست), we want the pattern `CVCVC` to actually be represented by the regular expression `C (V) C (V) C`, i.e. where the vowels are optional. Note, however, that the rule that constrains $T_4$ $V$ above only requires that the $V$ matches *if* there indeed is one. Hence, by declaring vowels in patterns (and vocalizations) to be optional, we can always parse any partially, fully, or unvocalized verb. Of course, fully unvocalized words will be much more ambiguous and yield more parses.

[7] The idea to preserve the gemination in the grammar is similar to the solutions regarding gemination and spreading of Forms II, V, and IX documented in Beesley (1998b) and Habash and Rambow (2006).

### 4.4 The final automaton

As mentioned above, the symbols $\{T_1, \ldots, T_n\}$ are only used during construction of the automaton for the convenience of writing the grammar, and shall be removed after intersecting the Base language with the Rules languages. This is a simple substitution $T_X \rightarrow \epsilon$, i.e. the empty string. Hence, the grammar is compiled as:

$$\texttt{Grammar} = h(\texttt{Base} \cap \texttt{Rules})$$

where $h$ is a homomorphism that replaces $T_X$ symbols with $\epsilon$, the empty string.

## 5 Efficiency Considerations

Because the construction method proposed can very quickly produce automata of considerable size, there are a few issues to consider when designing a grammar this way. Of primary concern is that since one is constructing deterministic automata, long-distance constraints should be kept to a minimum. Local constraints, which the majority of grammar rules encode, yield so-called $k$-testable languages when represented as finite automata, and the state complexity of their intersection grows additively. For larger $k$, however, growth is more rapid which means that, for example, when one is designing the content of the individual tapes, care should be taken to ensure that segments or symbols which are related to each other preferably align very closely on the tapes. Naturally, this same goal is of linguistic interest as well and a grammar which does not align grammatical information with segments in the input is likely not a good grammar. However, there are a couple of ways in which one can go astray. For instance, in the running example we have presented, one of the parse tapes has included the symbol `+3P +Masc +Sg`, aligned with the affix that represents the grammatical information:

...

| | | | | | | |
|---|---|---|---|---|---|---|
| $T_5$ | | | | | | a |
| $T_6$ | | | | | | +3P<br>+Masc<br>+Sg |

...

However, if it be the case that what the parse tape reflects is a prefix or a circumfix, as will be the case with the imperfective, subjunctive and jussive forms, the following alignment would be somewhat inefficient:

...

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $T_5$ | t | a | | | | | | |
| $T_6$ | | | | | | | | +3P<br>+Fem<br>+Sg |

...

This is because the prefix *ta*, which appears early in the word, is reflected on tape 6 at the end of the word, in effect unnecessarily producing a very long-distance dependency and hence duplicates of states in the automaton encoding the intervening material. A more efficient strategy is to place the parse or annotation tape material as close as possible to the segments which have a bearing on it, i.e.:

...

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $T_5$ | t | a | | | | | | |
| $T_6$ | +3P<br>+Fem<br>+Sg | | | | | | | |

...

This alignment can be achieved by a constraint in the grammar to the effect that the first non-blank symbol on the affix tape is in the same position as the first non-blank symbol on the affix parse tape.

It is also worth noting that our implementation does not yet restrict the co-occurrence of roots and forms, i.e. it will parse any word in any root in the lexicon in any of the forms I-VIII, X. Adding these restrictions will presumably produce some growth in the automaton. However, for the time being we have also experimented with accepting any triliteral root—i.e. any valid consonantal combination. This has drastically cut the size of the resulting automaton to only roughly 2,000 states without much overgeneration in the sense that words will not incorrectly be matched with the wrong root. The reason for this small footprint when not having a 'real' lexicon is fairly obvious—all dependencies between the root tape and the pattern tape and the input tape are instantly resolved in the span of one 'column' or 7 symbols.

## 6 Algorithmic additions

Naturally, one can parse words by simply intersecting `TapeL(1, word)` $\cap$ `Grammar`, where

`word` is the word at hand and printing out all the legal strings. Still, this is unwieldy because of the intersection operation involved and for faster lookup speeds one needs to consider an algorithmic extension that performs this lookup directly on the `Grammar` automaton.

### 6.1 Single-tape transduction

For our implementation, we have simply modified the automaton matching algorithm in the toolkit we have used, *foma*[8] to, instead of matching every symbol, matching the first symbol as the 'input', then outputting the subsequent $n$ (where $n$ is 7 in our example) legal symbols if the subsequent input symbols match. Because the grammar is quite constrained, this produces very little temporary ambiguity in the depth-first search traversal of the automaton and transduces an input to the output tapes in nearly linear time.

## 7 Future work

The transduction mechanism mentioned above works well and is particularly easy to implement when the first 'tape' is the input tape containing the word one wants to parse, since one can simply do a depth-first search until the the next symbol on the input tape (in our running example with 8 tapes, that would be 7 symbols forward) and discard the paths where the subsequent tape 1 symbols do not match, resulting in nearly linear running time. However, for the generation problem, the solution is less obvious. If one wanted to supply any of the other tapes with a ready input (such as form, root, and a combination of grammatical categories), and then yield a string on tape 1, the problem would be more difficult. Naturally, one can intersect various TapeX(n, content) languages against the grammar, producing all the possible input strings that could have generated such a parse, but this method is rather slow and results only in a few parses per second on our system. Devising a fast algorithm to achieve this would be desirable for applications where one wanted to, for instance, generate all possible vocalization patterns in a given word, or for IR purposes where one would automatically apply vocalizations to Arabic words.

[8] See the appendix.

## 8 Conclusion

We have described a straightforward method by which morphological analyzers for languages that exhibit root-and-pattern morphology can be built using standard finite-state methods to simulate multi-tape automata. This enables one to take advantage of already widely available standard toolkits designed for construction of single-tape automata or finite-state transducers. The feasibility of the approach has been tested with a limited implementation of Arabic verbal morphology that contains roughly 2,000 roots, yielding automata of manageable size. With some care in construction the method should be readily applicable to larger projects in Arabic and other languages, in particular to languages that exhibit root-and-pattern or templatic morphologies.

## References


Beesley, K. and Karttunen, L. (2003). *Finite-State Morphology*. CSLI, Stanford.

Beesley, K. R. (1996). Arabic finite-state analysis and generation. In *COLING '96*.

Beesley, K. R. (1998a). Arabic morphology using only finite-state operations. In *Proceedings of the Workshop on Computational Approaches to Semitic Languages COLING-ACL*, pages 50–57.

Beesley, K. R. (1998b). Consonant spreading in Arabic stems. In *ACL*, volume 36, pages 117–123. Association for Computational Linguistics.

Beeston, A. F. L. (1968). *Written Arabic: An approach to the basic structures*. Cambridge University Press, Cambridge.

Buckwalter, T. (2004). Arabic morphological analyzer 2.0. *LDC*.

Habash, N. and Rambow, O. (2006). MAGEAD: A morphological analyzer and generator for the Arabic dialects. *Proceedings of COLING-ACL 2006*.

Habash, N., Rambow, O., and Kiraz, G. (2005). Morphological analysis and generation for Arabic dialects. *Proceedings of the Workshop on Computational Approaches to Semitic Languages (ACL '05)*.

Hulden, M. (2008). Regular expressions and predicate logic in finite-state language processing.

In Piskorski, J., Watson, B., and Yli-Jyrä, A., editors, *Proceedings of FSMNLP 2008*.

Kataja, L. and Koskenniemi, K. (1988). Finite-state description of Semitic morphology: a case study of ancient Akkadian. In *COLING '88*, pages 313–315.

Kay, M. (1987). Nonconcatenative finite-state morphology. In *Proceedings of the third conference on European chapter of the Association for Computational Linguistics*, pages 2–10. Association for Computational Linguistics.

Kiraz, G. A. (1994). Multi-tape two-level morphology: A case study in Semitic non-linear morphology. In *COLING '94*, pages 180–186.

Kiraz, G. A. (2000). Multi-tiered nonlinear morphology using multitape finite automata: A case study on Syriac and Arabic. *Computational Linguistics*, 26(1):77–105.

Kiraz, G. A. (2001). *Computational nonlinear morphology: with emphasis on Semitic languages*. Cambridge University Press, Cambridge.

Koskenniemi, K. (1983). *Two-level morphology: A general computational model for word-form recognition and production*. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

Lavie, A., Itai, A., and Ornan, U. (1988). On the applicability of two level morphology to the inflection of Hebrew verbs. In *Proceedings of ALLC III*, pages 246–260.

McCarthy, J. J. (1981). A Prosodic Theory of Nonconcatenative Morphology. *Linguistic Inquiry*, 12(3):373–418.

van Noord, G. (2000). *FSA 6 Reference Manual*.

Wehr, H. (1979). *A Dictionary of Modern Written Arabic*. Spoken Language Services, Inc., Ithaca, NY.

Yli-Jyrä, A. and Koskenniemi, K. (2004). Compiling contextual restrictions on strings into finite-state automata. *The Eindhoven FASTAR Days Proceedings*.

Yona, S. and Wintner, S. (2008). A finite-state morphological grammar of Hebrew. *Natural Language Engineering*, 14(2):173–190.

## 9 Appendix

The practical implementation described in the paper was done with the freely available (GNU Licence) *foma* finite-state toolkit.[9]. However, all of the techniques used are available in other toolkits as well, such as *xfst* (Beesley and Karttunen, 2003), or *fsa* (van Noord, 2000)), and translation of the notation should be straightforward.

The functions for populating the tapes in section 4.2, were defined in *foma* as follows:

```
TapeL(X,Y) =
[[Y ∘ [[0×\X \X]* [0×X] ∑]*]₂
[X ε|\X \X]*]
TapeM(X,Y) = [[Y ∘ [0×[\X \X|X ε]]*
[0×\X \X]* [0×X] ∑]*]₂ [X ε|\X \X]*]
TapeA(X,Y) = [[Y ∘
[0×\X \X|X ε]* 0×X ∑]*]₂;
```

Here, `TapeX` is a function of two variables, `X` and `Y`. Transducer composition is denoted by $\circ$, cross-product by $\times$, the lower projection of a relation by $L_2$, and union by $|$. Brackets indicate grouping and $\Sigma$ any symbol. The notation $\backslash X$ denotes any single symbol, except $X$. The symbol $\varepsilon$ here is the special 'blank' symbol used to pad the tapes and keep them synchronized.

[9] `http://foma.sourceforge.net`