Abstract: [Arabic, Morphology, Finite State]

This paper describes a finite-state morphological analyzer of written Modern Standard Arabic words that is available for testing on the Internet at http://www.xrce.xerox.com/research/mltt/arabic. The system consists of the analyzer proper, running on a network server, and Java applets that run on the user's machine and render words in standard Arabic orthography both for input and output. An overview of the system is provided, including the history, finite-state technology, dictionary coverage and status.

1 Introduction

In 1996, the Xerox Research Centre Europe produced a large morphological analyzer for Modern Standard Arabic, henceforth Arabic (Beesley, 1996). In 1997, the rules were rewritten to more reliably support generation, and a Java user interface was added to allow users to interact with the system via the Internet in standard Arabic orthography. The analyzer-generator is based on dictionaries from an earlier project at ALPNET (Beesley, 1989; Beesley et al., 1989; Beesley, 1990; Buckwalter, 1990), but the entire system was extensively redesigned and rebuilt using Xerox finite-state technology.

The system accepts on-line orthographical words of Arabic and returns morphological analyses that identify affixes and separate roots from patterns. Input words may include full diacritics, partial diacritics, or no diacritics; and if diacritics are present, they reduce the amount of ambiguity accordingly. A fully-voweled spelling and a terse but useful English gloss are also returned as part of each analysis.

The system has wide dictionary coverage and is intended as a pedagogical dictionary-lookup aid, a comprehension-assistance tool, and as a component in larger natural-language-processing systems.

2 Challenges of the Arabic System

There are two principal challenges in building any serious morphological analyzer for written Arabic. First, the system must in fact accept as input words of Arabic, returning correct and useful morphological analyses. Second, in order to be psychologically acceptable to many users, the input and output must be rendered in traditional Arabic orthography rather than in romanization.

2.1 Arabic Morphological Analysis

In its simplest diagrammatic form, a morphological analyzer can be characterized as a black-box module (see Figure 1) that accepts words of Arabic and outputs morphological analyses. Often serving as components in larger natural-language-processing systems, morphological analyzers and their internal workings are often quite invisible to the average user. In computer analyses of Arabic, or of any other language, the input words are of
As for the content of the morphological analysis itself, it will always be somewhat theory-dependent, especially in the case of Arabic. In the broadest terms, a morphological analyzer should separate and identify the component morphemes of the input word, labeling them somehow with sufficient information to be useful for the purposes at hand. In the case of Arabic, one would certainly expect a morphological analyzer to separate and identify prefixed word-like morphemes such as the conjunctions \textit{wa–} and \textit{fa–}, prefixed prepositions such as \textit{bi–} and \textit{li–}, the definite article, verbal prefixes and suffixes, nominal case suffixes, and enclitic direct-object and possessive-pronoun suffixes. Where things become more complicated is in the formal analysis of the typical Arabic or, more generally, Semitic STEM; various lexicographers and linguists propose

1. Stems as one-part structures: i.e. a stem is simply treated as a single morpheme.

2. Stems as two-part constructs of a \textbf{ROOT} and a stem-framework \textbf{PATTERN}: e.g. \textit{ktb} (الّم ت ب) and \textit{drс} (دْرُس) where the underscores represent slots for the root consonants, sometimes termed \textit{RADICALS}. In such a system, the root and pattern morphemes are said informally to “interdigitate” together to form stems like \textit{kutib} (كْتِب). For such an analysis of Hebrew stems, see (Harris, 1941).

3. Stems as three-part constructs of a \textbf{ROOT}, a consonant-vowel (CV) \textbf{TEMPLATE}, and a \textbf{VOCALIZATION}: e.g. \textit{ktb} and template CVCVC and vocalization \textit{ui} to form \textit{kutib} (كْتِب), \textit{drs} and CVCCVC and vocalization \textit{a} to form \textit{darras} (دْرَاس), \textit{ktb} and CVVCVC and \textit{a} to form \textit{kaatab} (كَتَأِب), etc. The templates are drawn from a set of perhaps dozens, and the radicals and vowels are associated with the C and V slots via controversial association rules. Such a three-way division was popularized by some of the early work of John J. McCarthy (McCarthy, 1981).

4. Stems as multi-part constructs of a root, a prosodically motivated pattern (from a severely limited set), a vocalization, and various affixes. This approach characterizes some of the more recent work of McCarthy (McCarthy, 1993).

In what follows, I shall assume that stems are built from at least two morpheme components, including a root usually consisting of three (but sometimes two or four) radicals like \textit{ktb} (الّم ت ب), \textit{drс} (دْرُس), \textit{fил} (فْيْل), \textit{klkl} (كَلْكَل) or \textit{tm} (تْم). I shall also assume that a crucial task for a morphological analyzer is to identify these underlying roots, even if they are obscured in the surface string by phonological or orthographical processes. Beyond the arguments for the linguistic reality of Semitic roots, an entirely practical motivation for identifying roots is that words in traditional Arabic dictionaries, such as the authoritative Hans Wehr dictionary (Wehr, 1979), are organized under root headings. To look up a word in such a dictionary, the student must know the root.

Beyond the question of how the stem should be split up theoretically into morphemes, a scientific (testable and disprovable) theory must translate into a workable mechanism for generation and analysis. For a general survey of computational approaches to Arabic, see (Kiraz, 1994b). At ALPNET, we used a slight enhancement of the popular \textbf{TWO-LEVEL MORPHOLOGY} (Koskenniemi, 1983; Gajek et al., 1983; Beesley, 1990; Antworth, 1990), a theoretical and computational framework that has been used to build significant morphological analyzers for English, Greek, Russian, German, Polish, Hungarian, Finnish, Swedish, Danish, Norwegian, Swahili, Klingon and no doubt many other languages. The current system was built using Xerox finite-state technology, a set
of basic algorithms and compilers for building and manipulating finite-state transducers. In both the ALPNET and Xerox implementations, roots and patterns are formalized as regular languages that are straightforwardly combined into stems via intersection, a finite-state operation.

2.2 Arabic Script Input and Output

The ALPNET system was intended to be a component, not a stand-alone system with a user interface, and users were forced to interact with it using a Roman transliteration. This represented a psychological barrier to many reviewers, especially to native Arabic speakers. While trained linguists are used to making a distinction between a language and the orthography (or orthographies) used to write it (Sampson, 1985; Coulmas, 1989; Daniels and Bright, 1996), some observers of the old ALPNET system saw Roman characters on the screen and concluded immediately that we weren’t dealing with real Arabic at all.

It is important to make a distinction between what I will call TRANSLITERATIONS and TRANSCRIPTIONS. I use the term transliteration to denote an orthography that reproduces all and only the significant orthographical distinctions found in a conventional orthography (Arabic orthography in our case) but which carefully substitutes the original symbols with new symbols that are more convenient to store or display. The limitations of typewriters, printers and computer software in Europe and the Americas often make it convenient to represent Arabic orthography using Roman letters, as in the ALPNET system. To qualify as a transliteration, under this definition, the new orthography must be trivially and unambiguously convertible back and forth with the original orthography. Computer text encodings like UNICODE are also transliterations where the substituted symbols are numbers rather than letters. All text on a computer, whether English, Arabic or whatever, is transliterated. The trivial reversibility of traditional Arabic orthography and a proper transliteration mean that they are, for all computational purposes, equivalent.

Transliterations, in the sense just defined, must be distinguished from transcriptions, which usually serve for phonological description and may have little relation to the conventional orthography. Transcriptions have valid uses, and they represent possible orthographies for Arabic, but they are not automatically mappable into the conventional orthography. A computer system based on a transcription would be of no use in analyzing genuine Arabic text written in conventional Arabic orthography.

Despite the fact that the ALPNET system used a proper transliteration rather than a transcription of Arabic, the interface was still not esthetically acceptable to many users. The solution to this problem in the Xerox system was to write a Java interface that renders the underlying codes as genuine Arabic orthography. As HTML and Java still have no built-in support for the display of Arabic, a bitmap font, derived from a public-domain font by Yannis Haralambous, was smuggled into the applet disguised as integer arrays, and the rendering algorithm was implemented at low levels using Java graphics functions.

3 System Description

The information flow in the Xerox Arabic system is shown in Figure 2. Users access the system via the Arabic homepage at http://www.xrce.xerox.com/research/mtlt/arabic. The Arabic HTML entry page is mostly filled up with a Java applet that represents a virtual keyboard. Users can type in words either by mouse-clicking on the virtual-key objects or by typing the corresponding keys on their physical keyboards. Four different keyboard layouts are currently provided: PC-like, Mac-like, and the implementors’ preferred Buckwalter Transliteration laid out on both English and French keyboards. As words are typed, the appropriate UNICODE Arabic characters are added to an internal buffer (Figure 3). That buffer is “observed” by an Arabic Canvas object that renders the actual Arabic orthography on the screen, updating the display each time the buffer is changed in any way.

When the user presses the Enter key (or clicks on the Enter-key object) the buffer contents are sent to a Perl CGI script running on a server in Grenoble. The script applies each input word in an upward direction to the morphological analyzer (Figure 2), which is implemented as a finite-state transducer (FST). Typically there are several output strings, each representing a possible analysis of the input word.

When presenting the various solutions back to the user, it is valuable to display the fully-voweled (full diacritics) spelling of each solution. So each solution string is then applied in a downward direction to a morphological generator, which is exactly the same as the analyzer except for having a lower-level language that is restricted to fully-voweled strings. The various solutions are also tokenized into morphemes, which are looked up in a dictionary of English glosses. The Perl script incorporates the analyses, the generated fully-voweled strings,
and the English glosses into an HTML page which is sent back to the user’s Internet browser. Arabic words in the HTML document are enclosed in APPLET elements which cause the browser to invoke the Java applet that renders the string in traditional Arabic orthography on the user’s screen.

4 Finite-State Morphological Processing

4.1 General Finite-State Theory and Techniques

The Arabic Morphological Analyzer is built using finite-state compilers and algorithms, and the results are stored and run as finite-state transducers. As both Two-Level Morphology (Koskenniemi, 1983; Karttunen, 1983; Gajek et al., 1983; Antworth, 1990) and Finite-State Morphology (Karttunen et al., 1992; Karttunen, 1994) are abundantly documented elsewhere, only an outline of finite-state computing will be provided here. Both systems are based on the insight, originally due to C. Douglas Johnson (Johnson, 1972) and rediscovered by Ronald Kaplan and Martin Kay (Kaplan and Kay, 1981; Kaplan and Kay, 1994), that the rewrite rules used traditionally by linguists to describe phonological derivations are only finite-state in power and can be modeled as finite-state transducers.

Let’s step back a minute to see what this means. All computer scientists and formal linguists are acquainted with REGULAR EXPRESSIONS and the REGULAR LANGUAGES that they denote. In formal language theory, a LANGUAGE is simply a set of strings of symbols (which linguists usually think of as words consisting of letters). If we consider English, French and Arabic similarly as sets of strings, then linguists can, and do, write regular expressions, i.e. finite-state grammars, that model the natural language as closely as possible. From a computational point of view, regular expressions can be compiled into FINITE-STATE MACHINES (FSMs) that “accept” all the words of the language and “reject” all words that are not in the language. The mathematical properties of regular languages are well understood; they are closed under operations such as union, concatenation, and intersection.

A regular expression modeling a natural language like Spanish, compiled into a finite-state machine, could be the basis for a simple linguistic product like a spell checker in a word processor. Every word in the document
Figure 3: Arabic Script Display in Java
would be applied to the finite-state machine, and every word rejected by the machine would be flagged as misspelled. Such a spell checker would be useful and reliable to the extent that the formal language corresponds to the real Spanish language.

**Regular Relations**, which are not nearly so well known (Kaplan and Kay, 1994), are relations or “mappings” between two regular languages. For our human convenience, we can think of a finite-state relation as having an upper-side language and a lower-side language; and each string in one language is related to one or more strings in the other language. From another point of view, we can also think of relations as languages that consist of ordered pairs of strings, and any ordered pair of strings is either in the relation or it’s not. A **Finite-State Transducer** (FST) is the corresponding machine that accepts all and only the ordered pairs in the relation; if given a string from the lower language, an FST returns all the related strings in the upper language, and vice versa. Finite-state transducers can implement more interesting linguistic systems such as morphological analyzers, tokenizers, part-of-speech taggers, and even some simple syntactic parsers.

Consider the cascade of two rewrite rules in Figure 4.1 that maps an underlying string $kaNpat$ consisting of the morpheme $kaN$, with an underspecified nasal $N$, concatenated with another morpheme $pat$, into the surface string $kammat$. Johnson’s result tells us that the rules can be modeled as regular relations and can therefore be compiled into finite-state transducers.

The well-understood mathematics of finite-state transducers tells us that if there is a regular relation between $L_1$ and $L_2$, and another regular relation between $L_2$ and $L_3$, then there exists another regular relation that maps directly from $L_1$ to $L_3$, effectively discarding the intermediate level. If $R_1$ and $R_2$ can indeed be compiled into finite-state transducers $FST_1$ and $FST_2$, then the single transducer can be computed as the composition $FST_1 . o . FST_2$ of the two transducers. By successive application of composition, which is yet another well-understood finite-state operation, even a complex grammar with 60 levels could be reduced to two levels, with corresponding benefits in runtime speed. In addition, transducers are inherently bidirectional, allowing the grammar to be run either “forwards” for generation or “backwards” for analysis.

This was all very nice in theory, but efficient algorithms for compiling and manipulating FSTs did not exist in the early 1980s. Xerox started work to produce these algorithms and rule compilers based on them (Karttunen et al., 1987; Karttunen and Beesley, 1992). Soon it was realized that morphological information (such as part-of-speech tags, person, number, tense and mood tags) could be encoded as symbols within strings, and that thus lexicons themselves could be compiled into transducers (Karttunen, 1993). This lexical FST could then be composed together with the derivation rules to create a single all-encompassing FST called a **Lexical Transducer** (Karttunen et al., 1992; Karttunen, 1994) that contained all the lexical and derivational information, mapping directly between upper-level analysis strings and lower-level surface strings. Such a lexical transducer (Figure 5) then implements the “Black Box” morphological analyzer sketched above in Figure 1.

In the Xerox Arabic system, lexicons are written in the lexc language and are compiled into finite-state transducers. Rules to intersect roots and patterns, and derivational rules to perform deletion, epenthesis, assimilation and metathesis, are written in the twolc language and in another notation known as REPLACE RULES (Karttunen, 1995; Karttunen and Kempe, 1995; Kempe and Karttunen, 1996). Not surprisingly, the rules controlling the realization of hamza (the glottal stop) are particularly subtle. The rules also compile into finite-state transducers, and all the components of the grammar are combined into a single lexical transducer using the finite-state union and composition algorithms.

By keeping within the finite-state domain, grammatical components can be combined and modified using any valid finite-state operation. Lexical transducers can be run forwards to generate or backwards to analyze, and they are computationally very efficient for natural-language problems. The code that runs lexical transducers is completely language-independent.
4.2 Finite-State Grammars and the Arabic Stem

In Two-Level and Finite-State Grammars, dictionary-like formalisms (Antworth, 1990) and even high-level languages such as lexc (Karttunen, 1993) are used to define lexicons. The finite-state operations that they traditionally assume are unioning and concatenation. Linguists specify sublexicons of morphemes, such as Spanish verb stems or Arabic perfect verb endings, and these morpheme strings are formally unioned together. A notation of CONTINUATION CLASSES designates for each morpheme which other classes of morphemes can follow, and this translates naturally into concatenation. The effective limitation of classic finite-state lexicography to modeling concatenative morphology has been noted and criticized, and it has often been assumed that these systems would be useless for modeling non-concatenative morphology as in Arabic.

In the ALPNET Arabic system (Beesley, 1989; Beesley, 1990) and at Xerox, the Arabic lexicons also use the continuation-class notation to indicate the co-occurrence of morphemes in words, and most continuations are implemented in the traditional way as concatenation. However, the association of morphemes comprising the stem is formalized as intersection. At ALPNET, lacking actual finite-state algorithms such as intersection, the intersection of roots and patterns was simulated in C code at runtime, a process dubbed DETOURING. At Xerox, roots and patterns are pre-intersected at compile time using finite-state rules.

A simplified example can be included here. Let us assume that a stem like daras (دَارَس) is analyzed as a combination of a root drs, a CV pattern CVCVC and a vocalization a. We formalize roots as regular languages, e.g. 5

\[
\text{define drs} \ [ \text{d r s} ]/\? ; \\
\text{define ktb} \ [ \text{k t b} ]/\? ; \\
\]

where ? represents any symbol and where / is the ignore operator. The root drs is therefore defined as the regular language consisting of all strings containing d, r and s, in that order, and ignoring any characters around them. Similarly, we define C as the union of all the consonants in our fragment, V as the union of all vowels, and the various CV templates as regular languages consisting of concatenations of consonants and vowels.

\[
\text{define C} \ [ \text{d r s k t q b l} ] ; \quad \text{etcetera} \\
\text{define V} \ [ \text{a i u} ] ; \\
\text{define FormI} \quad \text{C V C V C} ; \\
\text{define FormII} \quad \text{C V C X V C} ; \\
\text{define FormIII} \quad \text{C V V C V C} ; \quad \text{etcetera} \\
\]

Vocalizations are defined similarly:

\[
\text{define PerfAct} \ [ \text{a*}]/\text{V} ; \\
\text{define PerfPass} \ [ \text{u* i}]/\text{V} ; \\
\]
where \[ a^* \] denotes the regular language consisting of any number of \( a \), ignoring all non-vowel symbols. With such definitions, abstract stems can then be computed straightforwardly by intersection, e.g. \( \text{drs} \cap \text{FormI} \cap \text{PerfAct} \) yields the regular language consisting of just one string, \( \text{daras} \); \( \text{drs} \cap \text{FormIII} \cap \text{PerfPass} \) yields stem \( \text{duuris} \); \( \text{drs} \cap \text{FormII} \cap \text{PerfPass} \) yields stem \( \text{durXis} \), where symbol \( X \) is realized later as a copy of the previous consonant (or an orthographical shadda) by subsequently applied variation rules, similar to Antworth’s finite-state treatment of reduplication in Tagalog (Antworth, 1990). The details of the formation of Arabic stems via intersection must be presented elsewhere (Beesley, 1997), and a new finite-state algorithm called \( \text{MERGE} \) is in development and promises results equivalent to intersection but with greater convenience and efficiency.

5 Conclusion

The Xerox Arabic Morphological Analysis and Generator was made available for testing on the Internet 9 September 1997. Currently the underlying lexicons include about 4930 roots, each one hand-encoded to indicate the subset of patterns with which it can combine; in practice, the average root participates in about 18 morphotactically distinct stems, yielding approximately 90,000 stems based on roots and patterns. Some of these are phonologically indistinguishable on the lower side, resulting in approximately 70,000 root-pattern intersections to be performed during compilation. The pattern dictionary contains about 400 phonologically distinct entries, many of them ambiguous. Other sublexicons include prefixes, suffixes, and whole stems that are not root-based. Altogether, the lexicon compiles into a finite-state transducer that accounts for 72,000,000 abstract words, each of which can be analyzed in any of its possible spellings, ranging from no diacritics to fully voweled. Additions to the lexicon are easy for those who are trained in the internal codes.

The Xerox system is still in the research stage, but the project will be continued and commercialized if sufficient interest is shown. It is already known that the dictionary needs more proper names, and testing will no doubt uncover other omissions. We need to devise a way to handle multi-word expressions before the work expands into part-of-speech disambiguation and parsing.

Work will continue on the user interface, including the provision of new virtual-keyboard layouts and cut-and-paste input from online text. The analyzer itself may also be redesigned to take advantage of the newest finite-state algorithms, such as 'merge', which provide a more efficient way to perform the intersections that build stems.

Notes

1 Kenneth R. Beesley, D.Phil. 1983, Edinburgh University, is a Principal Scientist at the Xerox Research Centre Europe, Grenoble Laboratory, Multi-Lingual Theory and Technology Group.

2 Arabic-script displays in this paper were encoded using the excellent ArabTeX package for \( \TeX \) and \( \LaTeX \) by Prof. Dr. Klaus Lagally of the University of Stuttgart. My thanks to him for his help, and especially for a custom-written style file that allow me to use my own favorite Arabic transliteration.

3 The highly influential early Arabic work of (McCarthy, 1981) inspired a wide range of challenges, responses, and attempts at formalization. The formal assumptions of McCarthy were generally attacked by (Hudson, 1986), defended by (Haile and Mtenje, 1988) and then attacked again by (Hudson, 1991). Martin Kay (Kay, 1982; Kay, 1987) accepted McCarthy’s data, but he formalized the association of roots, \( \text{CV} \) templates and vocalizations using a multi-level finite-state transducer. Other work inspired primarily by McCarthy’s data includes (Bird and Blackburn, 1991) and (Kiraz, 1994a).

4 McCarthy (McCarthy, 1981; McCarthy, 1982) cites the charming example of a Hijazi Bedouin play language wherein speakers freely scramble the order of the radicals within a root.

5 A large English system and many other two-level demonstrations are available free from the Summer Institute of Linguistics: http://www.sil.org/pckimmo/pc-kimmo.html. German, Russian, Swahili and the Scandinavian systems were created by Kimmo Koskenniemi, his students and associates: http://www.lingsoft.fi/. Another large German system was written by Anne Schiller (Schiller, 1996). MorphoLogic has used Two-Level Morphology to write large analyzers for Polish and Hungarian:
A two-level Klingon analyzer was built in three weeks of madness by the present author (Beesley, 1992b; Beesley, 1992a; Okrand, 1985).

The define statements in the text are executable commands in the Xerox xfst interface to the finite-state algorithms.

References


