Tracking Morphophonemic Transformation in Arabic Word Generation and Root Extraction

Sane Yagi[1] and Jim Yaghi[2]
[1]Department of Linguistics & Phonetics, University of Jordan, Amman, Jordan
[2]Computer Science Department, MacQuire University, Sydney, Australia

Abstract: *Performing root-based searching, concordancing, and grammar checking in Arabic requires an efficient method for matching stems with roots and vice versa. Such mapping is complicated by the hundreds of manifestations of the same root; the radicals often undergo replacement, fusion, inversion, and/or deletion. It is a challenge, therefore, to keep track of original radicals. An algorithm based on methods used by native speakers is proposed here to track root radicals in the generation process and the subsequent reversal process of root extraction. Verb roots are classified by the types of their radicals and the stems they generate. Roots are molded with morphosemantic and morphosyntactic patterns to generate stems modified for tense, voice, and mode, affixed for different subject number, gender, and person. The surface forms of applicable morphophonemic transformation are then derived using finite state machines. This paper defines what is meant by `stem', describes a stem generation engine that the authors developed, and outlines how a generated stem database is compiled for all Arabic verbs.*

Keywords: *Arabic, morphology, generation, extraction, root, finite state.*

Received October 9, 2005; accepted April 20, 2006

1. Introduction

Morphological parsers and analyzers for Arabic are required to dissect an input word and analyze its components in order to perform the even the simplest of language processing tasks. The letters of the majority of Arabic words undergo transformations rendering their roots unrecognizable. Without the root, it is difficult to identify a word's morphosemantic template, which is necessary for pinpointing its meaning, or its morphosyntactic pattern, which is essential for realizing properties of the verb, such as its tense, voice, mode, subject's number, gender, and person. It is fundamental that an analyser be able to reverse the transformations a word undergoes in order to match the separated root and template with the untransformed ones in its database. Unfortunately, defining rules to reverse transformations is not simple.

Research in Arabic morphology has primarily focused on morphological analysis rather than stem generation. Sliding window algorithms [5] use an approximate string matching approach of input words against lists of roots, morphological patterns, prefixes, and suffixes. Algebraic algorithms [4], on the other hand, assign binary values to morphological patterns and input words, then perform some simple algebraic operations to decompose a word into a stem and affixes. Permutation algorithms [2] use the input word's letters to generate all possible trilateral or quadrilateral sequences without violation of the original order of the letters which is then compared with items in a dictionary of roots until a match is found. Linguistic algorithms [9, 11] remove letters from an input word that belong to prefixes and suffixes and place the remainder of the word into a list. The members of this list are then tested for a match against a dictionary of morphological patterns.

The primary drawback of many of these techniques is that they attempt to analyze using the information found in the letters of the input word. When roots form words, root letters are often transformed by replacement, fusion, inversion, or deletion, and their positions are lost between stem and affix letters. Most attempts use various closest match algorithms, which introduce a high level of uncertainty. In this paper, we define Arabic verb stems such that root radicals, morphological patterns, and transformations are formally specified. When stems are defined this way, input words can be mapped to correct stem definitions, ensuring that transformations match root radicals rather than estimate them.

Morphological transformation in our definition is largely built around finite state morphology [3] which assumes that these transformations can be represented in terms of regular relations between regular language forms. Beesley [3] uses finite state transducers to encode the intersection between roots, morphological patterns, and the transformation rules that account for morphophonemic phenomena such as assimilation, deletion, epenthesis, metathesis, etc.

In this paper, a description of the database required for stem generation is presented, followed by a

definition of stem generation. Then the database together with the definition are used to implement a stem generation engine. This is followed by a suggestion for optimizing stem generation. Finally, a database of generated stems is compiled in a format useful to various applications that the conclusion alludes to.

In the course of this paper, roots are represented in terms of their ordered sequence of three or four radicals in a set notation, i. e., {F, M, L, Q}. When the capitalized Roman characters F, M, L, and Q are used, they represent a radical variable or place holder. They stand for First radical (F), Medial radical (M), Last radical in a trilateral root (L), and last radical in a Quadrilateral root (Q).

For readability, all Arabic script used here is followed by an orthographic transliteration between parentheses, using the Buckwalter standard[1]. Buckwalter's orthographic transliteration provides a one-to-one character mapping from Arabic to US-ASCII characters. With the exception of a few characters, this transliteration scheme attempts to match Arabic sounds to sounds of the Roman letters to the Arabic ones. The following list of Arabic-Roman pairs is a subset of the less obvious transliterations used here: ة (@); ى (Y); َ(a); ِ(i); ُ(u); ْ(o); and ّ(~).

2. Stem Generation Database

Arabic stems can be generated if lists of all roots and all morphological patterns are provided. It is necessary that this data be coupled with a database that links the roots with their morphological patterns (or templates) so that only valid stems are generated for each root. The roots in this database may be molded with morphosemantic and morphosyntactic patterns to generate intermediate form stems. The stems may then be transformed into final surface forms with a number of specific morphophonemic rules using a finite state transducer compiling language.

Figure 1 shows a summary of the stem generation tables and their relations. The *rootslist* table contains all verb roots from the popular Arabic dictionary, Al-Waseet [1], with F, M, L, and Q representing the table fields for up to four radicals per root. A root identifier is used to link this table to the *template* table. The *template* table lists all morphosemantic and morphosyntactic patterns used to generate stems from roots of a certain type. This table also specifies the syntactic properties of stems (voice and tense) generated by using the template entry. The *maindictionary* table links the *rootslist* and *template* tables together and specifies which entries apply to which roots.

[1]The complete table of orthographic transliteration may be found at http://www.qamus.org/transliteration.htm.

Stems generated with these tables are unaffixed stems. The *affix_id* field links each entry to a subject pronominal affix table that uses transformation rules to generate affixed stems. Although object pronominal affixes are not dealt with in this paper, they are generally agglutinating in nature and; therefore; cause no morphophonemic alterations to a stem. They can be added for generation or removed for analysis without affecting the stem at all.

Affixation and transformation rules are both specified using PERL regular expressions [6]. Regular expressions (Regexp) are an algebraic language that is used for building Finite State Transducers (FSTs) that accept regular language. In the next section, Regexp is used to perform morphophonemic transformations and to generate affixed forms of stems. If generated stems are to be useful for root extraction and morphological analysis, it is essential at every stage of generation to be able to track exactly which letters are members of the root radical set, which belong to the template, and what transformations occur on the untransformed stem to produce the final surface form.

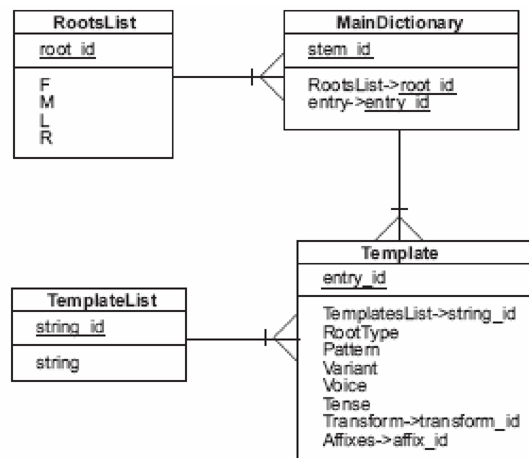

Figure 1. The stem generation database tables and their relations.

3. Definition of Stem Generation

In order to be useful in analysis applications, Arabic stems need to be in a surface form which will only undergo agglutinating changes for any further morphological modification. Stems should be defined in terms of the root radicals, morphosemantic and morphosyntactic template letters, and morphophonemic alterations. By doing so, inverting stem transformations becomes trivial. We require the automatic stem generator to always be aware of the origin of each of the letters in the stems it generates and to be able to distinguish between letters in the original radical set or in the template string. The stem generator may then be used to compile a complete list of all affixed stems from database roots while retaining all transformation information. The resulting list of stems may then be turned into a searchable index that holds the complete morphological analysis and classification for each entry.

Since Arabic words can have a maximum of four root radicals, a root radical set R is defined in terms of the ordered letters of the root as follows:

$$R = \{r_F, r_M, r_L, r_Q\} \quad (1)$$

In the database, pattern, root, variant, and voice-tense ids identify a particular morphological pattern s. Templates are used to generate a stem from a root. The text of s is defined in terms of the letters and diacritics of the template in sequence (x_1, ..., x_l) and the radical position markers or place holders (h_F, h_M, h_L, and h_Q), that indicate the positions that letters of the root should be slotted into:

$$S = x_1 \, x_2 \, ... \, h_F \, ... \, h_M \, ... \, h_L \, ... \, h_Q \, ... \, x_n \quad (2)$$

Stem Generator (SG) uses regular expressions as the language for compiling FSTs for morphophonemic transformations. Transformation rules take into account the context of root radicals in terms of their positions in the template and the nature of the template letters that surround them. Transformations are performed using combinations of regular expression rules applied in sequence, in a manner similar to how humans are subconsciously trained to process the individual transformations. The resulting template between one morphophonemic transformation and the next is an intermediate template. However, in order to aid the next transformation, the transformed radicals are marked by inserting their place holders before them. For example, h_Fرَh_Mسَh_Lمَ (FraMsaLma) is an intermediate template formed by the root radical set R = {ر, س, م } ({r, s, m}) and the morphological pattern s = h_Fَh_Mَh_Lَ (FaMaLa).

To create the initial intermediate template i_0 from the radical set R and morphological pattern s, a function Regexp (*String*, *SrchPat*, *ReplStr*) is defined to compile FSTs from regular expressions. The function accepts in its first argument a string that is tested for a match with the search pattern (*SrchPat*) in its second argument. If *SrchPat* is found, the matching characters in *String* are replaced with the replace string (*ReplStr*). This function is assumed to accept the standard PERL regular expression syntax.

A function, *CompileIntermediate* (R, s), accepts the radical set R and morphological pattern s to compile the first intermediate template i_0. A regular expression is built to make this transformation. It searches the morphological pattern text for radical place holders and inserts their respective radical values after them. Since *Regexp* performs substitutions instead of insertions, replacing each marker with itself followed by its radical value is effectively equivalent to inserting its radical value after it. Let p be a search pattern that matches all occurrences of place holders h_F, h_M, h_L, or h_Q in the morphological pattern, then an initial intermediate form i_0 may be compiled in the following manner:

$$\begin{aligned} i_0 &= CompilerIntermediate\ (R, s) \\ &= Regexp\ (s, p, pR_p) \\ &= \{x1 \, ... \, h_F r_F \, ... \, h_M \, r_M \, ... \, h_L \, r_L \, ... \, h_Q \, r_Q \, ... \, x_n\} \end{aligned} \quad (3)$$

Let T = {t_1...t_m} be the transformation rules applied on each intermediate template to create subsequent intermediate templates. Transformation rules are defined as:

$$t_j = (SrchPat_j, ReplStr_j) \quad (4)$$

A second function transform (i, t) is required to perform transformations. A subsequent intermediate template i_{j+1} is the recursive result of transforming the current intermediate template i_j with the next rule t_{j+1}. Each transformation is defined as:

$$\begin{aligned} i_{j+1} &= (t_j, t_{j+1}) \text{ for } 0 \leq j < m \\ &= Regexp\ (i_j, SrchPat_{j+1}, ReplStr_{j+1}) \end{aligned} \quad (5)$$

At any point in the transformation process, the current transformed state of radicals (R') and template string (s') may be decomposed from the current intermediate template as follows:

$$CompilerIntermediate\ (^{-1}i_j) = (R', s') \quad (6)$$

To turn final intermediate template i_m into a proper stem, a regular expression is built that deletes the place holders from the intermediate template. To do this with a regular expression, the place holders matched are replaced with the null string during the matching process as follows:

$$Regexp\ (i_m, p, null) \quad (7)$$

Basic stems are only modified for tense and voice. Additional morphosyntactic templates or affixation rules further modify proper stems for person, gender, number, and mode. Affixation rules are regular expressions like transformation rules. However, these rules modify final intermediate templates by adding prefixes, infixes, or suffixes, or by modifying or deleting stem letters. They require knowledge of the radical positions and occasionally their morphophonemic origins. Adding affixes to a stem operates on the intermediate template which retains the necessary information.

Let a be the affixation rule that is being applied to a certain intermediate template:

$$a = (SrchPat, ReplStr) \quad (8)$$

Now using the function transform that was defined earlier, affixes are added to i_m to produce the intermediate affixed template i_{m+1}:

$$\begin{aligned} i_{m+1} &= Transform\ (i_m, a) \\ & Regexp\ (i_m, SrchPat, ReplStr) \end{aligned} \quad (9)$$

To convert for output i_{m+1} to an affixed stem, one may remove place holders using the following:

$$Regexp\ Regexp\ (i_{m+1}, p, null) \quad (10)$$

With this definition, generated stems are described by intermediate templates. Intermediate templates retain knowledge of the current state of template and radical letters without losing the ability to recall their origins. This algorithm, therefore, would avoid guesswork in the identification of root radicals. Automatic rule-based stem generation and analysis are both facilitated by this feature of intermediate templates.

4. Stem Generation Engine

A stem generation engine may be built on the basis of the definition just advanced. The three components, *stem transformer, affixer, and slotter*, applied in sequence, make up SG. *Stem transformer* applies the appropriate transformation rules to the morphological pattern, *affixer* adds specific affixes to the transformed template; and *slotter* applies the radicals to the transformed affixed template to produce the final affixed stem. SG begins with a stem ID from the *maindictionary* table as input to *stem transformer* (See Figure 1). The root and entry associated with the stem ID are used to identify the radicals of the root, the morphological pattern string, a list of transformation rules, and an affix table ID.

Stem transformer applies transformation rules that are localized to the root radicals and letters of the template in the contexts of one another. To prepare the template and root for transformation, the engine begins by marking radicals in the template. *Stem transformer* is applied incrementally using the current radical set, the template string, and one transformation rule per pass, as in Figure 2. The output of each pass is fed back into *stem transformer* in the form of the jth-rule-transformed template string and radicals, along with the $(j+1)^{th}$ transformation rule. When all rules associated with the template are exhausted, the resultant template string and radicals are output to the next phase.

To illustrate, assume the morphological pattern s = إ h_F تَ $h_M h_L$ (AiFotaMaLa), the radical set R = {ذ, ك, ر} ({@, k, r}), and the transformation rule set $T = \{1, 12\}$. *Stem transformer* generates a proper stem using the following steps: Equation 3 above creates the initial intermediate template when passed the radical set and morphological template, thus producing:

$i_0 = CompilerIntermediate\ (R, s)$
= إ h_F ذْ تَ h_M كَ h_L رَ
(AiF@taMkaLra)

The first transformation rule $t_1 = 1$, $t_1 \in T$ is a regular expression that searches for a ت (t) following h_F and replaces ت (t) with a copy of r_F. To transform i_0 into i_1 with rule t_1, equation 5 is used, thus producing:

$i_1 = Transform\ (i_0, t_1)$
= إ h_F ذْ ذَ h_M كَ h_L رَ
(AiF@o@aMkaLra)

Next, a gemination rule $t_2 = 12$, $t_2 \in T$ is applied to i_1. The gemination regular expression searches for an unvowelled letter followed by a vowelled duplicate and replaces it with the geminated vowelled letter. Once more, equation 5 is used to make the transformation:

$i_2 = Transform\ (i_1, t_2)$
= إ h_F ذَّ h_M كَ h_L رَ
(AiF@~aMkaLra)

To obtain the proper stem from the intermediate template, the final intermediate template i_2 may be substituted into equation 7:

$Stem = Regexp\ (i_2, p, null)$
= اذَّكَرَ
(Ai@~akara)

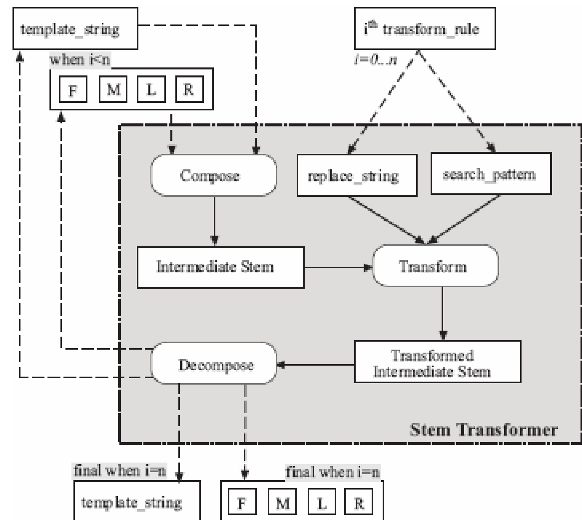

Figure 2. Stem transformer.

To summarize, the final output of *stem transformer* is a root molded into a template and a template-transformed radical set. These outputs are used as input to the affixation phase which succeeds stem transformation. *Affixer*, applied iteratively to the product of *stem transformer*, outputs 14 different subject-pronominally affixed morphosyntactic forms for every input except the imperative which only produces 5. There are 9 different tense-voice-mode combinations per subject pr onominal affix, so most roots produce 117 affixed stems per dictionary entry. *Affixer* is run with different replace strings that are specific to the type of affix being produced. It modifies copies of the transformed stem from the previous phase, as in Figure 3. Using the example cited shortly before, *affixer* is passed the last intermediate template i_m and the affix regular expression a. In this example, a is a regular expression that searches for $h_L r_L$ and

replaces it with $h_L r_L$ تْ (L$r_L$ato); this corresponds to the past active third person feminine singular affix. Now applying equation 9 produces:

i_3 = *Transform* (i_2, *a*)
= اِ h_F ذّ h_M َك h_L َر ْت
(AiF@~aMkaLrato)

In the last stage of stem generation, *slotter* as shown in Figure 4, replaces the place holders in the transformed template with the transformed radical set, producing the final form of the affixed stem. For example, the result of applying Equation 10 is:

Regexp (i_3, *p*, *null*) = اِذَّكَرَتْ
(Ai@~akarato)

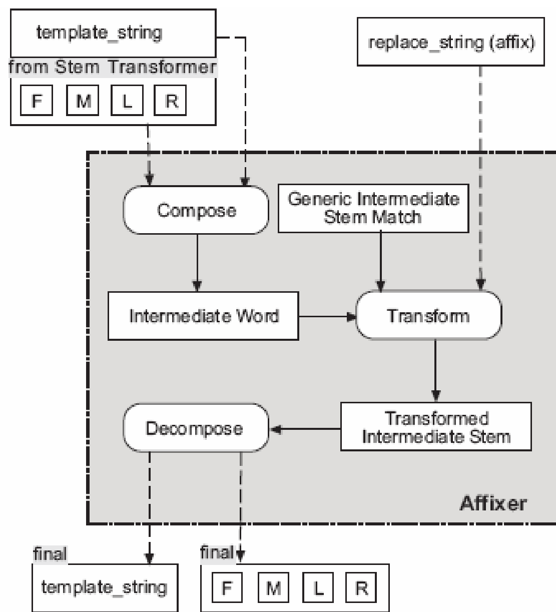

Figure 3. Affixer phase.

5. Optimization

Data produced for the use of SG was designed initially with no knowledge of the actual patterns and repetitions that occur with morphophonemic and affix transformation rules. In fact, SG is made to create stems this way: A root is added to a morphosemantic template, then morphosyntactic templates are applied to it, inducing in some patterns morphophonemic transformation. However, while this may be useful in many language teaching tools, it is extremely inefficient. The original data was used to discover patterns that would allow stems to be created in an optimal manner.

Following the classification in [8], there are 70 verb root types associated with 44 theoretically possible morphological patterns. There is an element of repetition present in the classification. In addition, the *template* table lists sequences of rules that operate on morphological patterns in a manner similar to how native speakers alter patterns phonemically. These rules could be composed into a single FST that yields the surface form.

For example, in the previous section, the morphophonemic transformation rule set $T = \{1, 12\}$ could have been written into one rule. In its non-optimized form the rule duplicates r_F in place of ت (t) creating intermediate form اِ h_F ذْ ذ h_M َك h_L َر (AiF@o@aMkaLra) and then deletes the first of the duplicate letters and replaces it with a gemination diacritic that is placed on the second repeated letter. The resulting surface form is اِذَّكَرَ (Ai@~akara). Instead, one rule could achieve the surface form by replacing the letter ت (t) in the template with a geminated ذ (@) yielding the same result.

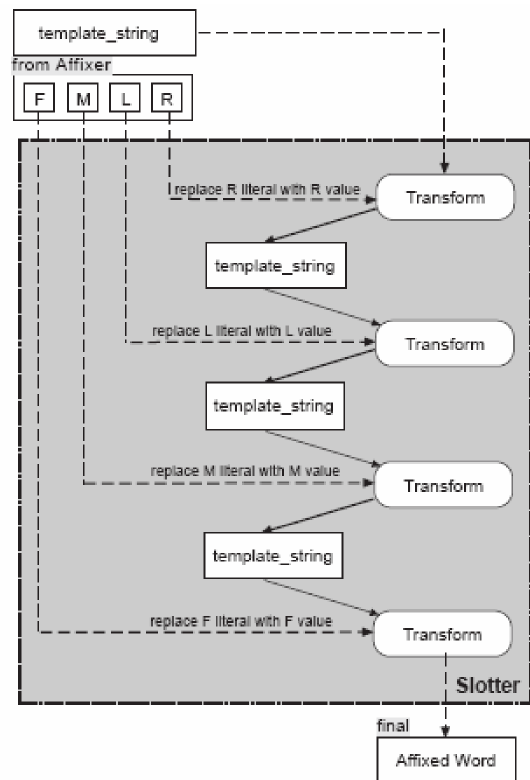

Figure 4. Slotter phase.

Compiling separate regular expressions for each transformation rule is costly in terms of processing time especially when used with back-references, as SG does. Back-references group a sub-pattern and refer to it either in the search pattern or substitute string. Such patterns are not constant and are required to be recompiled for every string they are used with. It is desirable, therefore, to minimise the number of times patterns are compiled. To optimise further, the transformation may be made on the morphological pattern itself, thus producing a sound surface form template. This procedure would eliminate the need to perform morphophonemic transformations on stems.

Each template entry in the *template* table (see Figure 1) is given a new field that contains the surface form template. This is a copy of the morphological pattern with morphophonemic transformations applied to it. Hence, a coding scheme is adopted that would

continue to retain letter origins and radical positions in the template. Any transformations that affect the morphological pattern alone are applied without further consideration. The coding scheme uses the Roman characters F, M, L, and Q to represent place holders in the templates. Each place holder is followed by a single digit indicating the type of transformation that occurs to the radical slotted in that position. The codes have the following meanings: 0 = no alteration, 1 = deletion, 2 = substitution, 3 = gemination. If the code used is 2, then the very next letter is used to replace the radical to which the code belongs.

Take for example, the *template* table entry for the root type 17 (all roots with F = و (w) and L = ي (y)), its morphological pattern اِ h_F ْتَ h_M َ h_L َ (AiFotaMaLa), and its variant (ID 0). The morphophonemic transformation rules applied to the template are T = {20, 12, 31, 34, 112}. These rules correspond to the following:

- 20 = Change r_F to a duplicate of the next letter ت (t).
- 12 = Geminate duplicate letters.
- 31 = Delete diacritic after the ي (y) in position h_L.
- 34 = Convert ي (y) to ا (A).
- 112 = Convert ا (A) to ى (Y).

The surface form template can be rewritten as اِ h_F 2تَّh_M0َ h_L2ى (AiF2t~aM0aL2Y). This can be used to form stems such as اِتَّدَى (Ait~adaY) by slotting the root {و, د, ي} ({w,d,y}).

The affix tables use a similar notation for coding their rules. Every affix rule indicates a change to be made to the surface form template and begins with a place holder followed by a code 0 or 2 unless the rule redefines the entire template in which case the entry begins with a 0. Radical place holders in affix rules define changes to the surface form template. These changes affect the template from the given radical position to the very next radical position or the end of the template, whichever is first.

Affix rules with code 0 following radical place holders signify that no change should be made to that section of the surface form template. However, a code 2 after a place holder modifies the surface form template in that position by replacing the letter that follows the code with the rest of that segment of the rule. Affix rules using code 2 after place holders override any other code for that position in the surface form template because affixation modifies morphophonemically transformed stems.

Creating affixed stems from templates and affixes formatted in this way becomes far more optimal. If a surface form template was specified as اِr_F2تَّr_M0َr_L2ى (AiF2t~aM0aL2Y) and it was to be combined with the affix rule r_L2يْ تُ مَ ا (L2yotumaA), then SG simply needs to align the affix rule with the surface form template using the place holder symbol in the affix rule and replace appropriately as in Table 1.

With the resulting affixed surface form, template SG may retain the radicals of the original root where they are unchanged, delete radicals marked with code 1 and 3, and substitute letters following code 2 in place of their position holders. If the example above is used with the root {و, د, ي} ({w, d, y}), the final stem is: اِتَّدَيْتُمَا (Ait~adayotumaA, meaning "the two of you have accepted compensation for damage").

To use the original regular expression, transformations would take an average of 18000 seconds to produce a total of 2.2 million valid stems in the database. With the optimized coding scheme, the time taken is reduced to a mere 720 seconds; that is 4% of the original time taken.

Table 1. Surface form template aligned with an affix entry rule.

Surface Form	اِ (Ai	r_F2تَّ **F**2t~a	r_M0َ **M**0a	r_L2ي **L**2Y)
Affix	(			r_L2يْ تُ مَ ا **L**2yotumA)
Combined Result	اِ (Ai	r_F2تَّ **F**2t~a	r_M0َ **M**0a	r_L2يْ تُ مَ ا **L**2yotumA)

6. Generated Stem Database Compiler

Once the dictionary database has been completed and debugged, an implementation of SG generates for every root, template, and affix the entire list of stems derived from a single root and all the possible template and affix combinations that may apply to that root entry. The average number of dictionary entries that a root can generate is approximately 2.5. Considering that each entry generates 117 different affixed stems, this yields an average of approximately 300 affixed stems per root. However, some roots (e. g., {ك, ت, ب} ({k, t, b})) produce 13 different entries, which makes approximately 1,500 affixed stems for each of such roots. The generated list is later loaded into a B-tree structured database file that allows fast stem search and entry retrieval. A web CGI has been built that would use the stem generation engine to produce all affixed stems of any given root. A section of the results of this appears in Figure 5.

7. Conclusions

In this paper, we have discussed our attempt at imitating the process used by Arabic speakers in generating stems from roots. We formulated a definition of the process, facilitating an encoding of Arabic stems. The encoding represents stems in terms of their components while still allowing a simple mapping to their final surface forms. A stem's components are a root, morphosemantic and morphosyntactic templates, and any morphophonemic alterations that the stem may have undergone. In doing so, the problem has been reduced to the much smaller task of obtaining stems for the words subject to

analysis, and then matching these against the surface forms of the pre-analyzed stems. The encoding retains most of the information essential to stem generation and analysis, allowing us to trace the various transformations that root radicals undergo when inflected. Root extractors and morphological analyzers can match an input word with a defined verb stem, then use the information in the definition to determine with certainty the stem's root and morphological pattern's meaning. The authors intend to use a similar strategy to define stems for Arabic nouns.

{و ق ي}	[0 – افْتَعَل]	هي
ماضي معلوم Past Active		اتَّقَتْ Ait~aqato
ماضي مجهول Past Passive		اُتُّقِيَتْ Aut~uqiyato
مضارع مرفوع معلوم Present Indicative Active		تَتَّقِي tat~aqiy
مضارع مرفوع مجهول Present Indicative Passive		تُتَّقَى tut~aqaY
مضارع منصوب معلوم Present Subjunctive Active		لن تَتَّقِيَ tat~aqiya

Figure 5. Output from the stem generator CGI.

Mapping from words to defined stems is now much easier. The stem generation algorithm here attempts to produce a comprehensive list of all inflected stems. Any verb may be found in this list if some simple conjoin removal rules are first applied. Conjoins are defined here as single letter conjunctions, future or question particles, emphasis affixes, or object pronominal suffixes that agglutinate to a verb stem. Because conjoins may attach to a verb stem in sequence and without causing any morphological alteration, extracting stems from Arabic words becomes similar to extracting stems from English words. In fact, many of the Arabic word analysis approaches reviewed in the introduction to this paper would yield more accurate results if applied to stem extraction instead of root extraction. It would become possible to use for this purpose conventional linguistic, pattern matching, or alg braic algorithms.

The dictionary database described here can be used to form the core of a morphological analyzer that derives the root of an input word, identifies its stem, and classifies its morphosemantic and morphosyntactic templates. An analyzer based on these principles may be used in many useful applications, some of which are detailed in [8]. Example applications include root, lemma based, and exact word analysis, searching, incremental searching, and concordancing.

References

[1] Academy A. L., *Al-Mu'jam Al-Waseet (Middle Dictionary)*, Arabic Language Academy, Cairo, Egypt, 1972.

[2] Al-Shalabi R. and Evens M., "A Computational Morphology System for Arabic," *in Proceedings of the COLING/ACL98*, Montrreal, Canada, pp. 66-72, 1998.

[3] Beesley K. R., "Finite-State Morphological Analysis and Generation of Arabic at Xerox Research: Status and Plans in 2001," *in Pceedings of ACL/EACL '2001 Arabic NLP Workshop*, Toulouse, France, pp. 1-8, 2001.

[4] El-Affendi M. A., "An Algebraic Algorithm for Arabic Morphological Analysis," *The Arabian Journal for Science and Engineering*, vol. 16, no. 4, pp. 605-611, 1991.

[5] El-Affendi M. A., "Performing Arabic Morphological Search on the Internet: A Sliding Window Approximate Matching Algorithm and Its Performance," *Technical Report*, King Saud University, 1999.

[6] Friedl Jeffery E. F., *Mastering Regular Expressions*, O'Reilly, 2002.

[7] Hamandi L., Zantout R., and Guessoum A., "Design and Implementation of an Arabic Morphological Analysis System," *in Proceedings of the International Conference on Research Trends in Science and Technology*, Beirut, Lebanon, pp. 325-331, 2002,

[8] Thalouth B. and Al-Dannan A., *A Comprehensive Arabic Morphological Analyzer/Generator*, IBM Kuwait Scientific Center, Kuwait, 1987.

[9] Yaghi J., "Computational Arabic Verb Morphology: Analysis and Generation," *Master Thesis*, University of Auckland, 2004.

[10] Yagi Sane M. and Harous S., "Arabic Morphology: An Algorithm and Statistics," *in Pceedings of the International Conference on Artificial Intelligence (IC-AI'1999)*, Las Vegas, Nevada, USA, pp. 19-25, 1999.

Sane Yagi obtained his BA from the University of Jordan, his MA from the University of Kansas, and his PhD from the University of Auckland. He is an associate professor of computational linguistics at the University of Sharjah. He taught at universities in America, New Zealand, Malaysia, Saudi Arabia, Oman, and the United Arab Emirates. His research interests include computational morphology and lexicography, computer-assisted language learning, and English education. He has co-authored number of papers in

these fields and a group of software titles that are currently in use at various educational institutions.

Jim Yaghi obtained his BInfSc degree in mathematics from Massey University, and BSc, PGDipSci, and MSc degrees in computer science from the University of Auckland. Currently, he is a researcher at DocRec Ltd, New Zealand for Arabic document recognition and reconstruction. He has published several works in Arabic computational linguistics and pattern recognition. He has also written software applications for language education, language tools, Arabic morphology and lexicography, and image and video processing and recognition.