Finite-State Registered Automata for Non-Concatenative Morphology

Yael Cohen-Sygal*
University of Haifa

Shuly Wintner†
University of Haifa

We introduce finite-state registered automata (FSRAs), a new computational device within the framework of finite-state technology, specifically tailored for implementing non-concatenative morphological processes. This model extends and augments existing finite-state techniques, which are presently not optimized for describing this kind of phenomena. We first define the model and discuss its mathematical and computational properties. Then, we provide an extended regular language whose expressions denote FSRAs. Finally, we exemplify the utility of the model by providing several examples of complex morphological and phonological phenomena, which are elegantly implemented with FSRAs.

1. Introduction

Finite-state (FS) technology has been considered adequate for describing the morphological processes of the world's languages since the pioneering works of Koskenniemi (1983) and Kaplan and Kay (1994). Several toolboxes provide extended regular expression description languages and compilers of the expressions to finite-state automata (FSAs) and transducers (FSTs) (Karttunen et al. 1996; Mohri 1996; van Noord and Gerdemann 2001a). While FS approaches to most natural languages have generally been very successful, it is widely recognized that they are less suitable for non-concatenative phenomena; in particular, FS techniques are assumed not to be able to efficiently account for the non-concatenative word formation processes that Semitic languages exhibit (Lavie et al. 1988).

While much of the inflectional morphology of Semitic languages can be rather straightforwardly described using concatenation as the primary operation, the main word formation process in such languages is inherently non-concatenative. The standard account describes words in Semitic languages as combinations of two morphemes: a root and a pattern.[1] The root consists of consonants only, by default three (although longer roots are known). The pattern is a combination of vowels and, possibly, consonants too, with "slots" into which the root consonants can be inserted. Words are created by **interdigitating** roots into patterns: The first consonant of the root is inserted into the first consonantal slot of the pattern, the second root consonant fills the second slot, and the third fills the last slot. After the root combines with the pattern, some

* Department of Computer Science, University of Haifa, 31905 Haifa, Israel. E-mail: yaelc@cs.haifa.ac.il.
† Department of Computer Science, University of Haifa, 31905 Haifa, Israel. E-mail: shuly@cs.haifa.ac.il.

1 An additional morpheme, **vocalization,** is used to abstract the pattern further; for the present purposes, this distinction is irrelevant.

Submission received: 17 August 2004; revised submission received: 15 June 2005; accepted for publication: 26 September 2005.

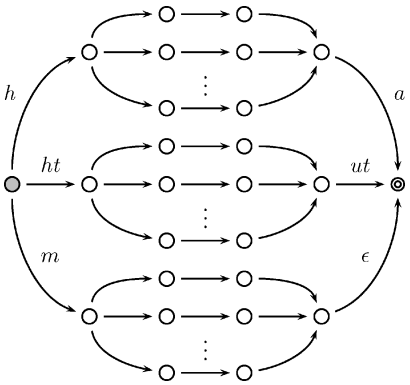

Figure 1
Naïve FSA with duplicated paths.

morpho-phonological alternations take place, which may be non-trivial but are mostly concatenative.

The major problem that we tackle in this work is **medium-distance dependencies,** whereby some elements that are related to each other in some deep-level representation (e.g., the consonants of the root) are separated on the surface. While these phenomena do not lie outside the descriptive power of FS systems, naïvely implementing them in existing finite-state calculi is either impossible or, at best, results in large networks that are inefficient to process, as the following examples demonstrate.

Example 1
We begin with a simplified problem, namely accounting for circumfixes. Consider three Hebrew patterns: *ha*□□*a*□*a*, *hit*□*a*□*a*□*ut*, and *mi*□□*a*□, where the empty boxes indicate the slots in the patterns into which the consonants of the roots are inserted. Hebrew orthography[2] dictates that these patterns be written *h*□□□*a*, *ht*□□□*ut*, and *m*□□□, respectively, i.e., the consonants are inserted into the '□' slots as one unit (i.e., the patterns can be viewed as circumfixes). An automaton that accepts all the possible combinations of three-consonant stems and these three circumfixes is illustrated in Figure 1.[3] Given r stems and p circumfixes, the number of its states is $(2r + 2)p + 2$, i.e., increases linearly with the number of stems and circumfixes. The number of arcs in this automaton is $3rp + 2p$, i.e, also $O(rp)$. Evidently, the three basic different paths that result from the three circumfixes have the same body, which encodes the stems. An attempt to avoid the duplication of paths is represented by the automaton of Figure 2, which accepts the language denoted by the regular expression $(ht + h + m)(root)(ut + a + \epsilon)$. The number of states here is $2r + 4$, i.e., is independent of the number of circumfixes. The number of arcs is $(3r + 2p)$, that is, $O(r + p)$, and thus, the complexity of the number of arcs is also reduced. Obviously, however, such an automaton over-generates by accepting also invalid words such as *m*□□□*ut*. In other words, it ignores the dependencies which hold between prefixes and suffixes of the same circumfix. Since finite-state devices have no

2 Many of the vowels are not explicitly depicted in the Hebrew script.
3 This is an over-simplified example; in practice, the process of combining roots with patterns is highly idiosyncratic, like other derivational morphological processes.

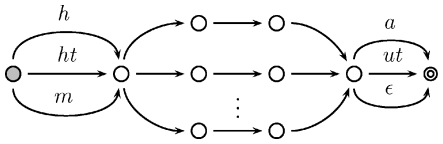

Figure 2
Over-generating FSA.

memory, save for the states, there is no simple and space-efficient way to account for such dependencies.

Example 2
Consider now a representation of Hebrew where all vowels are explicit, e.g., the pattern *hit*□*a*□*e*□. Consider also the roots *r.g.z, b.$.l*, and *g.b.r*. The consonants of a given root are inserted into the '□' slots to obtain bases such as *hitragez, hitba$el*, and *hitgaber*. The finite state automaton of Figure 3 is the minimized automaton accepting the language; it has fifteen states. If the number of three letter roots is r, then a general automaton accepting the combinations of the roots with this pattern will have $4r + 3$ states and $5r + 1$ arcs. Notice the duplicated arcs which stem from copying the pattern in the different paths.

Example 3
Another non-concatenative process is **reduplication:** The process in which a morpheme or part of it is duplicated. Full reduplication is used as a pluralization process in Malay and Indonesian; partial reduplication is found in Chamorro to indicate intensivity. It can also be found in Hebrew as a diminutive formation of nouns and adjectives:

keleb	klablab	$apan	$panpan	zaqan	zqanqan	$axor	$xarxar
dog	puppy	rabbit	bunny	beard	goatee	black	dark

qatan	qtantan
little	tiny

Let Σ be a finite alphabet. The language $L = \{ww \mid w \in \Sigma^*\}$ is known to be trans-regular, therefore no finite-state automaton accepts it. However, the language $L_n = \{ww \mid w \in \Sigma^*, |w| = n\}$ for some constant n is regular. Recognizing L_n is a finite approximation of the general problem of recognizing L. The length of the words in natural languages can in most cases be bounded by some $n \in \mathbb{N}$, hence the amount of reduplication in natural languages is practically limited. Therefore, the descriptive power of L_n is sufficient for the amount of reduplication in natural languages (by

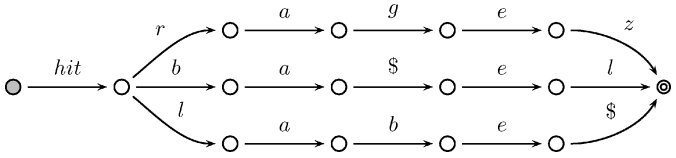

Figure 3
FSA for the pattern hit□a□e□.

constructing L_n for a small number of different ns). An automaton that accepts L_n can be constructed by listing a path for each accepted string (since Σ and n are finite, the number of words in L_n is finite). The main drawback of such an automaton is the growth in its size as $|\Sigma|$ and n increase: The number of strings in L_n is $|\Sigma|^n$. Thus, finite-state techniques can account for limited reduplication, but the resulting networks are space-inefficient.

As a final, non-linguistic, motivating example, consider the problem of n-bit incrementation, introduced by Kornai (1996).

Example 4
The goal of this example is to construct a transducer over $\Sigma = \{0, 1\}$ whose input is a 32 bit binary number and whose output is the result of adding 1 to the input. A transducer that performs addition by 1 on binary numbers has only 5 states and 12 arcs,[4] but this transducer is neither sequential nor sequentiable. The problem is that since the input is scanned left to right but the carry moves right to left, the output of the first bit has to be delayed, possibly even until the last input bit is scanned. Thus, for an n-bit binary incrementor, 2^n disjunctions have to be considered, and therefore a minimized transducer has to assign a separate state to each combination of bits, resulting in 2^n states and a similar number of transitions.

In this work we propose a novel FS model which facilitates the expression of medium-distance dependencies such as interdigitation and reduplication in an efficient way. Our main motivation is theoretical, i.e., reducing the complexity of the number of states and arcs in the networks; we show that these theoretical contributions result in practical improvements. In Section 3 we define the model formally, show that it is equivalent to FSAs and define many closure properties directly.[5] We then (Section 4) define a regular expression language for denoting FSRAs. In Section 5 we provide dedicated regular expression operators for some non-concatenative phenomena and exemplify the usefulness of the model by efficiently accounting for the motivating examples. In Section 6 we extend FSRAs to transducers. The model is evaluated through an actual implementation in Section 7. We conclude with suggestions for future research.

2. Related Work

In spite of the common view that FS technology is in general inadequate for describing non-concatenative processes, several works address the above-mentioned problems in various ways. We summarize existing approaches in this section.

Several works examine the applicability of traditional two-level systems for implementing non-concatenative morphology. Two-Level Morphology was used by Kataja and Koskenniemi (1988) to create a rule system for phonological and morphophonological alternations in Akkadian, accounting for word inflection and regular verbal derivation. As this solution effectively defines lexical representations of word-forms, its main disadvantage is that the final network is the naïve one, suffering from the space complexity problems discussed above. Lavie et al. (1988) examine the applicability of Two-

4 A complete explanation of the construction can be found in http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsexamples.html#Add1.

5 Many of the formal proofs and constructions, especially the ones that are similar to the case of standard FSAs, are suppressed; see Cohen-Sygal (2004) for the complete proofs and constructions.

Level Morphology to the description of Hebrew Morphology, and in particular to verb inflection. Their lexicon consists of three parts: verb primary bases (the past tense, third person, singular, masculine), verb prefixes, and verb suffixes. They attempt to describe Hebrew verb inflection as a concatenation of prefix+base+suffix, implementable by the Two-Level model. However, they conclude that "The Two-Level rules are not the natural way to describe . . . verb inflection process. The only alternative choice . . . is to keep all bases . . . it seems wasteful to save all the secondary bases of verbs of the same pattern."

Other works deal with non-concatenative morphology by extending ordinary FSAs without extending their expressivity. The traditional two-level model of Koskenniemi (1983) is expanded into n-tape automata by Kiraz (2000), following the insight of Kay (1987) and Kataja and Koskenniemi (1988). The idea is to use more than two levels of expression: The surface level employs one representation, but the lexical form employs multiple representations (e.g., root, pattern) and therefore can be divided into different levels, one for each representation. Elements that are separated on the surface (such as the root's consonants) are adjacent on a particular lexical level. For example, to describe circumfixation using this model, a 4-tape automaton of the form $\langle$*surface, PR pattern, circumfix, stem*$\rangle$ is constructed, so that each word is represented by 4 levels. The surface level represents the final form of the word. The *PR pattern* is the pattern in which the stem and the circumfix are combined (P represents the circumfix's position and R the root letter's position), e.g., *PRRRP*. The circumfix and stem levels represent the circumfix and the stem respectively.

For example, combining the Hebrew stem *pqd* with the circumfix *ht-ut* will have the 4-level representation shown in Figure 4. Notice that the symbols representing the circumfix in the PR pattern level (i.e., the occurrences of the symbol 'P'), the circumfix symbols in the circumfix level, and the circumfix symbols in the surface level are located in correlating places in the different levels. The same holds for the stem symbols. In this way, it is clear which symbols of the surface word belong to the circumfix, which belong to the stem, and how they combine together to create the final form of the word. The 4-tape automaton of Figure 5 accepts all the combinations created by circumfixing roots with the three circumfixes of Example 1. Each arc is attributed with a quadruplet, consisting of four correlating symbols in the four levels. Notice that as in FSAs, the paths encoding the roots are duplicated for each circumfix, so that this automaton is as space-inefficient as ordinary FSAs. Kiraz (2000) does not discuss the space complexity of this model, but the number of states still seems to increase with the number of roots and patterns. Moreover, the n-tape model requires specification of dependencies between symbols in different levels, which may be non-trivial.

Walther (2000a) suggests a solution for describing natural language reduplication using finite-state methods. The idea is to enrich finite-state automata with three new operations: *repeat*, *skip*, and *self loops*. *Repeat* arcs allow moving backwards within a string and thus repeat a part of it (to model reduplication). *Skip* arcs allow moving forwards in a string while suppressing the spell out of some of its letters; *self loop* arcs model infixation. In Walther (2000b), the above technique is used to describe Temiar

Stem	ϵ	p	q	d	ϵ
Circumfix	ht	ϵ	ϵ	ϵ	ut
PR pattern	P	R	R	R	P
Surface	ht	p	q	d	ut

Figure 4
4-tape representation for the Hebrew word htpqdut.

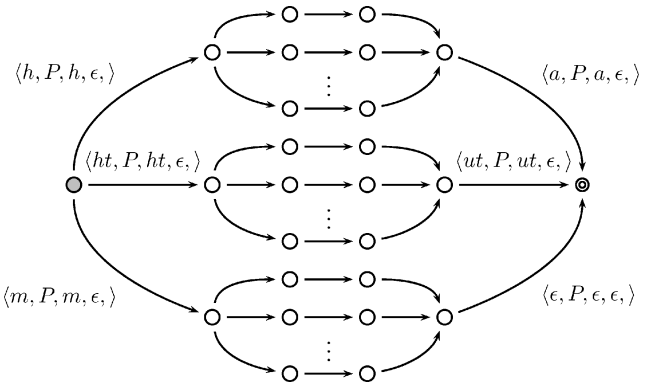

Figure 5
4-tape automaton for circumfixation example.

reduplication, but no complexity analysis of the model is given. Moreover, this technique does not seem to be able to describe interdigitation.

Beesley and Karttunen (2000) describe a technique, called *compile-replace*, for constructing FSTs, which involves reapplying the regular-expression compiler to its own output. The compile-replace algorithm facilitates a compact definition of non-concatenative morphological processes, but since such expressions compile to the naïve networks, no space is saved. Furthermore, this is a compile-time mechanism rather than a theoretical and mathematically founded solution.

Other works extend the FS model by enabling some sort of context-sensitivity. Blank (1985, 1989) presents a model, called Register Vector Grammar, introducing context-sensitivity by representing the states and transitions of finite-state automata as ternary-valued vectors, which need not be fully specified. No formal properties of this model are discussed. In a similar vein, Kornai (1996) introduces vectorized finite-state automata, where both the states and the transitions are represented by vectors of elements of a partially ordered set. The vectors are manipulated by operations of *unification* and *overwriting*. The vectors need not be fully determined, as some of the elements can be unknown (free). In this way information can be moved through the transitions by the overwriting operation and traversing these transitions can be sanctioned through the unification operation. As one of the examples of the advantages of this model, Kornai (1996) shows it can efficiently solve the problem of 32-bit binary incrementor (Example 4). Using vectorized finite-state automata, a 32-bit incrementor is constructed where first, using overwriting, the input is scanned and stored in the vectors, and then, using unification, the result is calculated where the carry can be computed from right to left. We return to this example in example 6, where we show how our own model can solve it efficiently. The formalism presented by Kornai (1996) allows a significant reduction in the network size, but its main disadvantage lies in the fact that it significantly deviates from the standard methodology of developing finite-state devices, and integration of vectorized automata with standard ones remains a challenge. Moreover, it is unclear how, for a given problem, the corresponding network should be constructed: Programming with vectorized automata seems to be unnatural, and no regular expression language is provided for them.

A more general approach to the design of finite-state machines is introduced by Mohri, Pereira, and Riley (2000). They introduce an object-oriented library for manipu-

lating finite-state devices that is based on the algebraic concepts of rational power series and semirings. This approach facilitates a high degree of generality as the algorithms are defined for the general algebraic notions, which can then be specialized according to the needs of the user. They exemplify the usefulness of this library by showing how to specialize it for the manipulation of weighted automata and transducers. Our work can be seen as another specialization of this general approach, tailored for ideally dealing with our motivating examples.

Several works introduce the notion of registers, whether for solving similar problems or motivated by different considerations. Krauwer and des Tombe (1981) refer to transducers with a finite number of registers when comparing transducers and context free grammars with respect to their capabilities to describe languages. They sketch a proof showing that such transducers are equivalent to ordinary finite-state transducers. However, they never formally define the model and do not discuss its ability to efficiently implement non-concatenative natural languages phenomena. Moreover, they do not show how the closure properties can be implemented directly on these registered transducers, and do not provide any regular language denoting such transducers.

Motivated by different considerations, Kaminski and Francez (1994) present a computational model which extends finite state automata to the case of infinite alphabets. This model is limited to recognizing only regular languages over infinite alphabets while maintaining closure under Kleene star and boolean operations, with the exception of closure under complementation. The familiar automaton is augmented with registers, used to store alphabet symbols, whose number is fixed for each automaton and can vary from one automaton to another. The model is designed to deal with infinite alphabets, and therefore it cannot distinguish between different symbols; it can identify different patterns but cannot distinguish between different symbols in the pattern as is often needed in natural languages. Our solution is reminiscent of Kaminski and Francez (1994) in the sense that it augments finite-state automata with finite memory (registers) in a restricted way, but we avoid the above-mentioned problem. In addition, our model supports a register alphabet that differs from the language alphabet, allowing the information stored in the registers to be more meaningful. Moreover, our transition relation is a more simplified extension of the standard one in FSAs, rendering our model a conservative extension of standard FSAs and allowing simple integration of existing networks with networks based on our model.

Finally, Beesley (1998) directly addresses medium-distance dependencies between separated morphemes in words. He proposes a method, called *flag diacritics*, which adds features to symbols in regular expressions to enforce dependencies between separated parts of a string. The dependencies are forced by different kinds of unification actions. In this way, a small amount of finite memory is added, keeping the total size of the network relatively small. Unfortunately, this method is not formally defined, nor are its mathematical and computational properties proved. Furthermore, flag diacritics are manipulated at the level of the extended regular expressions, although it is clear that they are compiled into additional memory and operators in the networks themselves. The presentations of Beesley (1998) and Beesley and Karttunen (2003) do not explicate the implementation of such operators and do not provide an analysis of their complexity. Our approach is similar in spirit, but we provide a complete mathematical and computational analysis of such extended networks, including a proof that the model is indeed regular and constructions of the main closure properties. We also provide dedicated regular expression operations for non-concatenative processes and show

how they are compiled into extended networks, thereby accounting for the motivating examples.

3. Finite-state Registered Automata

We define a new model, **finite-state registered automata** (FSRA), aimed at facilitating the expression of various non-concatenative morphological phenomena in an efficient way. The model augments finite-state automata with finite memory (registers) in a restricted way that saves space but does not add expressivity. The number of registers is finite, usually small, and eliminates the need to duplicate paths as it enables the automaton to "remember" a finite number of symbols. In addition to being associated with an alphabet symbol, each arc is also associated with an action on the registers. There are two kinds of actions, **read** and **write**. The read action, denoted R, allows traversing an arc only if a designated register contains a specific symbol. The write action, denoted W, allows traversing an arc while writing a specific symbol into a designated register. In this section we define FSRAs and show that they are equivalent to standard FSAs (Section 3.1). We then directly define several closure operations over FSRAs (Section 3.2) and provide some optimizations in Section 3.3. We conclude this section with a discussion of minimization (Section 3.4).

3.1 Definitions and Examples

Definition
A **finite-state registered automaton (FSRA)** is a tuple $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F \rangle$, where:

- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- Σ is a finite alphabet (the language alphabet).
- $n \in \mathbb{N}$ (indicating the number of registers).
- Γ is a finite alphabet including the symbol '#' (the registers alphabet). We use meta-variables u_i, v_i to range over Γ and u, v to range over Γ^n.
- The initial content of the registers is $\#^n$, meaning that the initial value of all the registers is 'empty'.
- $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma \times Q$ is the transition relation. The intuitive meaning of δ is as follows:
  - $(s, \sigma, R, i, \gamma, t) \in \delta$ where $i > 0$ implies that if A is in state s, the input symbol is σ, and the content of the i-th register is γ, then A may enter state t.
  - $(s, \sigma, W, i, \gamma, t) \in \delta$ where $i > 0$ implies that if A is in state s and the input symbol is σ, then the content of the i-th register is changed into γ (overwriting whatever was there before) and A may enter state t.
  - $(s, \sigma, R, 0, \#, t)$ implies that if A is in state s and the input symbol is σ, then A may enter state t. Notice that the content of register number 0 is always #. We use the shorthand notation (s, σ, t) for such transitions.
- $F \subseteq Q$ is the set of final states.

Definition
A configuration of A is a pair (q, u), where $q \in Q$ and $u \in \Gamma^n$ (q is the current state and u represents the registers content). The set of all configurations of A is denoted by Q^c. The pair $q_0^c = (q_0, \#^n)$ is called the **initial configuration,** and configurations with the first component in F are called **final configurations.** The set of final configurations is denoted by F^c.

Definition
Let $u = u_0u_1 \dots u_{n-1}$ and $v = v_0v_1 \dots v_{n-1}$. Given a symbol $\alpha \in \Sigma \cup \{\epsilon\}$ and an FSRA A, we say that a configuration (s, u) **produces** a configuration (t, v), denoted $(s, u) \vdash_{\alpha,A} (t, v)$, iff either one of the following holds:

- There exists i, $0 \leq i \leq n-1$, and there exists $\gamma \in \Gamma$, such that $(s, \alpha, R, i, \gamma, t) \in \delta$ and $u = v$ and $u_i = v_i = \gamma$; or
- There exists i, $0 \leq i \leq n-1$, and there exists $\gamma \in \Gamma$, such that $(s, \alpha, W, i, \gamma, t) \in \delta$ and for all k, $k \in \{0, 1, ..., n-1\}$, such that $k \neq i$, $u_k = v_k$ and $v_i = \gamma$.

Informally, a configuration c_1 produces a configuration c_2 iff the automaton can move from c_1 to c_2 when scanning the input α (or without any input, when $\alpha = \epsilon$) in one step. If the register operation is R, then the contents of the registers in the two configurations must be equal, and in particular the contents of the designated register in the two configurations should be the expected symbol (γ). If the register operation is W, then the contents of the registers in the two configurations is equal except for the designated register, whose contents in the produced configuration should be the expected symbol (γ).

Definition
A **run** of A on w is a sequence of configurations $c_0, ..., c_r$ such that $c_0 = q_0^c$, $c_r \in F^c$, and for every k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k,A} c_k$ and $w = \alpha_1 ... \alpha_r$. An FSRA A **accepts** a word w if there exists a run of A on w. Notice that $|w|$ may be less than r since some of the α_i may be ϵ. The **language** recognized by an FSRA A, denoted by $L(A)$, is the set of words over Σ^* accepted by A.

Example 5
Consider again example 1. We construct an efficient FSRA accepting all and only the possible combinations of stems and circumfixes. If the number of stems is r, we define an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, 2, \delta, \{q_f\}\rangle$ where:

- $Q = \{q_0, q_1, \dots, q_{2r+2}, q_f\}$
- $\Sigma = \{a, b, c, \dots, z, ht, ut\}$
- $\Gamma = \{ht\square\square\square ut, h\square\square\square a, m\square\square\square, \#\}$
- $\delta = \{(q_0, ht, W, 1, ht\square\square\square ut, q_1), (q_0, h, W, 1, h\square\square\square a, q_1)\}$
  $\cup$
  $\{(q_0, m, W, 1, m\square\square\square, q_1), (q_{2r+2}, ut, R, 1, ht\square\square\square ut, q_f)\}$
  $\cup$
  $\{(q_{2r+2}, a, R, 1, h\square\square\square a, q_f), (q_{2r+2}, \epsilon, R, 1, m\square\square\square, q_f)\}$
  $\cup$
  $\{(q_1, \alpha_1, q_i), (q_i, \alpha_2, q_{i+1}), (q_{i+1}, \alpha_3, q_{2r+2}) \mid 2 \leq i \leq 2r$ and
  $\alpha_1\alpha_2\alpha_3$ is the i-th stem$\}$.

This automaton is shown in Figure 6. The number of its states is $2r + 4$ (like the FSA of Figure 2), that is, $O(r)$, and in particular independent of the number of circumfixes. The number of arcs is also reduced from $O(r \times p)$, where p indicates the number of circumfixes, to $O(r + p)$.

Example 6
Consider again example 2. The FSRA of Figure 7 also accepts the same language. This automaton has seven states and will have seven states for any number of roots. The number of arcs is also reduced to $3r + 3$.

Next, we show that finite-state registered automata and standard finite state automata recognize the same class of languages. Trivially, every finite-state automaton has an equivalent FSRA: Every FSA is also an FSRA since every transition (s, σ, t) in an FSRA is a shorthand notation for $(s, \sigma, R, 0, \#, t)$. The other direction is also simple.

Theorem 1
Every FSRA has an equivalent finite-state automaton.

We prove this by constructing an equivalent FSA *to a given FSRA*. The construction is based on the fact that in FSRAs the number of registers is finite, as are the sets Γ and Q, the register alphabet and states, respectively. Hence the number of configurations is finite. The FSA's states are the configurations of the FSRA, and the transition function simulates the 'produces' relation. Notice that this relation holds between configurations depending on Σ only, similarly to the transition function in an FSA. The constructed FSA is non-deterministic, with possible ϵ-moves. The formal proof is suppressed.

The number of configurations in A is $|Q| \times |\Gamma|^n$, hence the growth in the number of states when constructing A' from A might be in the worst case exponential in the number of registers. In other words, the move from FSAs to FSRAs can yield an exponential *reduction* in the size of the network. As we show below, the reduction in the number of states can be even more dramatic.

The FSRA model defined above allows only one register operation on each transition. We extend it to allow up to k register operations on each transition, where k is determined for each automaton separately. The register operations are defined as a sequence (rather than a set), in order to allow more than one operation on the same

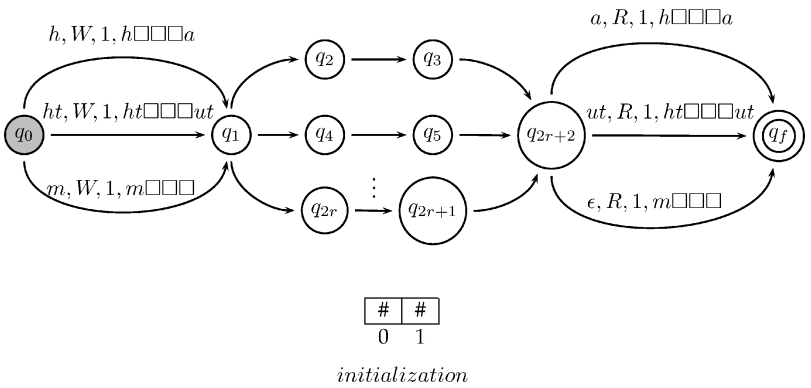

Figure 6
FSRA for circumfixation.

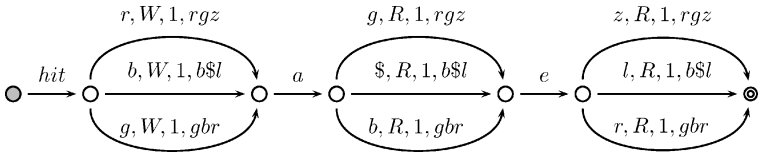

Figure 7
FSRA for the pattern hit□a□e□.

register over one transition. This extension allows further reduction of the network size for some automata as well as other advantages that will be discussed presently.

Definition
An **order-k finite-state registered automaton (FSRA-k)** is a tuple $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$, where:

- $Q, q_0, \Sigma, \Gamma, n, F$ and the initial content of the registers are as before.
- $k \in \mathbb{N}$ (indicating the maximum number of register operations allowed on each arc).
- Let $Actions_n^\Gamma = \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma$. Then

$$\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \left(\bigcup_{j=1}^{k} \left\{ \langle a_1, ..., a_j \rangle \mid \text{for all } i,\ 1 \leq i \leq j,\ a_i \in Actions_n^\Gamma \right\} \right) \times Q$$

  is the transition relation. δ is extended to allow each transition to be associated with a series of up to k operations on the registers. Each operation has the same meaning as before.

The register operations are executed in the order in which they are specified. Thus, $(s, \sigma, \langle a_1, ..., a_i \rangle, t) \in \delta$ where $i \leq k$ implies that if A is in state s, the input symbol is σ and all the register operations $a_1, ..., a_i$ are executed successfully, then A may enter state t.

Definition
Given $a \in Actions_n^\Gamma$ we define a relation over Γ^n, denoted $u \Vdash_a v$ for $u, v \in \Gamma^n$. We define $u \Vdash_a v$ where $u = u_0 \ldots u_{n-1}$ and $v = v_0 \ldots v_{n-1}$ iff the following holds:

- if $a = (R, i, \gamma)$ for some i, $0 \leq i \leq n-1$ and for some $\gamma \in \Gamma$, then $u = v$ and $u_i = v_i = \gamma$.
- if $a = (W, i, \gamma)$ for some i, $0 \leq i \leq n-1$ and for some $\gamma \in \Gamma$, then for all $k \in \{0, 1, \ldots, n-1\}$ such that $k \neq i$, $u_k = v_k$ and $v_i = \gamma$.

This relation is extended to series over $Actions_n^\Gamma$. Given a series $\langle a_1, ..., a_p \rangle \in (Actions_n^\Gamma)^p$ where $p \in \mathbb{N}$, we define a relation over Γ^n denoted $u \Vdash_{\langle a_1, \ldots, a_p \rangle} v$ for $u, v \in \Gamma^n$. We define $u \Vdash_{\langle a_1, \ldots, a_p \rangle} v$ iff the following holds:

- if $p = 1$, then $u \Vdash_{a_1} v$.
- if $p > 1$, then there exists $w \in \Gamma^n$ such that $u \Vdash_{a_1} w$ and $w \Vdash_{\langle a_2, \ldots, a_p \rangle} v$.

Definition
Let $u, v \in \Gamma^n$. Given a symbol $\alpha \in \Sigma \cup \{\epsilon\}$ and an FSRA-k A, we say that a configuration (s, u) **produces** a configuration (t, v), denoted $(s, u) \vdash_{\alpha,A} (t, v)$, iff there exist $\langle a_1, \ldots, a_p \rangle \in (Actions_n^{\Gamma})^p$ for some $p \in \mathbb{N}$ such that $(s, \alpha, \langle a_1, \ldots, a_p \rangle, t) \in \delta$ and $u \Vdash_{\langle a_1,\ldots,a_p \rangle} v$.

Definition
A **run** of A on w is a sequence of configurations $c_0, \ldots, c_r$ such that $c_0 = q_0^c$, $c_r \in F^c$, and for every l, $1 \leq l \leq r$, $c_{l-1} \vdash_{\alpha_l,A} c_l$ and $w = \alpha_1 \ldots \alpha_r$. An FSRA-k A **accepts** a word w if there exists a run of A on w. The **language** recognized by an FSRA-k A, denoted by $L(A)$, is the set of words over Σ^* accepted by A.

Example 7
Consider the Arabic nouns *qamar* (moon), *kitaab* (book), *$ams* (sun), and *daftar* (notebook). The definite article in Arabic is the prefix *al*, which is realized as *al* when preceding most consonants; however, the 'l' of the prefix assimilates to the first consonant of the noun when the latter is 'd', '$', etc. Furthermore, Arabic distinguishes between definite and indefinite case markers. For example, nominative case is realized as the suffix *u* on definite nouns, *un* on indefinite nouns. Examples of the different forms of Arabic nouns are:

word	nominative definite	nominative indefinite
qamar	'alqamaru	qamarun
kitaab	'alkitaabu	kitaabun
$ams	'a$$amsu	$amsun
daftar	'addaftaru	daftarun

The FSRA-2 of Figure 8 accepts all the nominative definite and indefinite forms of the above nouns. In order to account for the assimilation, register 2 stores information about the actual form of the definite article. Furthermore, to ensure that definite nouns occur with the correct case ending, register 1 stores information of whether or not a definite article was seen.

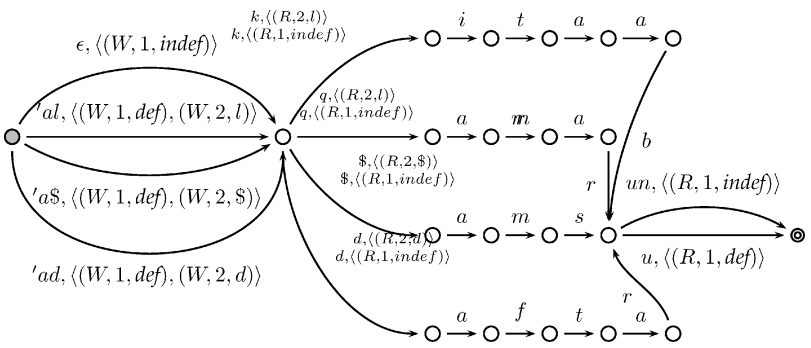

Figure 8
FSRA-2 for Arabic nominative definite and indefinite nouns.

FSRA-k and FSRAs recognize the same class of languages. Trivially, every FSRA has an equivalent FSRA-k: Every FSRA is an FSRA-k for $k = 1$. The other direction is also simple.

Theorem 2
Every FSRA-k has an equivalent FSRA.

We show how to construct an equivalent FSRA (or FSRA-1) A' given an FSRA-k A. Each transition in A is replaced by a series of transitions in A′, each of which performs one operation on the registers. The first transition in the series deals with the new input symbol and the rest are ϵ-transitions. This construction requires additional states to enable the addition of transitions. Each transition in A that is replaced requires the addition of as many states as the number of register operations performed on this transition minus one. The formal construction is suppressed.

In what follows, the term FSRA will be used to denote FSRA-k. Simple FRSA will be referred to as FSRA-1. For the sake of emphasis, however, the term FSRA-k will still be used in some cases.

FSRA is a very space-efficient finite-state device. The next theorem shows how ordinary finite-state automata can be encoded efficiently by the FSRA-2 model. Given a finite-state automaton A, an equivalent FSRA-2 A' is constructed. A' has three states and two registers (in fact, only one register is used since register number 0 is never addressed). One state functions as a representative for the final states in A, another one functions as a representative for the non-final states in A, and the third as an initial state. The register alphabet consists of the states of A and the symbol '#'. Each arc in A has an equivalent arc in A' with two register operations. The first reads the current state of A from the register and the second writes the new state into the register. If the source state of a transition in A is a final state, then the source state of the corresponding transition in A' will be the final states representative; if the source state of a transition in A is a non-final state, then the source state of the corresponding transition in A' will be the non-final states representative. The same holds also for the target states. The purpose of the initial state is to write the start state of A into the register. In this way A' simulates the behavior of A. Notice that the number of arcs in A' equals the number of arcs in A plus one, i.e., while FSRAs can dramatically reduce the number of states, compared to standard FSAs, a reduction in the number of arcs is not guaranteed.

Theorem 3
Every finite-state automaton has an equivalent FSRA-2 with three states and two registers.

Proof 1
Let $A = \langle Q, q_0, \Sigma, \delta, F\rangle$ be an FSA and let $f : Q \to \left\{q_f, q_{nf}\right\}$ be a total function defined by

$$f(q) = \begin{cases} q_f & q \in F \\ q_{nf} & q \notin F \end{cases}$$

Construct an FSRA-2 $A' = \langle Q', q'_0, \Sigma', \Gamma', 2, 2, \delta', F' \rangle$, where:

- $Q' = \{q'_0, q_{nf}, q_f\}$. q'_0 is the initial state, q_f is the final states representative, and q_{nf} is the non-final states representative
- $\Sigma' = \Sigma$
- $\Gamma = Q \cup \{\#\}$
- $F' = \{q_f\}$
- $\delta' = \{(f(s), \sigma, \langle (R, 1, s), (W, 1, t) \rangle, f(t)) \mid (s, \sigma, t) \in \delta\}$
  $\cup$
  $\{(q'_0, \epsilon, \langle (W, 1, q_0) \rangle, f(q_0))\}$

The formal proof that $L(A) = L(A')$ is suppressed. ■

3.2 Closure Properties

The equivalence shown in the previous section between the classes of languages recognized by finite-state automata and finite-state registered automata immediately implies that finite-state registered automata maintain the closure properties of regular languages. Applying the regular operations to finite-state registered automata can be easily done by converting them first into finite-state automata. However, as shown above, such a conversion may result in an exponential increase in the size of the automaton, invalidating the advantages of this model. Therefore, we show how some of these operations can be defined directly for FSRAs. The constructions are mostly based on the standard constructions for FSAs with some essential modifications. In what follows, let $A_1 = \langle Q_1, q^1_0, \Sigma_1, \Gamma_1, n_1, k_1, \delta_1, F_1 \rangle$ and $A_2 = \langle Q_2, q^2_0, \Sigma_2, \Gamma_2, n_2, k_2, \delta_2, F_2 \rangle$ be finite-state registered automata.

3.2.1 Union. Two FSRAs, A_1, A_2, are unioned into an FSRA A in the same way as in FSAs: by adding a new initial state and connecting it with ϵ-arcs to each of the (former) initial states of A_1, A_2. The number of registers and the maximal number of register operations per arc in A is the maximum of the corresponding values in A_1, A_2. Notice that in any specific run of A, the computation goes through just one of the original automata; therefore the same set of registers can be used for strings of $L(A_1)$ or $L(A_2)$ as needed.

3.2.2 Concatenation. We show two different constructions of an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ to recognize $L(A_1) \cdot L(A_2)$. Concatenation in finite-state automata is achieved by leaving only the accepting states of the second automaton as accepting states and adding an ϵ-arc from every accepting state of the first automaton to the initial state of the second automaton. Doing just this in FSRA is insufficient because using the same registers might cause undesired effects: The result might be affected by the content left in the registers after dealing with a substring from $L(A_1)$. Thus, this basic construction is used with care. The first alternative is to employ more registers in the FSRA. In this way when dealing with a substring from $L(A_1)$ the first n_1 registers are used, and when moving to deal with a substring from $L(A_2)$ the next n_2 registers are used. The second alternative is to use additional register operations that clear the content of the registers before handling the next substring from $L(A_2)$. This solution may be less intuitive but will be instrumental for Kleene closure below.

3.2.3 Kleene Closure. The construction is based on the concatenation construction. Notice that it cannot be based on the first alternative (adding registers) due to the fact that the number of iterations in Kleene star is not limited, and therefore the number of registers needed cannot be bounded. Thus, the second alternative is used: Register operations are added to delete the content of registers. The construction is done by turning the initial state into a final one (if it is not already final) and connecting each of the final states to the initial state with an ϵ-arc that is associated with a register operation that deletes the contents of the registers, leaving them ready to handle the next substring.

3.2.4 Intersection. For the intersection construction, assume that A_1 and A_2 are ϵ-free (we show an algorithm for removing ϵ-arcs in Section 3.3.1). The following construction simulates the runs of A_1 and A_2 simultaneously. It is based on the basic construction for intersection of finite-state automata, augmented by a simulation of the registers and their behavior. Each transition is associated with two sequences of operations on the registers, one for each automaton. The number of the registers is the sum of the number of registers in the two automata. In the intersection automaton the first n_1 registers are designated to simulate the behavior of the registers of A_1 and the next n_2 registers simulate the behavior of A_2. In this way a word can be accepted by the intersection automaton iff it can be accepted by each one of the automata separately. Notice that register operations from δ_1 and δ_2 cannot be associated with the same register. This guarantees that no information is lost during the simulation of the two intersected automata.

3.2.5 Complementation. Ordinary FSAs are trivially closed under complementation. However, given an FSA A whose language is $L(A)$, the minimal FSA recognizing the complement of $L(A)$ can be exponentially large. More precisely, for any integer $n > 2$, there exists a non-deterministic finite-state automaton (NFA) with n states A, such that any NFA that accepts the complement of $L(A)$ needs at least 2^{n-2} states (Holzer and Kutrib 2002). We have no reason to believe that FSRAs will demonstrate a different behavior; therefore, we maintain that in the worst case, the best approach for complementing an FSRA would be to convert it into FSA and complement the latter. We therefore do not provide a dedicated construction for this operator.

3.3 Optimizations

3.3.1 ϵ-removal. An ϵ-arc in an FSRA is an arc of the form $(s, \epsilon, \langle\vec{a}\rangle, t)$ where $\vec{a}$ is used as a meta-variable over $\left(Actions_n^{\Gamma}\right)^+$ (i.e., $\vec{a}$ represents a vector of register operations). Notice that this kind of arc might occur in an FSRA by its definition. Given an FSRA that might contain ϵ-arcs, an equivalent FSRA without ϵ-arcs can be constructed. The construction is based on the algorithm for ϵ-removal in finite-state automata, but the register operations that are associated with the ϵ-arc have to be dealt with, and this requires some care. The resulting FSRA has one more state than the original, and some additional arcs may be added, too.

The main problem is ϵ-loops; while these can be easily removed in standard FSAs, here such loops can be associated with register operations which must be accounted for. The number of possible sequences of register operations along an ϵ-loop is unbounded, but it is easy to prove that there are only finitely many *equivalence classes* of such sequences: Two sequences are in the same equivalence class if and only if they have the same effect on the state of the machine; since each machine has a finite number of configurations (see theorem 1), there are only finitely many such equivalence classes.

Therefore, the basic idea behind the construction is as follows: If there exists an ϵ-path from q_1 to q_2 with the register operations $\vec{a}$ over its arcs, and an arc $(q_2, \sigma, \langle\vec{b}\rangle, q_3)$

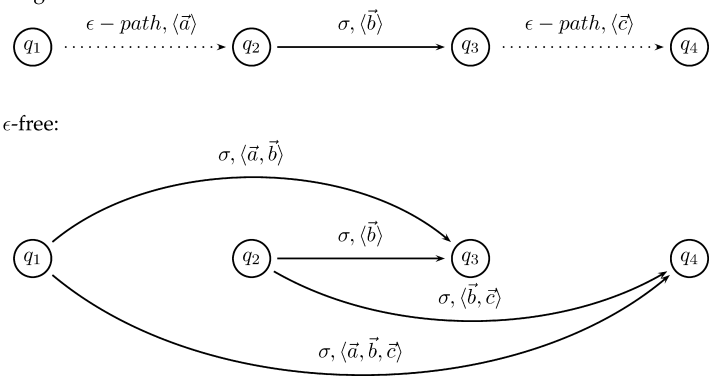

Figure 9
ϵ removal paradigm.

where $\sigma \neq \epsilon$, and an ϵ-path from q_3 to q_4 with the register operations $\vec{c}$ over its arcs, then the equivalent ϵ-free network will include the arcs $(q_2, \sigma, \langle \vec{b} \rangle, q_3)$, $(q_1, \sigma, \langle \vec{a}, \vec{b} \rangle, q_3)$, $(q_2, \sigma, \langle \vec{b}, \vec{c} \rangle, q_4)$, and $(q_1, \sigma, \langle \vec{a}, \vec{b}, \vec{c} \rangle, q_3)$, with all the ϵ-arcs removed. This is illustrated in Figure 9. Notice that if q_1 and q_2 are the same state, then states q_2 and q_3 will be connected by two parallel arcs differing in their associated register operations; the same holds for states q_2 and q_4. Similarly, when q_3 and q_4 are the same state.

In addition to the above changes, special care is needed for the case in which the empty word is accepted by the original automaton. The formal construction is similar in spirit to the ϵ-removal paradigm in weighted automata (Mohri 2000), where weights along an ϵ-path need to be gathered. Therefore, we suppress the formal construction and the proof of its correctness.

3.3.2 Optimizing Register Operations. In FSRAs, traversing an arc depends not only on the input symbol but also on satisfying the series of register operations. Sometimes, a given series of register operations can never be satisfied, and thus the arc to which it is attached cannot be traversed. For example, the series of register operations $\langle (W, 1, a), (R, 1, b) \rangle$ can never be satisfied, hence an arc of the form $(q_1, \sigma, \langle (W, 1, a), (R, 1, b) \rangle, q_2)$ is redundant. In addition, the constructions of Sections 3.2 and 3.3.1 might result in redundant states and arcs that can never be reached or can never lead to a final state. Moreover, in many cases a series of register operations can be minimized into a shorter series with the same effect. For example, the series of register operations $\langle (W, 1, a), (R, 1, a), (W, 1, b) \rangle$ is equal in its effect to the series $\langle (W, 1, b) \rangle$. Therefore, we show an algorithm for optimizing a given FSRA by minimizing the series of register operations over its arcs and removing redundant arcs and states.

For a given FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F \rangle$, we construct an equivalent FSRA $A' = \langle Q, q_0, \Sigma, \Gamma, n, \delta', F \rangle = Opt(A)$, such that δ' is created from δ by removing redundant arcs and by optimizing all the series of register operations. We begin by defining $\left(Actions_n^{\Gamma}\right)^+_{|i}$ as the subset of $\left(Actions_n^{\Gamma}\right)^+$ that consists only of operations over the i-th register. Define a total function $sat_i : \left(Actions_n^{\Gamma}\right)^+_{|i} \longrightarrow \{true, false\}$ by:

$$sat_i(\vec{a}) = \begin{cases} true & \text{if there exist } u, v \in \Gamma^n \text{ such that } u \Vdash_{\vec{a}} v \\ false & \text{otherwise} \end{cases}$$

$sat_i(\vec{a}) = true$ iff the series of register operations $\vec{a}$ is satisfiable, i.e., there exists a configuration of register contents for which all the operations in the series can be executed successfully. Determining whether $sat_i(\vec{a}) = true$ by exhaustively checking all the vectors in Γ^n may be inefficient. Therefore, we show a necessary and sufficient condition for determining whether $sat_i(\vec{a}) = true$ for some $\vec{a} \in \left(Actions_n^\Gamma\right)^+_{|i}$, which can be checked efficiently. In addition, this condition will be useful in optimizing the series of register operations as will be shown later. A series of register operations over the i-th register is *not satisfiable* if either one of the following holds:

- A write operation is followed by a read operation expecting a different value.
- A read operation is immediately followed by a read operation expecting a different value.

Theorem 4
For all $\vec{a} = \langle (op_1, i, \gamma_1), (op_2, i, \gamma_2), \ldots, (op_s, i, \gamma_s) \rangle \in \left(Actions_n^\Gamma\right)^+_{|i}$, $sat_i(\vec{a}) = false$ if and only if either one of the following holds:

1. There exists k, $1 \leq k < s$, such that $op_k = W$ and there exists m, $k < m \leq s$, such that $op_m = R$, $\gamma_k \neq \gamma_m$ and for all j, $k < j < m$, $op_j = R$.
2. There exists k, $1 \leq k < s$, such that $op_k = op_{k+1} = R$ and $\gamma_k \neq \gamma_{k+1}$.

Notice that if $i = 0$, then by the definition of FSRAs, all the register operations in the series are the same operation, which is $(R, 0, \#)$; and this operation can never fail. In addition, if all the operations in the series are write operations, then again, by the definition of FSRAs, these operations can never fail. If none of the two conditions of the theorem holds, then the series of register operations is satisfiable.

We now show how to optimize a series of operations over a given register. An optimized series is defined only over satisfiable series of register operations in the following way:

- If all the operations are write operations, then leave only the last one (since it will overwrite all its predecessors).
- If all the operations are read operations, then by theorem 4, they are all the same operation, and in this case just leave one of them.
- If there are both read and write operations, then distinguish between two cases:
  - If the first operation is a write operation, leave only the last write operation in the series.
  - If the first operation is a read operation, leave the first operation (which is read) and the last write operation in the series. If the last write operation writes into the register the same symbol that the read operation required, then the write is redundant; leave only the read operation.

Definition
Define a function $min_i : \left(Actions_n^\Gamma\right)^+_{|i} \longrightarrow \left(Actions_n^\Gamma\right)^+_{|i}$. Let $\vec{a} = \langle (op_1, i, \gamma_1), \ldots, (op_s, i, \gamma_s)\rangle$. If $sat_i(\vec{a}) = true$ then:

- If for all k, $1 \leq k \leq s$, $op_k = W$, define $min_i(\vec{a}) = \langle (W, i, \gamma_s)\rangle$.
- If for all k, $1 \leq k \leq s$, $op_k = R$ then define $min_i(\vec{a}) = \langle (R, i, \gamma_s)\rangle$.
- If there exists m, $1 \leq m \leq s$ such that $op_m = W$ and if there exists t, $1 \leq t \leq s$, such that $op_t = R$ then:

$$min_i(\vec{a}) = \begin{cases} \langle (W, i, \gamma_j)\rangle & \text{if } op_1 = W \text{ and} \\ & \text{for all } k,\ j < k \leq s,\ op_k = R \\ \langle (R, i, \gamma_1), (W, i, \gamma_j)\rangle & \text{if } op_1 = R \text{ and} \\ & \text{for all } k,\ j < k \leq s,\ op_k = R \text{ and } \gamma_1 \neq \gamma_j \\ \langle (R, i, \gamma_1)\rangle & \text{if } op_1 = R \text{ and if there exists } j,\ 1 \leq j \leq s, \\ & \text{such that for all } k,\ j < k \leq s, \\ & op_k = R \text{ and } \gamma_1 = \gamma_j \end{cases}$$

The formal proof that $min_i(\vec{a})$ is the minimal equivalent series of register operations of $\vec{a}$ is suppressed.

We now show how to optimize a series of register operations. Define a function $min : \left(Actions_n^\Gamma\right)^+ \longrightarrow \left(Actions_n^\Gamma\right)^+ \cup \{null\}$. For all $\vec{a} \in \left(Actions_n^\Gamma\right)^+$ define $min(\vec{a}) = \vec{b}$ where $\vec{b}$ is obtained from $\vec{a}$ by:

- Each subseries $\vec{a}_i$ of $\vec{a}$, consisting of all the register operations on the i-th register, is checked for satisfaction. If $sat_i(\vec{a}_i) = false$ then the arc cannot be traversed and $min(\vec{a}) = \vec{b} = null$. If $sat_i(\vec{a}_i) = true$ then $\vec{a}_i$ is replaced in $\vec{a}$ by $min(\vec{a}_i)$. Notice that the order of the minimized subseries in the complete series is unimportant as they operate on different registers.
- If there exists $i \neq 0$, such that $\vec{a}_i$ is not empty, then the subseries $\vec{a}_0$ consisting only of operations of the form $(R, 0, \#)$ is deleted from $\vec{a}$.

Finally, given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$, construct an equivalent FSRA $A' = \langle Q, q_0, \Sigma, \Gamma, n, \delta', F\rangle = Opt(A)$ where

$$\delta' = \left\{ \left(q_1, \sigma, \langle min(\vec{a})\rangle, q_2\right) \mid \left(q_1, \sigma, \langle \vec{a}\rangle, q_2\right) \in \delta \text{ and } min(\vec{a}) \neq null \right\}$$

$Opt(A)$ is optimized with respect to register operations.

Like FSAs, FSRAs may have states that can never be reached or can never lead to a final state. These states (and their connected arcs) can be removed in the same way they are removed in FSAs. In sum, FSRA optimization is done in two stages:

1. Minimizing the series of register operations over the FSRA transitions.
2. Removing redundant states and arcs.

Notice that stage 1 must be performed before stage 2 as it can result in further reduction in the size of the network when performing the second stage. For a given FSRA A, define $OPT(A)$ as the FSRA obtained from $Opt(A)$ by removing all the redundant

states and transitions. An FSRA A is **optimized** if $OPT(A) = A$ (notice that $OPT(A)$ is unique, i.e., if $B = OPT(A)$ and $C = OPT(A)$, then $B = C$).

3.4 FSRA Minimization

FSRAs can be minimized along three different axes: states, arcs, and registers. Reduction in the number of registers can always be achieved by converting an FSRA to an FSA (Section 3.1), eliminating registers altogether. Since FSRAs are inherently non-deterministic (see the discussion of linearization below), their minimization is related to the problem of non-deterministic finite-state automata (NFA) minimization, which is known to be NP-hard.[6] However, while FSRA arc minimization is NP-hard, FSRA state minimization is different. Recall that in theorem 3 we have shown that any FSA has an equivalent FSRA-2 with 3 states and 2 registers. It thus follows that any FSRA has an equivalent FSRA-2 with 3 states (simply convert the FSRA to an FSA and then convert it to an FSRA-2 with 3 states). Notice that minimizing an FSRA in terms of states or registers can significantly increase the number of arcs. As many implementations of finite-state devices use space that is a function of the number of arcs, the benefit that lies in such minimization is limited. Therefore, a different minimization function, involving all the three axes, is called for. We do not address this problem in this work. As for arc minimization, we cite the following theorem. As its proof is most similar to the corresponding proof on NFA, we suppress it.

Theorem 5
FSRA arc minimization is NP-hard.

The main advantage of finite-state devices is their linear recognition time. In finite-state automata, this is achieved by determinizing the network, ensuring that the transition relation is a function. In FSRAs, in contrast, a functional transition relation does not guarantee linear recognition time, since multiple possible transitions can exist for a given state and a given input symbol. For example, given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$, and some $q, q_1, q_2 \in Q$ and $\sigma \in \Sigma$, two arcs such as $(q, \sigma, \langle(W, 1, a)\rangle, q_1)$, $(q, \sigma, \langle(W, 1, b)\rangle, q_2) \in \delta$ do not hamper the functionality of the FSRA transition relation. However, they do imply that for the state q and for the same input symbol (σ), more than one possible arc can be traversed. We use **deterministic** to denote FSRAs in which the transition relation is a function, and a new term, **linearized,** is used to denote FSRAs for which linear recognition time is guaranteed.

Generally, a FSRA is linearized if it is optimized, ϵ-free, and given a current state and a new input symbol, and at most one transition can be traversed. Thus, if the transition relation includes two arcs of the form $(q, \sigma, \langle\vec{a}\rangle, q_1), (q, \sigma, \langle\vec{b}\rangle, q_2)$, then $\vec{a}$ and $\vec{b}$ must be a contradicting series of register operations. Two series of register operations are contradicting if at most one of them is satisfiable. Since the FSRA is optimized, each series of register operations is a concatenation of subseries, each operating on a different register; and the subseries operating on the i-th register must be either empty or $\langle(W, i, \gamma)\rangle$ or $\langle(R, i, \gamma)\rangle$ or $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$. $\langle(W, i, \gamma)\rangle$ contradicts neither $\langle(R, i, \gamma)\rangle$ nor $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$. $\langle(R, i, \gamma)\rangle$ and $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$ are contradicting only if $\gamma \neq \gamma_1$.

6 While this theorem is a part of folklore, we were unable to find a formal proof. We explicitly prove this theorem in Cohen-Sygal (2004).

Definition
An FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$, is linearized if it is optimized, ϵ-free, and for all $(q, \sigma, \langle \vec{a} \rangle, q_1), (q, \sigma, \langle \vec{b} \rangle, q_2) \in \delta$ such that $\langle \vec{a} \rangle \neq \langle \vec{b} \rangle$, where $\langle \vec{a} \rangle = \langle (op_1^1, i_1^1, \gamma_1^1), \ldots, (op_k^1, i_k^1, \gamma_k^1) \rangle$ and $\langle \vec{b} \rangle = \langle (op_1^2, i_1^2, \gamma_1^2), \ldots, (op_m^2, i_m^2, \gamma_m^2) \rangle$, there exists j_1, $1 \leq j_1 \leq k$ and there exists j_2, $1 \leq j_2 \leq m$, such that $op_{j_1}^1 = op_{j_2}^2 = R$, $i_{j_1}^1 = i_{j_2}^2$ and $\gamma_{j_1}^1 \neq \gamma_{j_2}^2$.

A naïve algorithm for converting a given FSRA into an equivalent linearized one is to convert it to an FSA and then determinize it. In the worst case, this results in an exponential increase in the network size. As the following theorem shows, FSRA linearization is NP-complete.

Theorem 6
FSRA linearization is NP-complete.

Proof 2
Evidently, given an FSRA A, it can be verified in polynomial time that A is linearized. Therefore, FSRA linearization is in NP.

Let ϕ be a CNF formula with m clauses and n variables. Construct an FSRA A such that $L(A) = \{\epsilon\}$ if ϕ is satisfiable, otherwise $L(A) = \emptyset$.
Let $x_1, \ldots, x_n$ be the variables of ϕ. Define $A = \langle Q, q_0, \Sigma, \Gamma, n, 1, \delta, F \rangle$, such that:

- $Q = \{q_0, q_1, \ldots, q_{n+m}\}$
- $F = \{q_{n+m}\}$
- Σ is irrelevant (choose any Σ).
- $\Gamma = \{T, F\}$.
- $$\delta = \{(q_{i-1}, \epsilon, (W, i, T), q_i) \mid 1 \leq i \leq n\} \cup \{(q_{i-1}, \epsilon, (W, i, F), q_i) \mid 1 \leq i \leq n\}$$
$$\cup$$
$$\{(q_{n+i-1}, \epsilon, (R, j, T), q_{n+i}) \mid 1 \leq i \leq m \text{ and } x_j \text{ occurs in the } i\text{-th clause}\}$$
$$\cup$$
$$\{(q_{n+i-1}, \epsilon, (R, j, F), q_{n+i}) \mid 1 \leq i \leq m \text{ and } \overline{x_j} \text{ occurs in the } i\text{-th clause}\}$$

  Notice that each path in A is of length $m + n$. The first n arcs in the path write an assignment into the registers, then it is possible to traverse the remaining m arcs in the path only if the assignment stored into the registers satisfies ϕ.

For example, for the CNF formula $(x_1 \vee \overline{x_2} \vee x_5) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_3 \vee \overline{x_4} \vee x_5)$, the FSRA of Figure 10 is constructed. Observe that the number of states and arcs in this FSRA is $O(mn)$. Now, linearize A into an FSRA A' and assume this can be done in polynomial time. By the definition of linearized FSRA, A' does not contain ϵ-arcs. Therefore, $\epsilon \in L(A')$ iff the initial state of A' is also a final one. Hence, ϕ is satisfiable iff the initial state of A' is also a final one. ■

4. A Regular Expression Language for FSRAs

Regular expressions are a formal way for defining regular languages. Regular language operations construct regular expressions in a convenient way. Several toolboxes (software packages) provide extended regular expression description languages and compil-

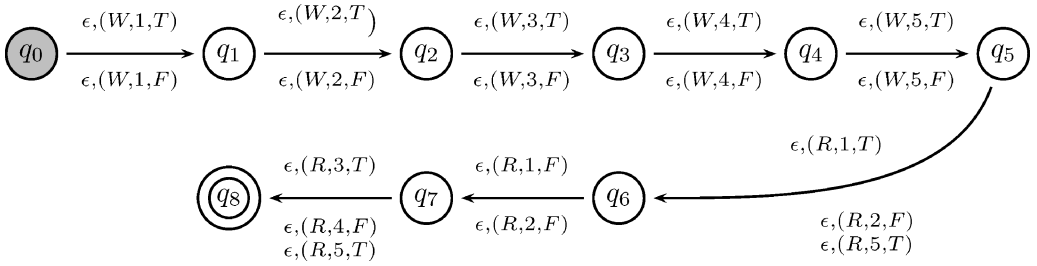

Figure 10
FSRA for a given CNF formula.

ers of the expressions to finite-state devices, automata, and transducers (see Section 1). We provide a regular expression language for constructing FSRAs, the denotations of whose expressions are FSRAs. In the following discussion we assume the regular expression syntax of XFST (Beesley and Karttunen 2003) for basic expressions.[7]

Definition
Let $Actions_n^\Gamma = \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma$, where n is the number of registers and Γ is the register alphabet. If R is a regular expression and $\vec{a} \in \left(Actions_n^\Gamma\right)^+$ is a series of register operations, then the following are also regular expressions: $\vec{a} \triangleright R$, $\vec{a} \triangleright \triangleright R$, $\vec{a} \triangleleft R$, and $\vec{a} \triangleleft \triangleleft R$.

We now define the denotation of each of the above expressions. Let R be a regular expression whose denotation is the FSRA A, and let $\vec{a} \in \left(Actions_n^\Gamma\right)^+$. The denotation of $\vec{a} \triangleleft R$ is an FSRA A' obtained from A by adding a new node, q, which becomes the initial node of A', and an arc from q to the initial node of A; this arc is labeled by ϵ and associated with $\vec{a}$. Notice that in the regular expression $\vec{a} \triangleleft R$, R and $\vec{a}$ can contain operations on joint registers. In some cases, one would like to distinguish between the registers used in $\vec{a}$ and in R. Usually, it is up to the user to correctly manipulate the usage of registers, but in some cases automatic distinction seems desirable. For example, if R includes a circumfix operator (see below), its corresponding FSRA will contain register operations created automatically by the operator. Instead of remembering that circumfixation always uses register 1, one can simply distinguish between the registers of $\vec{a}$ and R via the $\vec{a} \triangleleft \triangleleft R$ operator. This operator has the same general effect as the previous one, but the transition relation in its FSRA uses fresh registers that are added to the machine.

In a similar way, the operators $\vec{a} \triangleright R$ and $\vec{a} \triangleright \triangleright R$ are translated into networks. The difference between these operators and the previous ones is that here, the register operations in $\vec{a}$ are executed *after* traversing all the arcs in the FSRA denoted by R. Using these additional operators, it is easy to show that every FSRA has a corresponding regular expression denoting it, by a trivial modification of the construction presented by Kleene (1956).

Example 8
Consider the case of vowel harmony in Warlpiri (Sproat 1992), where the vowel of suffixes agrees in certain aspects with the vowel of the stem to which it is attached.

7 In particular, concatenation is denoted by juxtaposition and ϵ is denoted by 0.

A simplified account of the phenomenon is that suffixes come in two varieties, one with *'i'* vowels and one with *'u'* vowels. Stems whose last vowel is *'i'* take suffixes of the first variety, whereas stems whose last vowel is *'u'* or *'a'* take the other variety. The following examples are from Sproat (1992) (citing Nash (1980)):

1. *maliki+kiḷi+ḷi+lki+ji+li*
   (dog+PROP+ERG+then+me+they)
2. *kuḍu+kuḷu+ḷu+lku+ju+lu*
   (child+PROP+ERG+then+me+they)
3. *minija+kuḷu+ḷu+lku+ju+lu*
   (cat+PROP+ERG+then+me+they)

An FSRA that accepts the above three words is denoted by the following complex regular expression:

```
define LexI [m a l i k i];    % words ending in 'i'
define LexU [k u d u];        % words ending in 'u'
define LexA [m i n i j a];    % words ending in 'a'
! Join all the lexicons and write to register 1 'u' or 'i'
! according to the stem's last vowel.
define Stem [<(W,1,i)> ◁ LexI] | [<(W,1,u)>  ◁ [LexU | LexA]];
! Traverse the arc only if the scanned symbol is the content of
! register 1.
define V [<(R,1,i)> ▷ i] | [<(R,1,u)> ▷ u];
define PROP [+ k V l V];      % PROP suffix
define ERG  [+ l V];          % ERG suffix
define Then [+ l k V];        % suffix indicating 'then'
define Me   [+ j V];          % suffix indicating 'me'
define They [+ l V];          % suffix indicating 'they'
! define the whole network
define WarlpiriExample Stem PROP ERG Then Me They;
```

Register 1 stores the last vowel of the stem, eliminating the need to duplicate paths for each of the different cases. The lexicon is divided into three separate lexicons (LexI, LexU, LexA), one for each word ending ('i', 'u', or 'a' respectively). The separate lexicons are joined into one (the variable Stem) and when reading the last letter of the base word, its type is written into register 1. Then, when suffixing the lexicon base words, the variable V uses the the content of register 1 to determine which of the symbols 'i', 'u' should be scanned and allows traversing the arc only if the correct symbol is scanned. Note that this solution is applicable independently of the size of the lexicon, and can handle other suffixes in the same way.

Example 9
Consider again Example 7. The FSRA constructed for Arabic nominative definite and indefinite nouns can be denoted by the following regular expression:

```
! Read the definite article (if present).
! Store in register 1 whether the noun is definite or indefinite.
! Store in register 2 the actual form of the definite article.
```

```
define Prefix [<(W,1,indef)> ◁ 0] | [<(W,1,def),(W,2,l)> ◁ 'al] |
       [<(W,1,def),(W,2,$)> ◁ 'a$] | [<(W,1,def),(W,2,d)> ◁ 'ad];
! Normal base - definite and indefinite
define Base [ [<(R,2,l)> ◁ 0] | [<(R,1,indef)> ◁ 0] ]
       [ [k i t a a b] | [q a m a r] ];
! Bases beginning with $ - definite and indefinite
define $Base [ [<(R,2,$)> ◁ 0] | [<(R,1,indef)> ◁ 0] ] [$ a m s];
! Bases beginning with d - definite and indefinite
define dBase [ [<(R,2,d)> ◁ 0] | [<(R,1,indef)> ◁ 0] ] [d a f t a r];
! Read definite and indefinite suffixes.
define Suffix [<(R,1,def)> ▷ u] | [<(R,1,indef)> ▷ un];
! The complete network.
define ArabicExample Prefix [Base | $Base | dBase] Suffix;
```

The variable Prefix denotes the arcs connecting the first two states of the FSRA, in which the definite article (if present) is scanned and information indicating whether the word is definite or not is saved into register 1. In addition, if the word is definite then register 2 stores the actual form of the definite article. The lexicon is divided into several parts: The Base variable denotes nouns that do not trigger assimilation. Other variables ($Base, dBase) denote nouns that trigger assimilation, where for each assimilation case, a different lexicon is constructed. Each part of the lexicon deals with both its definite and indefinite nouns by allowing traversing the arcs only if the register content is appropriate. The variable Suffix denotes the correct suffix, depending on whether the noun is definite or indefinite. This is possible using the information that was stored in register 1 by the variable Prefix.

5. Linguistic Applications

We demonstrated in examples 5 and 6 that FSRAs can model some non-concatenative phenomena more efficiently than standard finite-state devices. We now introduce new regular expression operators, accounting for our motivating linguistic phenomena, and show how expressions using these operators are compiled into the appropriate FSRA.

5.1 Circumfixes

We introduce a dedicated regular expression operator for circumfixation and show how expressions using this operator are compiled into the appropriate FSRA. The operator accepts a regular expression, denoting a set of bases, and a set of circumfixes, each of which is a pair of regular expressions (prefix, suffix). It yields as a result an FSRA obtained by applying each circumfix to each of the bases. The main purpose of this operator is to deal with cases in which the circumfixes are pairs of strings, but it is defined such that the circumfixes can be arbitrary regular expressions.

Definition
Let Σ be a finite set such that $\Box, \{, \}, \langle, \rangle, \otimes \notin \Sigma$. We define the $\otimes$ operation to be of the form

$$R \otimes \{\langle \beta_1 \Box \gamma_1 \rangle \langle \beta_2 \Box \gamma_2 \rangle \ldots \langle \beta_m \Box \gamma_m \rangle\}$$

where: $m \in \mathbb{N}$ is the number of circumfixes; R is a regular expression over Σ denoting the set of bases; and β_i, γ_i for $1 \leq i \leq m$ are regular expressions over Σ denoting the prefix and suffix of the i-th circumfix, respectively.

Notice that R, β_i, γ_i may denote infinite sets. To define the denotation of this operator, let $A_i^{\beta}, A_i^{\gamma}$ be the FSRAs denoted by β_i, γ_i, respectively. The operator yields an FSRA constructed by concatenating three FSRAs. The first is the FSRA constructed from the union of the FSRAs $A'^{\beta}_1, \ldots, A'^{\beta}_m$, where each A'^{β}_i is an FSRA obtained from A_i^{β} by adding a new node, q, which becomes the initial node of A'^{β}_i, and an arc from q to the initial node of A_i^{β}; this arc is labeled by ϵ and associated with $\langle (W, 1, \beta_i \Box \gamma_i) \rangle$ (register 1 is used to store the circumfix). In addition, the register operations of the FSRA A_i^{β} are shifted by one register in order not to cause undesired effects by the use of register 1. The second FSRA is the FSRA denoted by the regular expression R (again, with one register shift) and the third is constructed in the same way as the first one, the only difference being that the FSRAs are those denoted by $\gamma_1, \ldots, \gamma_m$ and the associated register operation is $\langle (R, 1, \beta_i \Box \gamma_i) \rangle$. Notice that the concatenation operation, defined in Section 3.2.2, adjusts the register operations in the FSRAs to be concatenated, to avoid undesired effects caused by using joint registers. We use this operation to concatenate the three FSRAs, leaving register 1 unaffected (to handle the circumfix).

Example 10
Consider the participle-forming combinations in German, e.g., the circumfix *ge-t*. A simplified account of the phenomenon is that German verbs in their present form take an *n* suffix but in participle form they take the circumfix *ge-t*. The following examples are from Sproat (1992):

säuseln	*'rustle'*	*gesäuselt*	*'rustled'*
brüsten	*'brag'*	*gebrüstet*	*'bragged'*

The FSRA of Figure 11, which accepts the four forms, is denoted by the regular expression

$$[s\ \ddot{a}\ u\ s\ e\ l \mid b\ r\ \ddot{u}\ s\ t\ e] \otimes \{\langle \epsilon \Box n \rangle \langle g\ e \Box t \rangle\}$$

This regular expression can be easily extended to accept more German verbs in other forms. More circumfixation phenomena in other languages such as Indonesian and Arabic can be modeled in the same way using this operator.

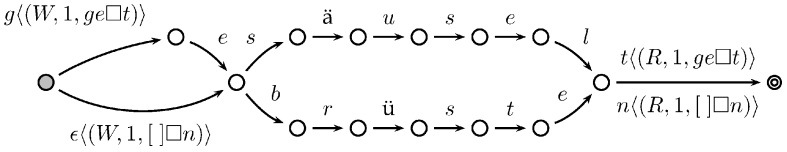

Figure 11
Participle-forming combinations in German.

Example 11
Consider again Example 5. The FSRA accepting all the possible combinations of stems and the Hebrew circumfixes *h-a, ht-ut, m-ϵ* can be denoted by the regular expression $R \otimes \{\langle h\Box a\rangle\langle ht\Box ut\rangle\langle m\Box\epsilon\rangle\}$ where R denotes an FSA accepting the roots.

5.2 Interdigitation

Next, we define a dedicated operator for interdigitation. It accepts a set of regular expressions, representing a set of roots, and a list of patterns, each of which containing exactly n slots. It yields as a result an FSRA denoting the set containing all the strings created by splicing the roots into the slots in the patterns. For example, consider the Hebrew roots *r.$.m, p.&.l, p.q.d* and the Hebrew patterns *hit□a□e□, mi□□a□, ha□□a□a*. The roots are all trilateral, and the patterns have three slots each. Given these two inputs, the new operator yields an FSRA denoting the set {*hitra$em, hitpa&el, hitpaqed, mir$am, mip&al, mipqad, har$ama, hap&ala, hapqada*}.

Definition
Let Σ be a finite set such that $\Box, \{, \}, \langle, \rangle, \oplus \notin \Sigma$. We define the **splice** operation to be of the form

$$\{\langle\alpha_{11}, \alpha_{12}, ..., \alpha_{1n}\rangle, \langle\alpha_{21}, \alpha_{22}, ..., \alpha_{2n}\rangle, ..., \langle\alpha_{m1}, \alpha_{m2}, ..., \alpha_{mn}\rangle\}$$

$$\oplus$$

$$\{\langle\beta_{11}\Box\beta_{12}\Box...\beta_{1n}\Box\beta_{1\,n+1}\rangle, \langle\beta_{21}\Box\beta_{22}\Box...\beta_{2n}\Box\beta_{2\,n+1}\rangle, ..., \langle\beta_{k1}\Box\beta_{k2}\Box...\beta_{kn}\Box\beta_{k\,n+1}\rangle\}$$

where:

- $n \in \mathbb{N}$ is the number of slots (represented by '□') in the patterns into which the roots letters should be inserted.
- $m \in \mathbb{N}$ is the number of roots to be inserted.
- $k \in \mathbb{N}$ is the number of patterns.
- α_{ij}, β_{ij} are regular expressions (including regular expressions denoting FSRAs).

The left set is a set of roots to be inserted into the slots in the right set of patterns. For the sake of brevity, β_i and α_i are used as shorthand notations for $\beta_{i1}\Box\beta_{i2}\Box...\Box\beta_{i(n+1)}$ and $\alpha_{i1}\alpha_{i2}...\alpha_{in}$, respectively.

Consider first the case where $\alpha_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ and $\beta_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq k$ and $1 \leq j \leq n+1$. In this case the splice operation yields as a result an FSRA-1 $A = \langle Q, q_0, \Sigma, \Gamma, 3, \delta, F\rangle$, such that $L(A) = \{\beta_{j1}\alpha_{i1}\beta_{j2}\alpha_{i2}...\beta_{jn}\alpha_{in}\beta_{j(n+1)} \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq k\}$, where:

- $Q = \{q_0, q_1, ..., q_{2n+1}\}$
- $F = \{q_{2n+1}\}$
- $\Sigma = \left(\{\alpha_{ij} \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq n\} \cup \{\beta_{ij} \mid 1 \leq i \leq k\ ,\ 1 \leq j \leq n+1\}\right) \setminus \{\epsilon\}$

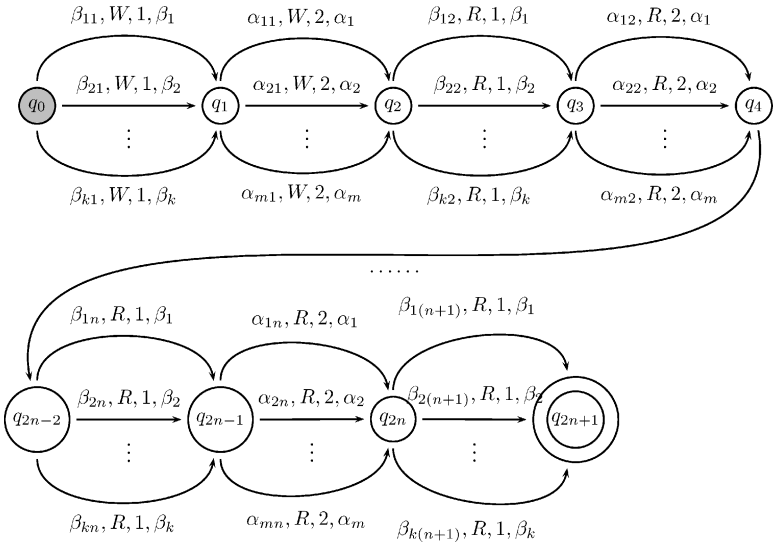

Figure 12
Interdigitation FSRA – general.

- $\Gamma = \{\beta_i |\ 1 \leq i \leq k\} \cup \{\alpha_i |\ 1 \leq i \leq m\} \cup \{\#\}$.
- $\delta = \{(q_0, \beta_{i1}, W, 1, \beta_i, q_1) |\ 1 \leq i \leq k\}$
  $\cup$
  $\{(q_1, \alpha_{i1}, W, 2, \alpha_i, q_2) |\ 1 \leq i \leq m\}$
  $\cup$
  $\{(q_{2j-2}, \beta_{ij}, R, 1, \beta_i, q_{2j-1}) |\ 1 \leq i \leq k\ ,\ 2 \leq j \leq n+1\}$
  $\cup$
  $\{(q_{2j-1}, \alpha_{ij}, R, 2, \alpha_i, q_{2j}) |\ 1 \leq i \leq m\ ,\ 2 \leq j \leq n\}$

This FSRA is shown in Figure 12. It has 3 registers, where register 1 remembers the pattern and register 2 remembers the root. Notice that the FSRA will have 3 registers and $2n + 2$ states for any number of roots and patterns. The number of arcs is $k \times (n + 1) + m \times n$. In the (default) case of trilateral roots, for m roots and k patterns the resulting machine has a constant number of states and $O(k + m)$ arcs.

In the general case, where α_{ij} and β_{ij} can be arbitrary regular expressions, the construction of the FSRA denoted by this operation is done in the same way as in the case of circumfixes with two main adjustments. The first is that in this case the final FSRA is constructed by concatenating $2n + 1$ intermediate FSRAs (n FSRAs for the n parts of the roots and $n + 1$ FSRAs for the $n + 1$ parts of the patterns). The second is that here, 2 registers are used to remember both the root and the pattern. We suppress the detailed description of the construction.

Example 12
Consider again the Hebrew roots *r.$.m, p.&.l, p.q.d* and the Hebrew patterns *hit□a□e□*, *mi□□a□*, and *ha□□a□a*. The splice operation

$$\{\langle r, \$, m\rangle\langle p, \&, l\rangle\langle p, q, d\rangle\} \oplus \{\langle hit\Box a\Box e\Box\rangle\langle mi\Box\Box a\Box\rangle\langle ha\Box\Box a\Box a\rangle\}$$

yields the FSRA of Figure 13. The ϵ-arc was added only for the convenience of the drawing.

It should be noted that like other processes of derivational morphology, Hebrew word formation is highly idiosyncratic: Not all roots combine with all patterns, and there is no systematic way to determine when such combinations will be realized in the language. Yet, this does not render our proposed operators useless: One can naturally characterize classes of roots and classes of patterns for which all the combinations exist. Furthermore, even when such a characterization is difficult to come by, the splice operator can be used, in combination with other extended regular expression operators, to define complex expressions for generating the required language. This is compatible with the general approach for using finite-state techniques, implementing each phenomenon independently and combining them together using closure properties.

5.3 Reduplication

We now return to the reduplication problem as was presented in example 3. We extend the finite-state registered model to efficiently accept $L_n = \{ww \mid w \in \Sigma^*,\ |w| = n\}$, a finite instance of the general problem, which is arguably sufficient for describing reduplication in natural languages. Using FSRAs as defined above does not improve space efficiency, because a separate path for each reduplication is still needed. Notice that the different symbols in L_n have no significance except the pattern they create. Therefore, FSRAs are extended in order to be able to identify a pattern without actually distinguishing between different symbols in it. The extended model, FSRA*, is obtained from the FSRA-1 model by adding a new symbol, '*', assumed not to belong to Σ, and by forcing Γ to be equal to Σ. The '*' indicates equality between the input symbol and the designated register content, eliminating the need to duplicate paths for different symbols.

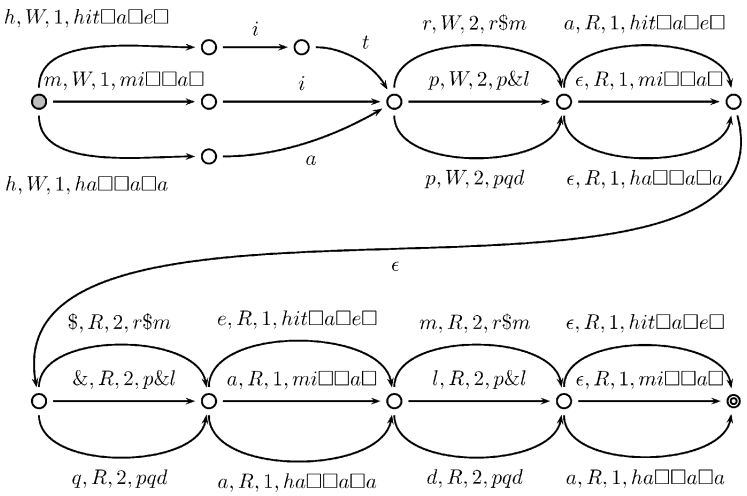

Figure 13
Interdigitation example.

Definition
Let $* \notin \Sigma$. An **FSRA*** is an FSRA-1 where $\Sigma = \Gamma$ (and thus includes '#') and the transition function is extended to be $\delta \subseteq Q \times \Sigma \cup \{\epsilon, *\} \times \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Sigma \cup \{*\} \times Q$. The extended meaning of δ is as follows:

- $(s, \sigma, R, i, \gamma, t) \in \delta$, $(s, \sigma, W, i, \gamma, t) \in \delta$ where $\sigma, \gamma \neq *$ imply the same as before.
- $(s, \sigma, R, i, *, t) \in \delta$ and $(s, *, R, i, \sigma, t) \in \delta$ for $\sigma \neq \epsilon$ imply that if the automaton is in state s, the input symbol is σ and the content of the i-th register is the same σ, then the automaton may enter state t.
- $(s, \sigma, W, i, *, t) \in \delta$ and $(s, *, W, i, \sigma, t) \in \delta$ for $\sigma \neq \epsilon$ imply that if the automaton is in state s and the input symbol is σ, then the content of the i-th register is changed to σ, and the automaton may enter state t.
- $(s, *, R, i, *, t) \in \delta$ implies that if the automaton is in state s, the input symbol is some $\sigma \in \Sigma$ and the content of the i-th register is the same σ, then the automaton may enter state t.
- $(s, *, W, i, *, t) \in \delta$ implies that if the automaton is in state s and the input symbol is some $\sigma \in \Sigma$, then the content of the i-th register is changed to the same σ, and the automaton may enter state t.

With this extended model we can construct an efficient registered automaton for L_n: The number of registers is n+1. Registers 1, ..., n remember the first n symbols to be duplicated. Figure 14 depicts an extended registered automaton that accepts L_n for $n = 4$. Notice that the number of states depends only on n and not on the size of Σ. Figure 15 schematically depicts an extended registered automaton that accepts L_n for some $n \in \mathbb{N}$. The language $\{ww \mid |w| \leq n\}$ for some $n \in \mathbb{N}$ can be generated by a union of FSRA*, each one generating L_n for some $i \leq n$. Since n is usually small in natural language reduplication, the resulting automaton is manageable, and in any case, considerably smaller than the naïve automaton.

5.4 Assimilation

In example 7, FSRAs are used to model assimilation in Arabic nominative definite nouns. Using the FSRA* model defined above, further reduction in the network size can be achieved. The FSRA* of Figure 16 accepts all the nominative definite forms of the Arabic nouns *kitaab*, *qamar*, and *daftar* (more nouns can be added in a similar way). Register 1 stores information about the actual form of the definite article, to ensure that assimilation occurs when needed and only then. Notice that in this FSRA, in contrast to

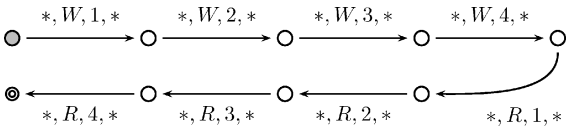

Figure 14
Reduplication for $n = 4$.

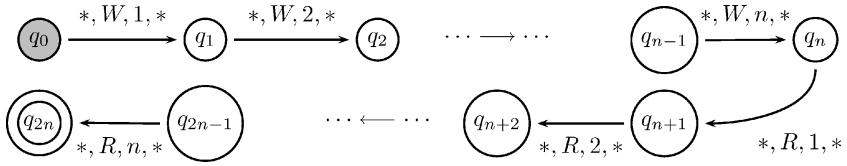

Figure 15
Reduplication – general case.

the FSRA of Figure 8, the definite Arabic article *al* is not scanned as one symbol but as two separate symbols.

6. Finite-state Registered Transducers

We extend the FSRA model to **finite-state registered transducers** (FSRT), denoting relations over two finite alphabets. The extension is done by adding to each transition an output symbol. This facilitates an elegant solution to the problem of binary incrementors which was introduced in Example 4.

Example 13
Consider again the 32-bit incrementor example introduced in Example 4. Recall that a sequential transducer for an n-bit binary incrementor would require 2^n states and a similar number of transitions. Using the FSRT model, a more efficient n-bit transducer can be constructed. A 4-bit FSRT incrementor is shown in Figure 17. The first four transitions copy the input string into the registers, then the input is scanned (using the registers) from right to left (as the carry moves), calculating the result, and the last four transitions output the result (in case the input is 1^n, an extra 1 is added in the beginning). Notice that this transducer guarantees linear recognition time, since from each state only one arc can be traversed in each step, even when there are ϵ-arcs. In the same way, an n-bit transducer can be constructed for all $n \in \mathbb{N}$. Such a transducer will have n registers, $3n + 1$ states and $6n$ arcs. The FSRT model solves the incrementor problem in much the same way it is solved by vectorized finite-state

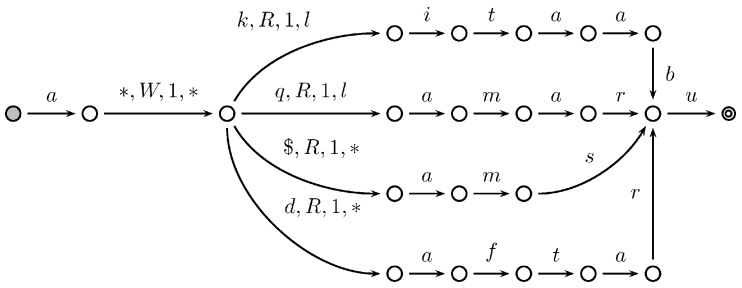

Figure 16
FSRA* for Arabic nominative definite nouns.

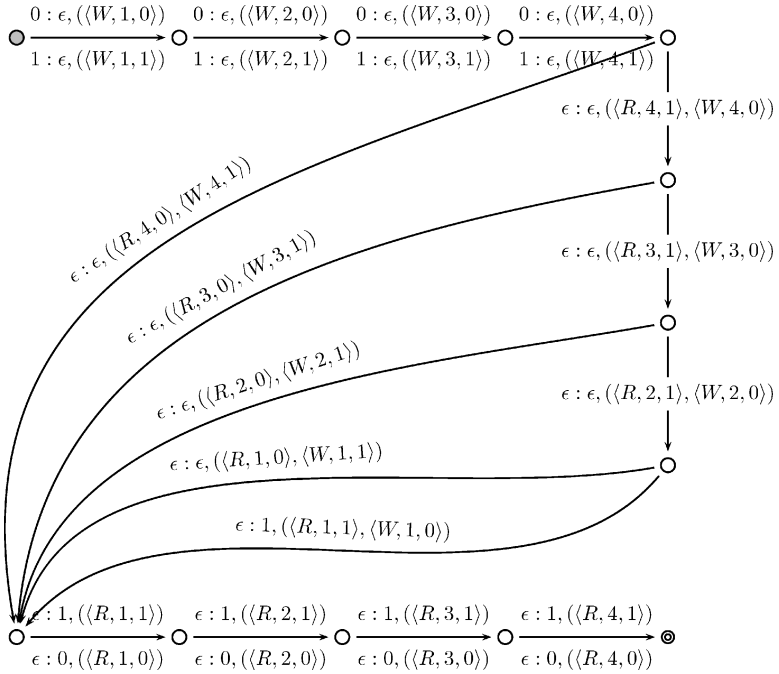

Figure 17
4-bit incrementor using FSRT.

automata, but the FSRT solution is more intuitive and is based on existing finite-state techniques.

It is easy to show that FSRTs, just like FSRAs, are equivalent to their non-registered counterparts. It immediately implies that FSRTs maintain the closure properties of regular relations. As in FSRAs, implementing the closure properties directly on FSRTs is essential for benefiting from their space efficiency. The common operators such as union, concatenation, etc., are implemented in the same ways as in FSRAs. A direct implementation of FSRT composition is a naïve extension of ordinary transducer composition, based on the intersection construction of FSRAs. We explicitly define these operations in Cohen-Sygal (2004).

7. Implementation and Evaluation

In order to practically compare the space and time performance of FSRAs and FSAs, we have implemented the special operators introduced in Sections 4 and 5 for circumfixation and interdigitation, as well as direct construction of FSRAs. We have compared FSRAs with ordinary FSAs by building corresponding networks for circumfixation, interdigitation, and n-bit incrementation. For circumfixation, we constructed networks for the circumfixation of 1,043 Hebrew roots and 4 circumfixes. For interdigitation we constructed a network accepting the splicing of 1,043 roots into 20 patterns. For n-bit incrementation we constructed networks for 10-bit, 50-bit, and 100-bit incrementors. Table 1 displays the size of each of the networks in terms of states, arcs, and actual file size.

Table 1
Space comparison between FSAs and FSRAs.

Operation	Network type	States	Arcs	Registers	File size
Circumfixation	FSA	811	3,824	–	47kB
(4 circumfixes, 1,043 roots)	FSRA	356	360	1	16kB
Interdigitation	FSA	12,527	31,077	–	451kB
(20 patterns, 1,043 roots)	FSRA	58	3,259	2	67kB
10-bit incrementor	Sequential FST	268	322	–	7kB
	FSRT	31	60	10	2kB
50-bit incrementor	Sequential FST	23,328	24,602	–	600kB
	FSRT	151	300	50	8kB
100-bit incrementor	Sequential FST	176,653	181,702	–	4.73Mb
	FSRT	301	600	100	17kB

Table 2
Time comparison between FSAs and FSRAs.

		200 words	1,000 words	5,000 words
Circumfixation	FSA	0.01s	0.02s	0.08s
(4 circumfixes, 1,043 roots)	FSRA	0.01s	0.02s	0.09s
Interdigitation	FSA	0.01s	0.02s	1s
(20 patterns, 1,043 roots)	FSRA	0.35s	1.42s	10.11s
10-bit incrementor	Sequential FST	0.01s	0.05s	0.17s
	FSRT	0.01s	0.06s	0.23s
50-bit incrementor	Sequential FST	0.13s	0.2s	0.59s
	FSRT	0.08s	0.4s	1.6s

Clearly, FSRAs provide a significant reduction in the network size. In particular, we could not construct an n-bit incrementor FSA for any n greater than 100 as a result of memory problems, whereas using FSRAs we had no problem constructing networks even for $n = 50,000$.

In addition, we compared the recognition times of the two models. For that purpose, we used the circumfixation, interdigitation, 10-bit incrementation, and 50-bit incrementation networks to analyze 200, 1,000, and 5,000 words. As can be seen in Table 2, time performance is comparable for the two models, except for interdigitation, where FSAs outperform FSRAs by a constant factor. The reason is that in this network the usage of registers is massive and thereby, there is a higher cost to the reduction of the network size, in terms of analysis time. This is an instance of the common tradeoff of time versus space: FSRAs improve the network size at the cost of slower analysis time in some cases. When using finite-state devices for natural language processing, often the generated networks become too large to be practical. In such cases, using FSRAs can make network size manageable. Using the closure constructions one can build desired networks of reasonable size, and at the end decide whether to convert them to ordinary FSAs, if time performance is an issue.

8. Conclusions

In this work we introduce finite-state registered networks (automata and transducers), an extension of finite-state networks which adds a limited amount of memory, in the

form of *registers*, to each transition. We show how FSRAs can be used to efficiently model several non-concatenative morphological phenomena, including circumfixation, root and pattern word formation in Semitic languages, vowel harmony, and limited reduplication.

The main advantage of finite-state registered networks is their space efficiency. We show that every FSA can be simulated by an equivalent FSRA with three states and two registers. For the motivating linguistic examples, we show a significant decrease in the number of states *and* the number of transitions. For example, to account for all the possible combinations of r roots and p patterns, an ordinary FSA requires $O(r \times p)$ arcs whereas an FSRA requires only $O(r + p)$. As a non-linguistic example, we show a transducer that computes n-bit increments of binary numbers. While an ordinary (sequential) FST requires $O(2^n)$ states and arcs, an FSRT which guarantees linear recognition time requires only $O(n)$ states and arcs.

In spite of their efficiency, finite-state registered networks are equivalent, in terms of their expressive power, to ordinary finite state networks. We provide an algorithm for converting FSRAs to FSAs and prove the equivalence of the models. Furthermore, we provide direct constructions of the main closure properties of FSAs for FSRAs, including concatenation, union, intersection, and composition.

In order for finite-state networks to be useful for linguistic processing, we provide a regular expression language denoting FSRAs. In particular, we provide a set of extended regular expression operators that denote FSRAs and FSRTs. We demonstrate the utility of the operators by accounting for a variety of complex morphological and phonological phenomena, including circumfixation (Hebrew and German), root-and-pattern (Hebrew), vowel harmony (Warlpiri), assimilation (Arabic), and limited reduplication. These dedicated operators can be used in conjunction with standard finite state calculi, thereby providing a complete set of tools for the computational treatment of non-concatenative morphology.

This work opens a variety of directions for future research. An immediate question is the conversion of FSAs to FSRAs. While it is always possible to convert a given FSA to an FSRA (simply add one register which is never used), we believe that it is possible to automatically convert space inefficient FSAs to more compact FSRAs. A pre-requisite is a clear understanding of the parameters for minimization: These include the number of states, arcs, and registers, and the size of the register alphabet. For a given FSRA, the number of states can always be reduced to a constant (theorem 3) and registers can be done away with entirely (by converting the FSRA to an FSA, Section 3.1). In contrast, minimizing the number of arcs in an FSRA is NP-hard (Section 3.4). A useful conversion of FSAs to FSRAs must minimize some combination of these parameters, and while it may be intractable in general, it can be practical in many special cases. In particular, the case of finite languages (acyclic FSAs) is both of practical importance and — we conjecture — can result in good compaction.

More work is also needed in order to establish more properties of FSRTs. In particular, we did not address issues such as sequentiality or sequentiability for this model. Similarly, FSRA* can benefit from further research. All the closure constructions for FSRA*s can be done in a similar way to FSRAs, with the exception of intersection. For intersection, we believe that the use of predicates (van Noord and Gerdemann 2001b) can be beneficial. Furthermore, the use of predicates can be beneficial for describing natural language reduplication where the reduplication is not as bounded as the example we deal with in this work. In addition, the FSRA* model can be extended into transducers.

Finally, in Section 7 we discuss an implementation of FSRAs. Although we have used this system to construct networks for several phenomena, we are interested in

constructing a network for describing the complete morphology of a natural language containing many non-concatenative phenomena, e.g., Hebrew. A morphological analyzer for Hebrew, based on finite-state calculi, already exists (Yona and Wintner 2005), but is very space-inefficient and, therefore, hard to maintain. It would be beneficial to compact such a network using FSRTs, and to inspect the time versus space tradeoff on such a comprehensive network.

Acknowledgments

We are grateful to Dale Gerdemann for his help and inspiration. We thank Victor Harnik and Nissim Francez for their comments on an earlier version of this paper. We are also thankful to the anonymous reviewers, whose comments helped substantially to improve this article. This research was supported by The Israel Science Foundation (grant no. 136/01).

References

Beesley, Kenneth R. 1998. Constraining separated morphotactic dependencies in finite-state grammars. In *Proceedings of FSMNLP-98*, pages 118–127, Bilkent, Turkey.

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

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

Blank, Glenn D. 1985. A new kind of finite-state automaton: Register vector grammar. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 749–755, UCLA.

Blank, Glenn D. 1989. A finite and real-time processor for natural language. *Communications of the ACM*, 32(10):1174–1189.

Cohen-Sygal, Yael. 2004. Computational implementation of non-concatenative morphology. Master's thesis, Department of Computer Science, University of Haifa, Israel.

Holzer, Markus and Martin Kutrib. 2002. State complexity of basic operations on nondeterministic finite automata. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata, 7th International Conference, CIAA 2002*, volume 2608 of *Lecture Notes in Computer Science*, Springer, pages 148–157.

Kaminski, Michael and Nissim Francez. 1994. Finite memory automata. *Theoretical Computer Science*, 134(2):329–364.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Kataja, Laura and Kimmo Koskenniemi. 1988. Finite-state description of Semitic morphology: A case study of ancient Akkadian. In *Proceedings of COLING 88, International Conference on Computational Linguistics*, pages 313–315, Budapest.

Kay, Martin. 1987. Nonconcatenative finite-state morphology. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 2–10, Copenhagen, Denmark.

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

Kleene, S. C. 1956. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, pages 3–42.

Kornai, András. 1996. Vectorized finite-state automata. In *Proceedings of the Workshop on Extended Finite-State Models of Languages in the 12th European Conference on Artificial Intelligence*, pages 36–41, Budapest.

Koskenniemi, Kimmo. 1983. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production*. The Department of General Linguistics, University of Helsinki.

Krauwer, Steven and Louis des Tombe. 1981. Transducers and grammars as theories of language. *Theoretical Linguistics*, 8:173–202.

Lavie, Alon, Alon Itai, Uzzi Ornan, and Mori Rimon. 1988. On the applicability of two-level morphology to the inflection of Hebrew verbs. Technical Report 513,

Department of Computer Science, Technion, 32000 Haifa, Israel.
Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.
Mohri, Mehryar. 2000. Generic epsilon-removal algorithm for weighted automata. In Sheng Yu and Andrei Paun, editors, *5th International Conference, CIAA 2000*, volume 2088, Springer-Verlag, pages 230–242.
Mohri, Mehryar, Fernando Pereira, and Michael Riley. 2000. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32.
Nash, David. 1980. *Topics in Warlpiri Grammar*. Ph.D. thesis, Massachusetts Institute of Technology.
Sproat, Richard W. 1992. *Morphology and Computation*. MIT Press, Cambridge, MA.
van Noord, Gertjan and Dale Gerdemann. 2001a. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen, editors, *Automata Implementation, 4th International Workshop on Implementing Automata, WIA'99, Potsdam, Germany, Revised Papers*, number 2214 in *Lecture Notes in Computer Science.* Springer.
van Noord, Gertjan and Dale Gerdemann. 2001b. Finite state transducers with predicates and identity. *Grammars*, 4(3):263–286.
Walther, Markus. 2000a. Finite-state reduplication in one-level prosodic morphology. In *Proceedings of the First Conference of the North American Chapter of the Association for Computational Linguistics*, pages 296–302, Seattle.
Walther, Markus. 2000b. Temiar reduplication in one-level prosodic morphology. In *Proceedings of SIGPHON, Workshop on Finite-State Phonology*, pages 13–21, Luxembourg.
Yona, Shlomo and Shuly Wintner. 2005. A finite-state morphological grammar of Hebrew. In *Proceedings of the ACL-2005 Workshop on Computational Approaches to Semitic Languages*, Ann Arbor.