Constraining Separated Morphotactic Dependencies in Finite-State Grammars

Kenneth R. Beesley[1]
Xerox Research Centre Europe
Grenoble Laboratory
6, chemin de Maupertuis
38240 MEYLAN, France

Abstract: [Morphology, Morphotactics, Finite State, Separated Dependencies]

This paper examines dependencies between separated (non-adjacent) morphemes in natural-language words and a variety of ways to constrain them in finite-state morphology. Methods include running separate constraining transducers at runtime, composing in constraints at compile time, feature unification, and the use of FLAG DIACRITICS. *Examples are provided from Modern Standard Arabic. In choosing a practical solution, developers must weigh the size, performance and flexibility of the overall system.*

1 Introduction

In finite-state morphotactics, the efficient constraint of separated (non-adjacent) morpheme dependencies is a serious practical challenge. This paper will examine some typical separated dependencies, using examples from Modern Standard Arabic, showing various methods that have been invented, and perhaps reinvented several times, to block lexical overgeneration. The challenge in working systems is to enforce the necessary constraints without causing the lexicons to explode in size and without slowing the runtime performance too badly.

The term MORPHOLOGY, as used by linguists in the Two-Level and Finite-State traditions, encompasses both MORPHOTACTICS (also called MORPHOSYNTAX), and the phonological or orthographical VARIATION rules that map between LEXICAL strings (i.e. abstract or underlying strings) and SURFACE strings. The theory and practical use of finite-state variation rules are well documented (Koskenniemi, 1983; Karttunen, 1983; Antworth, 1990; Karttunen and Beesley, 1992; Sproat, 1992; Karttunen, 1994) and will not be dealt with here. In the area of morphotactics, the commonly available languages for finite-state lexical specification provide linguists with a notation wherein related classes of morphemes, e.g. verb endings, noun endings, direct-object clitic suffixes, etc., are grouped together into sublexicons, and each individual morpheme is assigned a CONTINUATION CLASS which designates which subclasses of morphemes can follow it in a valid word (Karttunen, 1993). In formal terms, the grouping together of related morphemes into sublexicons translates into the union operation, and continuations translate into the concatenation operation. As far as concatenating languages are

concerned, these two finite-state operations are often sufficient for defining the language of possible lexical strings.

Where there are morphotactic dependencies, i.e. where some morphemes require or prohibit the appearance of other morphemes in a word, and where the morphemes in question are adjacent, the necessary dependencies can be constrained via appropriate definition of the continuation classes. However, when similar co-occurrence restrictions exist between morphemes that are physically separated in a word, then the continuation-class notation breaks down and must be supplemented by one of the mechanisms to be discussed below. We shall conclude with a presentation of FLAG DIACRITICS as a practical compromise that keeps lexicons small, runs efficiently, provides linguists with a notation reminiscent of feature-unification, and is compatible with general finite-state computation.

2 Arabic Morphotactics

Arabic and other Semitic languages are most notable for having a partially non-concatenative morphotactics wherein STEMS are formed by the interdigitation of ROOTS and PATTERNS, a process naturally formalized in finite-state morphology as intersection (Kataja and Koskenniemi, 1988; Beesley, 1996; Beesley, 1998). However, stem formation does not concern us here; we shall look only at noun examples, assuming that we have a sublexicon of thousands of noun stems including **kitaab** ("book"), **kutub** ("books"), **kaatib** ("scribe"), **maktab** ("office"), **miktaab** ("typewriter"), **daaris** ("student"), **mudarris** ("teacher"), **tadriis** ("instruction"), etc. Outside of the stems, Arabic morphotactics is fairly straightforward, involving prefixes and suffixes that concatenate to the stems in the usual way.

However, a too straightforward description of Arabic morphotactics will overgenerate seriously because of separated dependencies. To illustrate this, we note first that words can begin with a noun stem and end with any one of six mutually-exclusive case suffixes, schematically

```
                    +u          Definite Nominative
                    +a          Definite Accusative
                    +i          Definite Genitive
NounStem
                    +un         Indefinite Nominative
                    +an         Indefinite Accusative
                    +in         Indefinite Genitive
```

Compiled into a finite-state machine, with non-final states represented as single circles, the start state labeled **S**, the final state represented as a double circle, and transitions represented as labeled arcs, we get a diagram as in Figure 1. We also use just **kitaab** and **daaris** to represent the entire union of noun stems. Every path through the FSM represents a valid lexical word, including **kitaab+u**, **kitaab+a**, **kitaab+in**, **daaris+un**, etc.

Arabic noun stems can optionally co-occur with an overt definite-article, represented here as just **l+**, which concatenates to the stem as a prefix. The most straightforward way to represent an optional prefix in finite-state terms is as in Figure 2, where the 0 represents an empty or "epsilon" arc. However, if the overt definite article is present, then the indefinite case suffixes are in fact illegal. The new diagram overgenerates, producing illegal strings like **l+kitaab+un*.

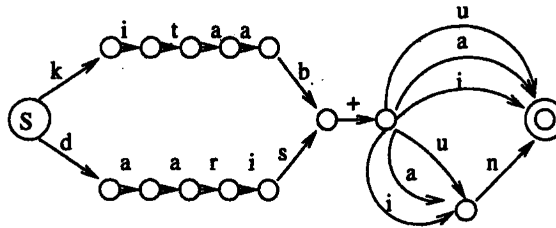

Figure 1: Noun Stems and Case Endings

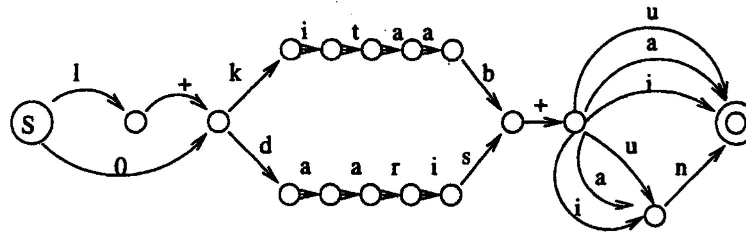

Figure 2: An Overgenerating Lexicon

To complicate the picture further, certain Arabic prepositions like **bi+** can attach as prefixes in front of **l+**, or directly to the front of the noun stems if no **l+** is present. This **bi+** by itself always requires the genitive case ending, either **+i** or **+in**. When **bi+** combines with **l+**, the only legal case ending is **+i**.

```
bi+kitaab+i
bi+kitaab+in
bi+l+kitaab+i
```

There are many more such separated dependencies in Arabic,[2] but these two will suffice.

3 Constraining Separated Dependencies

There are several quite workable ways to constrain the **l+** and **bi+** dependencies and prevent overgeneration; these include running concurrent rule transducers at runtime, composing the constraints into the lexicon at compile time, and resorting to a feature-unification system like D-PATR. All have advantages and disadvantages. We'll examine these approaches and end with a presentation of Flag Diacritics, which for our purposes provide an optimal compromise of size, performance, and notational perspicuity.

3.0.1 Running Concurrent Rule Transducers

Even if a lexicon overgenerates by itself, separate rules applying concurrently during lookup can block and eliminate illegal solutions at runtime. In a classic KIMMO-style two-level morphology, the phonological variation rules are compiled into finite-state transducers that are applied in parallel at runtime, constraining possible pairings of lexical and surface strings.

```
                     LEXICAL LEVEL

Rule1  Rule2  Rule3  Rule4  Rule5  ...  Rulen

                     SURFACE LEVEL
```

If suitable "feature" symbols are injected into the lexical strings, then separated dependencies can be treated as pseudo-phonological phenomena. Let **^Art** be defined as a single symbol (with a multi-character print name) that occurs always and only with **l+**, the definite article prefix; and let **^NeedGen** occur always and only with prefixes like **bi+**.

```
l ^Art +

b i ^NeedGen +
```

Similarly, let symbols **^Nom**, **^Acc** and **^Gen** occur always and only with nominative, accusative and genitive case suffixes respectively. And let **^Indef** mark all and only indefinite case suffixes, while **^Def** marks all and only definite case suffixes. The case-suffix strings then would look like the following (the position of the feature symbols in the strings is not important).

```
+ u ^Def ^Nom
+ a ^Def ^Acc
+ i ^Def ^Gen

+ u n ^Indef ^Nom
+ a n ^Indef ^Acc
+ i n ^Indef ^Gen
```

Overall, the resulting lexicon will generate legal strings that look like the following

```
k i t a a b + u ^Def ^Nom
d a a r i s + a n ^Indef ^Acc
l ^Art + k i t a a b + a ^Def ^Acc
b i ^NeedGen + l ^Art + d a a r i s + i ^Def ^Gen
```

and many illegal strings including

```
l ^Art + k i t a a b + a n ^Indef ^Acc
b i ^NeedGen + l ^Art + d a a r i s + a ^Def ^Acc
b i ^NeedGen + l ^Art + d a a r i s + i n ^Indef ^Gen
```

As in standard Two-Level Morphology notation, let *l:s* be a symbol pair relating a lexical symbol *l* and a surface symbol *s*. Let *l:* denote all symbol pairs from the alphabet of the rules with *l* on the lexical side (not specifying what is on the surface side); and let *:s* denote all symbol pairs from the alphabet with *s* on the surface side. The symbol *:* by itself therefore stands for any symbol pair in the alphabet. Assuming that each feature symbol like **^Art** is realized in surface strings as zero, the empty string, the necessary constraints are imposed by the following two-level rules:

```
Rule 1:        ^Art:0 /<=  _  :*  ^Indef:0 ;

Rule 2:        ^NeedGen:0 /<=  _  :*  [ ^Nom:0 | ^Acc:0 ] ;
```

Rule 1 specifies that the lexical **^Art** symbol never occurs in a string where it is followed by the **^Indef** symbol. Compiled into a finite-state transducer and consulted constantly during analysis, it "remembers" when it sees an **^Art** symbol by moving into an "I saw **^Art**" state. Once in that state, if the rule sees **^Indef** anywhere in the remainder of the word, it fails, causing the analysis to backtrack for a different solution. Similarly, Rule 2 remembers when it sees **^NeedGen**, and, having seen it, will fail if it subsequently sees **^Nom** or **^Acc**.[3]

In a typical research-oriented Two-Level Morphology, where dozens of phonological rules are consulted at every step at runtime, the addition of a few morphotactic constraint rules like Rule 1 and Rule 2 will hardly be noticed, and they work perfectly well. In some commercial environments, acceptable performance can be achieved by pre-composing and intersecting most of the component transducers of a system together into a single transducer, but reserving the constraint of long-distance dependencies to a separate filter or small set of filters which run concurrently at runtime, weeding out illegal solutions on the fly. However, in commercial applications where performance is absolutely critical,[4] running multiple transducers at runtime, or even just two, is sometimes an unattractive overhead.

3.0.2 Composing in Constraints at Compile Time

While the solution above shortstops illegal analyses at runtime, by detecting and blocking illegal paths through the lexicon, an alternative approach is simply to eliminate the illegal paths altogether at compile time. Xerox applications have traditionally intersected all the rule transducers into a single transducer; and because the lexicon also compiles into a transducer, it can be composed together with the rules into a single data object called a LEXICAL TRANSDUCER (Karttunen et al., 1992; Karttunen, 1994). If constraints are also composed into the lexical transducer at compile time, then at runtime only a single transducer is manipulated, rather than multiple transducers, with proportional improvement in the performance.

Assuming the presence of the lexical feature symbols discussed above, we can characterize one set of illegal strings in regular-expression terms as **[:* ^Art :* ^Indef :*]**. An equivalent notation, using the Xerox "contain" operator (**$**) is **$[^Art :* ^Indef]**. Another set of illegal strings is characterized as **$[^NeedGen :* [^Nom | ^Acc]]**. All the illegal strings can be notated as the union of these two languages, and the complement (**~**) of this union matches all lexical strings except the illegal ones: **~[$[^Art :* ^Indef] | $[^NeedGen :* [^Nom | ^Acc]]]**.

When this complement language (the "Filter") is composed on top of the overgenerating lexical transducer at compile time, only good strings are matched, and the unmatched illegal strings simply disappear in the process of the composition. (The composition operator is shown as **.o.**.)

```
~[ $[ ^Art :* ^Indef]  |  $[ ^NeedGen :* [ ^Nom | ^Acc ] ] ]
.o.
OvergeneratingLexiconFST
```

While this solution is formally elegant, and while the result runs very efficiently with less backtracking, it often causes the resulting transducer to explode in size. In general, structures

between the dependent morphemes get copied when the constraints are "composed into" the lexicon itself. In our example, the entire sublexicon of noun stems is copied once, for the **l+** restriction, and then that result is copied again, to capture the **bi+** restriction, almost quadrupling the final size of the transducer. In a full-scale system such an explosion may be unacceptable.

3.1 Feature Unification

Both the runtime rules and the compile-time filters require that feature-like symbols be injected into lexical strings. Some linguists dislike that practice or simply prefer to express morphotactic constraints using more abstract (and traditional) semantic features that exist in a realm separate from the phonological symbols.

Feature-unification notation, as in D-PATR, has been proposed and even implemented in a number of morphology systems for constraining morphotactic dependencies (Karttunen, 1984; Karttunen, 1986; Bear, 1986; Kataja and Koskenniemi, 1988; Beesley et al., 1989; Trost, 1990). The basic idea is that each morpheme can be assigned a set of *feature:value* pairs, and as each morpheme is identified during analysis, its feature set must unify with the unified feature set for all the morphemes previously encountered. The final feature set for a successful analysis characterizes the entire word.

This method also works well, keeping the network small, and it allows the use of powerful notational conventions that are already familiar from semi-formal linguistic description. However, this method requires a separate feature-unification mechanism to run in parallel with the usual morphological symbol processing, and this may degrade performance unacceptably.

3.1.1 Flag Diacritics

The final method to be presented here is FLAG DIACRITICS, a method that injects feature-like symbols into phonological strings but which recognizes and interprets them specially during lookup to enforce the indicated dependencies. The very finite amount of memory required is carried by the enhanced lookup process itself. Flag diacritics were inspired by the "feature requirements" of the Ment model (Blaberg, 1994) and by similar schemes in use at Xerox,[5] but related schemes have apparently been invented and reinvented many times going back to the days of ATNs (Fraser and Bobrow, 1969; Kornai, 1996).

Despite the prejudice against injecting feature-like symbols into phonological strings, there are many practical advantages in doing so. For one, this keeps the overall system limited to finite-state networks, which can be manipulated and modified freely by operations such as union and composition. And in many cases, feature symbols like **^Art**, **^Indef** and **^Nom** appearing in lexical strings can be useful information for the human user or for subsequent parsing.[6]

Flag diacritics are defined like any other multicharacter symbols in a **lexc** grammar (Karttunen, 1993), but they are always bounded by @-signs to give them a distinctive orthography that can be recognized automatically by the lookup routines. The spelling of each flag diacritic contains a first field indicating an operation, a second field indicating the name of a feature, and in most cases a third field indicating a feature value. Fields are separated by periods, schematically *@Operation.FeatureName.FeatureValue@*.

```
@C.Feat@        clear ((re)set to the neutral value),
                      cannot fail
```

@P.Feat.Val@	positive (re)set of Feat to Val, cannot fail
@N.Feat.Val@	negative (re)set of Feat to the complement of Val, cannot fail
@U.Feat.Val@	unify-test, succeeds iff the current value of Feat is compatible with Val (if Feat is neutral, then sets it to Val)
@R.Feat.Val@	require-test, succeeds iff Feat has been set to Val
@R.Feat@	require-test, succeeds iff Feat is set to some value other than neutral
@D.Feat.Val@	disallow-test, succeeds iff the Feat has been set to a value that is incompatible with Val
@D.Feat@	disallow-test, succeeds iff Feat is neutral

In many practical applications, the **U** commands are often sufficient, with the others supplied for convenience and completeness. Getting back to the Arabic examples, we can build the lexicon so that the lower-side language includes symbol @U.ART.YES@ as part of the definite article and so that the incompatible indefinite case suffixes are all marked @U.ART.NO@. In the course of analyzing a word, if a definite article is present, the lookup routine will find symbol @U.ART.YES@, interpret it as an epsilon for purposes of matching the input string, but setting and remembering that feature **ART** is set to value **YES**. If the symbol @U.ART.NO@ is found further on in the analysis path, the unification[7] will fail and the analyzer will be forced to backtrack to try to find another solution. Similarly, the **bi+** prefix can be marked @U.CASE.GEN@, with the various case endings marked @U.CASE.NOM@, @U.CASE.ACC@ or @U.CASE.GEN@ as appropriate.

The use of flag diacritics in general entails a slight runtime performance penalty, compared to composing in the same restrictions, because of increased backtracking. The flag diacritic notation is also not as powerful or perspicuous as genuine feature unification, as with D-PATR; but the runtime overhead of the simplified flag-diacritic checking is very light compared to the alternatives. Developers can also selectively remove flag diacritics from their analyzers by using the 'eliminate flag' algorithm, which simply composes the constraints into the network structure itself, with the usual penalties in size. In experiments on German, Arabic and Hungarian morphology systems, experiments often show that only a handful of restrictions cause a noticeable size explosion, and only these need to be handled with flag diacritics.

The use of just five flag diacritics reduced the size of a Hungarian morphological analyzer from over 38 megabytes to under 5 megabytes. The two systems are not completely comparable, but the benefit is even greater than these figures indicate. The 38 megabyte system had composed into it a small set of constraints chosen carefully to keep it from blowing up into an even bigger size. The 4.6 megabyte machine includes more important constraints, encoded as flag diacritics, that cannot be composed in because the size of the network becomes uncomputably large.

4 Conclusion

Pure finite-state networks have no stack or other "memory" to store information about what morphemes or features have been accumulated; each transition from one state to the next depends only on the current input symbol. Where languages have separated morphotactic dependencies, as with the Arabic **l+** and **bi+** morphemes, capturing the dependencies in a pure finite-state network requires copying the structures between the dependent morphemes, with a resulting explosion in size. To keep such systems small, some way is required to inject a tiny bit of memory into the overall system. In traditional KIMMO-style systems, the memory can be simulated in the states of concurrently running rule transducers. In a system with an auxiliary feature-unification mechanism, the memory is implemented in the unified feature sets that are calculated and passed along during analysis. With Flag Diacritics and similar mechanisms, all the feature information is represented as special symbols in the network itself, and the lookup routines are modified to recognize such symbols, perform a small set of unification-like operations, and remember the results. In working morphology systems at Xerox, Flag Diacritics have proved to be an optimal compromise, keeping networks small, providing good runtime performance, and retaining the advantages of computing with finite-state machines.

Notes

[1]Kenneth R. Beesley, D.Phil. 1983, University of Edinburgh, is a Principal Scientist at the Grenoble Laboratory of the Xerox Research Centre Europe.

[2]For example, definite case endings are optionally followed by possessive-pronoun suffixes as in **kitaab+u+hu** ("his book"), but these possessive-pronoun suffixes are also incompatible with the overt definite article, e.g. __*l+kitaab+u+hu__. The possible combinations of imperfect verb prefixes and suffixes can also be described in terms of separated dependencies.

[3]In the current grammar, where every noun string eventually gets marked __^Def__ or __^Indef__, the same constraint could be imposed by the rule __^Art:0 /<= _ \^Def:0* .#.__ where __\^Def:0__ denotes all symbol pairs other than __^Def:0__ and __.#.__ represents the end of word. Similarly, the constraint imposed by Rule 2 could also be stated as __^NeedGen:0 /<= _ \^Gen:0* .#.__

[4]Customers demanding maximum efficiency include database providers and Internet search engines that use finite-state morphological analysis, or baseform reduction, for indexing of massive document collections. With their servers running at capacity, any compromise in performance translates directly into additional expenditures for hardware.

[5]Ron Kaplan and Mike Wilkins, 1994–1995, personal communications; also http://www.xrce.xerox.com/research/mltt/fsNLP/train.html.

[6]In KIMMO-style systems, various featural or "diacritical" symbols like a stress mark (e.g. ') were often injected into lexical strings to control phenomena such as consonant doubling in English: **'budget+ing**/*budgeting* vs. **for'get+ing**/*forgetting*. The two-level rules were written to match (or not match) such diacritics so that they would fire and double the consonants only for words marked with stress on the final syllable. Such feature symbols tended to accumulate in the lexical strings, and there was no satisfactory way to get rid of them. When a user entered a string for generation, he or she would have to know the system intimately in order to include

all the diacritical symbols in the right places. However, in full finite-state morphology, not limited to two levels, lexical-level stress marks and similar symbols can do their work and then be removed trivially by a final composition of a "clean-up" transducer on top of the network, mapping them to the empty string.

[7]Flag diacritics are intentionally nonmonotonic, so the use of the term "unification" for the U operation is not quite accurate. It was kept for historical reasons.

References

Antworth, E. L. (1990). *PC-KIMMO: a two-level processor for morphological analysis*. Number 16 in Occasional publications in academic computing. Summer Institute of Linguistics, Dallas.

Bear, J. (1986). A morphological recognizer with syntactic and phonological rules. In *COLING'86*, pages 272–276. Association for Computational Linguistics.

Beesley, K. R. (1996). Arabic finite-state morphological analysis and generation. In *COLING'96*, volume 1, pages 89–94, Copenhagen. Center for Sprogteknologi. The 16th International Conference on Computational Linguistics.

Beesley, K. R. (1998). Arabic stem morphotactics via finite-state intersection. Paper presented at the 12th Symposium on Arabic Linguistics, Arabic Linguistic Society, 6-7 March, 1998, Champaign, IL.

Beesley, K. R., Buckwalter, T., and Newton, S. N. (1989). Two-level finite-state analysis of Arabic morphology. In *Proceedings of the Seminar on Bilingual Computing in Arabic and English*, Cambridge, England. No pagination.

Blaberg, O. (1994). *The Ment Model—Complex States in Finite State Morphology*. Number 27 in Ruul. Uppsala University, Uppsala.

Fraser, B. and Bobrow, D. (1969). An augmented state transition network analysis procedure. In *Proceedings of the International joint Conference on Artificial Intelligence*, Washington D.C.

Karttunen, L. (1983). KIMMO: a general morphological processor. In Dalrymple, M., Doron, E., Goggin, J., Goodman, B., and McCarthy, J., editors, *Texas Linguistic Forum*, number 22, pages 165–186. Department of Linguistics, The University of Texas at Austin, Austin, TX.

Karttunen, L. (1984). Features and values. In *COLING'84*.

Karttunen, L. (1986). D-patr: A development environment for unification-based grammars. In *COLING'86*, pages 74–80.

Karttunen, L. (1993). Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, CA.

Karttunen, L. (1994). Constructing lexical transducers. In *COLING'94*, Kyoto, Japan.

Karttunen, L. and Beesley, K. R. (1992). Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA.

Karttunen, L., Kaplan, R. M., and Zaenen, A. (1992). Two-level morphology with composition. In *COLING'92*, pages 141–148, Nantes, France.

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

Kornai, A. (1996). Vectorized finite state automata. In *Extended Finite State Models of Language: ECAI'96*, pages 36–41. European Coordinating Committee for Artificial Intelligence (ECCAI), Budapest.

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

Sproat, R. (1992). *Morphology and Computation*. MIT Press, Cambridge, MA.

Trost, H. (1990). The application of two-level morphology to non-concatenative german morphology. In Karlgren, H., editor, *COLING'90*, volume 2, pages 371–376.