Incremental Grammar Development using Finite State Tools

Mike Maxwell
Linguistic Data Consortium
3600 Market St., Suite 810
Philadelphia, PA 19104 USA
maxwell@ldc.upenn.edu

Abstract

Finite State parsing tools are generally optimized for run-time efficiency. But a field linguist needs compile-time efficiency, so that incremental changes can be made quickly as new morphemes are discovered and grammar rules revised. Using an available finite state toolkit, the Xerox xfst program, I show how incremental changes can be rapidly compiled by extracting the set of morphemes which can co-occur with a given morpheme, imposing constraints and rules on only that subset plus the new morpheme, and merging the constrained subset back into the larger lexicon.

1 Introduction

Much attention has been devoted to the development and optimization of finite state tools for "language crunching," i.e. fast processing of large quantities of text. Such tools work well for a knowledgeable computational linguist with access to grammatical and lexical resources for the target language. The linguist builds and compiles a lexicon and a morphological/ phonological grammar for the transducer on a fast machine with ample memory; these are then compiled together into a more compact form for language crunching on smaller and/or slower computers.

In contrast, relatively little attention has been devoted to incremental grammar development, such as may occur in field research on languages which have not been extensively studied.[1] In this scenario, a linguist who often knows little or nothing about finite state transducers, and less about a particular computational tool (and who may not have advanced training in linguistics either), and who may be equipped with a relatively slow computer with limited memory, begins with a very small text corpus, perhaps a few thousand words, and no grammar or dictionary. The task thus becomes one of incrementally building a dictionary and grammar based on a small (but growing) corpus. Grammar debugging is a crucial part of this task, and one which inevitably involves revisions and blind alleys.

The focus of this paper is the latter scenario: incremental development of a grammar and dictionary. In particular, I will focus on the development of a morphological grammar, which includes the following components:

- A dictionary of morphemes, possibly including allomorphs;
- Morphotactic restrictions on the morphemes;
- Morphosyntactic restrictions on the morphemes;
- Phonological restrictions on the occurrence of lexically listed allomorphs; and
- Phonological rules.

I will assume that an alphabetic writing system exists, and that it will not change during the grammar development.[2] For concreteness, I focus on the Xerox finite state system, including the

[1] One exception to this generalization is the Boas project, described in Oflazer, Nirenburg and McShane 2001. An assumption made for the Boas project which is not made here is that a trained computational linguist will be available throughout the project. Nevertheless, the techniques described in the current paper to speed up compilation should also be useful under the scenario described by Oflazer et al.

[2] This is often an incorrect assumption, since orthographies of newly written languages are nearly always in flux. However, many orthography changes can be automated using finite state techniques.

lexc and xfst programs.[3] Finally, I will not discuss the incremental discovery and implementation of morphosyntactic restrictions on morphemes (but see Maxwell, Simons and Hayashi 2002 for one approach to this). Nor will I discuss debugging a sequence of phonological rules, a topic deserving of another paper.

Section two of this paper describes incremental development of the components listed above from the field linguist's perspective. Section three describes how finite state tools can support such incremental grammar development.

2 The Field Linguist's Perspective

For commercially viable languages, it is usually feasible to equip a highly trained linguist with a fast, large memory machine, together with existing printed or computer-readable grammars and dictionaries; train the linguist in the specifics of the chosen finite state tools; and put him or her in an office for six months or a year to produce the finite state grammar.[4] In contrast, for smaller languages—particularly minority and endangered languages—the best person available may have limited training in computational linguistics. Often there are no existing grammars or dictionaries; in fact, a dictionary and a human-readable grammar may be among the desired outputs. The office may be a hut in the village, and the computer may be an older laptop with limited memory and speed.[5]

In the absence of existing grammars and dictionaries, the field linguist must discover the grammar and morphemes, a process which may extend over years. Computer tools should be usable during the entire grammar discovery process, not just at the end, after the grammar and dictionary are (ostensibly) analyzed. One tool which has traditionally proven useful in field linguistics is interlinear text analysis.[6] The linguist begins by transcribing a text, and glossing it at a relatively high level (often the sentence level), giving a 'free translation'. Next, the user may assign a rough gloss to each word; but this step is often skipped in favor of glossing at the morpheme level. For purposes of discussion, I will assume that the linguist understands the morphology and lexicon of the language well enough to mark morpheme (or allomorph) breaks (subject to revision), and assign at least tentative glosses and categories to those morphemes.

Thus, in the initial stages of analysis, the user is finding and glossing allomorphs in texts, grouping these allomorphs into morphemes, and assigning meaning to the morphemes—thereby building a morpheme dictionary.[7]

The next step is to find phonological constraints on individual allomorphs, and morphotactic and morphosyntactic constraints on the morphemes. Finally, the linguist may generalize from allomorphy constraints to phonological rules deriving surface allomorphs from underlying forms (which must also be discovered).

These are not discrete steps; even when the analysis has reached an advanced state, there is likely to be an admixture of earlier stages.

Manually marking morpheme breaks and glossing morphemes quickly becomes fatiguing and error prone. Most interlinear text processing programs (such as the widely used Shoebox program, SIL 2000) include a parser[8], which suggests parses for words in text based on a dictionary of morphemes and a morphological

[3] These programs are described in Beesley and Karttunen Forthcoming; see also http://www.xrce.xerox.-com/competencies/content-analysis/fsmbook/. The lexc program has largely been replaced by xfst, but the lexc file format described here remains the same.

[4] Another approach involves machine learning of morphology from corpora (see Goldsmith 2001 and the papers in SigPhon 2002). While not without its merits, in the case of minority languages there are seldom sufficient corpora for this methodology to be feasible. At any rate, many of the issues discussed in this paper also arise under corpora-based approaches.

[5] The techniques described were tested on a 100 MHz machine with 32 megabytes of memory, running Linux. Today's typical PC is much faster, with more memory. But people working in minority languages—particularly in the Third World—often have computers which lag far behind what the commercial or research world would consider adequate.

[6] This task is described in Simons and Versaw 1992.

[7] Often a bilingual word dictionary is more useful to the language community than a morpheme dictionary, but I will ignore this distinction here; henceforth, the term 'dictionary' will be used to refer to a linguist's morpheme lexicon.

[8] A finite state parser is actually a transducer capable of synthesis as well as analysis. The focus in this paper will however be on parsing.

grammar. The techniques described below are intended to be built into such an integrated program, but may be adapted to stand-alone use through an appropriate scripting language.

3 Finite State support for Incremental Grammar Development

The fundamental requirement for support of incremental grammar development is that small changes must be fast enough that the user can continue working with relatively little interruption. For example, if a user discovers a new morpheme while glossing text, it must be possible to quickly add it to the dictionary, so that it is immediately available for parsing the text (which may well contain further instances of the morpheme). Likewise, if the user discovers a phonological constraint on an allomorph, the grammar development system must allow the user to state that constraint, and for the parser to immediately make use of it to constrain the parses it offers.[9]

In software development, the ability to make fast incremental changes is typical of interpreters, as opposed to compilers. Finite state tools which perform heavy optimization resemble compilers, in that changes potentially force recomputation of large portions of the network, if not the entire network. However, certain kinds of changes, such as unioning two transducers, may not require such extensive recompilation. The problem then becomes one of minimizing the recompilation required when making incremental changes, by replacing an expensive recompilation with one or more less expensive recompilations. It turns out that assuming certain finite state operations have been implemented in an efficient way (as they have been in the Xerox tools), this is possible for a useful set of otherwise slow operations.

The first sub-section below discusses the phonotactic constraints on allomorphs can be represented, while the following sub-section discusses the incremental addition, deletion and modification of morphemes.

[9] Adding constraints has the potential to invalidate existing parses. Likewise, adding morphemes to the dictionary has the potential to provide a better parse (one the user didn't think of) for already parsed words. I do not discuss here the issue of how the system might validate already-parsed text.

3.1 Constraining Allomorphs

As discussed above, in early stages of analysis it may be necessary to posit allomorphs, rather than deriving allomorphs from underlying forms. I therefore begin with the definition of allomorphs and their phonotactic constraints in finite state terms, since imposing those allomorphy constraints is a large factor in slowing the addition of new morphs or morphemes to the lexicon.

The analysis of allomorphy may be divided into three stages:

1. At first, the linguist may deal just with allomorphs and their conditioning environments.
2. Later, he begins to use phonological rules to derive some allomorphs from underlying forms, while other allomorphs will still be described distributionally.
3. There may be a third stage in which all allomorphs (apart from suppletion) are derived by phonological rules.

Computational tools should support all three stages of analysis. Moreover, there are cases where allomorphy cannot be plausibly described in terms of phonological rules (Carstairs 1990, Maxwell 1996); in such a language, the analysis can never reach stage (3).

The Xerox tools are really intended for stage (3), i.e. for rule-based allomorphy. Fortunately, xfst can handle stages (1) and (2) as well. The following sub-sections describe how distributionally based allomorphy can be described in the Xerox system, while section 3.2 describes how incremental changes can be handled efficiently.

Allomorph Environments

A common notation for allomorph distribution is shown in the following example. Suppose we have a lexeme meaning 'dog' with three disjunctively ordered allomorphs:

```
  foo  / LEnv1 __ REnv1
  foot / LEnv2 __ REnv2
  fee  / elsewhere
```

The disjunctive ordering implies that the second allomorph will appear in the environment `LEnv2__REnv2`, *except* where the environment also matches `LEnv1__REnv1`, in which case the first allomorph must appear. The third allomorph—the *elsewhere* case—appears wherever neither of the other allomorphs appear.

In the Xerox system, the lexicon is usually stored in lexc files. Stems with different parts of speech, or affixes which have different morphotactics, belong to different continuation classes, represented as follows:[10]

```
Lexicon <InLexicon>
  gloss1:form1 <OutLexicon>;
  gloss2:form2 <OutLexicon>;
  ...
```

But this accounts only for morphotactics; there is no provision in the lexc formalism for listing the conditioning environments of allomorphs, i.e. the phonotactics. Thus, for the lexeme meaning ‘dog’ in the above example, the allomorphs must be listed in the lexc file as follows (assuming for illustrative purposes that noun roots are followed by suffixes marking number):

```
Lexicon NounRoot
   dog:foo      Number ;
   dog:foot     Number ;
   dog:fee      Number ;
   ...
```

The phonological constraints on the allomorphs must be represented separately, in an xfst file. In order to capture the disjunctive ordering implicit in the conceptual notation above, two constructs are necessary in xfst: restriction rules, and filters. Restriction rules take the form

```
<allomorph> =>
     <LeftEnvironment_1> _
     <RightEnvironment_1>
```

while filters take the form

```
~[?* <LeftEnvironment_2>
     <Allomorph>
     <RightEnvironment_2>
  ?*]
```

If the left environment of the filter begins with a word boundary, or the right environment ends with a word boundary, the corresponding ‘`?*`’ is omitted.[11]

The allomorph is represented by a string (identical to the form listed in the lexc file), and the left and right environments are regular expressions. Then for the above example, we have the following in xfst:

```
{foo} => LEnv1 __ REnv1
  !'foo' appears only in
  ! this environment
{foot} => LEnv2 __ REnv2
  !'foot' appears only in
  ! this environment
~[?* LEnv1 {foot} REnv1 ?*]
  !…except not in environ
  ! for 'foo'
!'fee' can appear anywhere,
! except:
~[?* LEnv1 {fee} REnv1 ?*]
  !…in environ for 'foo'
~[?* LEnv2 {fee} REnv2 ?*]
  !…or in environ for 'foot'
```

(Curly braces are used in an xfst file around a string representing a sequence of phonemes; this convention is not used in a lexc file.) Note that some of these filters may be redundant (if two allomorphs have mutually distinct environments of occurrence, or if for the third allomorph, not being able to occur in the first environment implies not being able to occur in the second environment). But this usually does no harm, apart from making compilation a bit slower.

However, there is a complication: the restriction rules and filters should apply only to entire allomorphs, not to pieces. That is, the restriction that the allomorph *foo* appears only in some environment should not apply to another morpheme *fool*, or indeed to the allomorph *foot* (should the ‘t’ be compatible with `REnv1`). This can be accomplished in part by the use of morpheme boundary markers around each allomorph. Thus, the lexc entries for roots become:

```
Lexicon NounRoot
   dog:#foo#    Number ;
   dog:#foot#  Number ;
   dog:#fee#    Number ;
   ...
```

[10] The use of glosses here and elsewhere is for expository purposes, since their use results in a much larger network than would the use of an underlying form which more or less resembles the surface form.

[11] There is a notational shorthand, namely `~$[<left environment> <allomorph> <right environment>]`, meaning any string *containing* the specified sequence is disallowed. However, this is not usable for the case where the environment mentions a word boundary, since word boundaries are not treated in xfst as characters. They must instead be represented by the absence of a “?*”at the left or right-hand end of the environment.

Lexical entries for infixes, like those for roots, would contain a boundary marker to the left and right. Since it is only necessary to place a single boundary marker on each side of each root and affix, it is sufficient to include in the lexicon a boundary marker to the left of prefixes and to the right of suffixes. The resulting notation is similar to that used in SPE (Chomsky and Halle 1968). Thus lexical entries for a prefix and a suffix would look like this:

```
Lexicon NounPrefix
   REPET:#re    Noun ;
   ...
Lexicon NounSuffix
   PROG:ing#    # ;
   ...
```

Regular expressions describing allomorphy constraints on the above roots in xfst are then:

```
{#foo#} => LEnv1 __ REnv1 ;
..!Allomorph 'foo' appears
  ! only in this environment
{#foot#} => LEnv2 __ REnv2;
  !Alomorph 'foot' appears
  ! only in this environment
~[?* LEnv1 {#foot#} REnv1
  ?*];
  !…but not in environment
  ! for 'foo'
!Alomorph 'fee' can appear
! anywhere, except:
~[?* LEnv1 {#fee#} REnv1
  ?*]
  !…in environ for 'foo'…
~[?* LEnv2 {#fee#} REnv2
  ?*]
  !…or in environ for 'foot'
```

The regular expressions for affixes would be similar, referring to the boundary marker on both sides (one of which belongs to the affix, and one to the adjacent morpheme).

An additional xfst rule erases the boundary markers at the end of the derivation:

```
# → 0 ;
```

Homographic Allomorphs

Things get still more complicated. Suppose there are two (or more) lexemes with (at least) one homographic allomorph between them, but where the allomorphs of the two lexemes have different phonological restrictions (whether correctly, or as an artifact of analysis). It then becomes necessary to distinguish homographic allomorphs. Unfortunately, the gloss given in the lexc lexicon file is invisible to xfst rules. Fortunately, it is possible to tag the forms in lexc with diacritics which are visible to xfst. The sample lexc lexicon file would look like this, where I have added a verb allomorph which is homographic to the noun allomorph *foo*:

```
Multichar_Symbols  +Sg +Pl
   %^H1 %^H2
Lexicon Root
   NounRoot;
   VerbRoot;
Lexicon NounRoot
   dog:foo%^H1 NounSuffix;
   dog:fee     NounSuffix;
Lexicon VerbRoot
   run:foo%^H2 VerbSuffix;
   run:fum     VerbSuffix;
```

The diacritic tags `^H1` and `^H2` distinguish the allomorphs of the two morphemes. The following regular expressions implement the distinct phonological conditions on the allomorphs:

```
{foo}%^H1 =>
  LEnv1/Flags __ REnv1/Flags
  ~[?* LEnv1/Flags {fee}
       REnv1/Flags ?*]
{foo}%^H2 =>
  LEnv2/Flags __ Renv2/Flags
  ~[?* LEnv2/Flags {fum}
       REnv2/Flags ?*]
```

The added '/ Flags' means that the regular immediately preceding expression may contain any number of 'Flags' (a constant defined to include all homograph tags). A final rule deletes the tags once they have done their job.

The use of homograph flags requires keeping track of homographs in the lexicon. Allomorphs which do not have homographs do not need flags, but it may be simpler to assign flags to all allomorphs, so that it does not become necessary to add a flag to an existing allomorph when a new homographic allomorph is added. Note that homograph flags can be re-used, e.g. if there were no homographs in the entire lexicon, all allomorphs could use the same flag.

4 Additions and Deletions

In the process of glossing text, a user will inevitably encounter new morphemes, and new allomorphs of morphemes. Ideally, these should be added to the lexicon immediately, since if they appear in a particular text, there is a good chance they will appear more than once in that text. The implication is that addition of morphemes must be reasonably fast.

The user may also decide that previous decisions about morphemes are incorrect—either a morpheme does not exist, or (more likely), a previously added morpheme must be changed.

The obvious way to add, delete or change morphemes is to re-compile the entire lexicon. In the early stages of analysis, new morphemes are likely to be discovered frequently. But because the lexicon is small at this stage, recompilation will generally be fast. As the lexicon grows, however, while the need for adding new lexemes presumably decreases, recompilation becomes slower and more memory intensive. At some point, recompilation may become impractically slow, particularly if there are significant constraints among morphemes.

Timing and memory usage during (re-)compilation can be adversely affected if a large number of morphosyntactic constraints between morphemes are incorporated into the finite state grammar, particularly if these involve constraints between prefixes and suffixes. An alternative approach is to check morphosyntactic constraints at run time, e.g. by using a more traditional parser to build structure and check feature constraints over the morphemes that the finite state transducer finds, or by using the diacritic flag technique of the Xerox tools. However, I will only be considering morphotactics here, not morphosyntax.

Incremental Addition of Morphemes

When a new morpheme (or allomorph) is added, allomorphy constraints and phonological rules are often the limiting factor to rapid recompilation, since the allomorphs which can appear to the right or left of the new morpheme must be determined. As analysis progresses, a growing proportion of new morphemes will be roots, rather than affixes. This turns out to be fortunate: apart from compounding and incorporation, what appears to the left and right of a root is affixes, and natural languages have more roots than affixes. The addition of roots therefore involves fewer changes to existing morphemes in the lexicon than does the addition of affixes.

Morphotactics of Additions

As mentioned above, recompilation of the lexicon as each new morpheme is added can be unacceptably slow. Fortunately, recompilation of the entire lexicon can be avoided by including a variable in each continuation class in the lexicon file. These variables act as place holders where new morphemes can be added, as in the following excerpt from a lexc file:

```
Multichar_Symbols
  +NounRoot #MoreNs
Lexicon +NounRoot
   fox:zorro   #;
   #MoreNs     #;
```

The place-holding variable here is `'#MoreNs'`. The following xfst code can then be used to add a new noun, with two allomorphs, to the compiled version of the above file. (Homographs will be treated below.)

```
/*First splice in the
  allomorphs:*/
define ADD [
     {dog}:{sin}
    |{dog}:{sim}
    | %#MoreNs];
substitute defined ADD for
     %#MoreNs;
/*…then define their
 phonological constraints:*/
read regex {sim} =>
       _ BilabialC ;
read regex
 ~[?* {sin} BilabialC ?* ];
compose net;
```

The place-holding variable appears again at the end of the 'or' list in the 'define' line, so that it can be used for further additions of noun roots.

Allomorphy of Additions

The use of the place-holding variables in the lexc file adds a complication to the allomorphy statements: until an actual morpheme (that is, an allomorph) is spliced into the network in place of the variable, we don't know what allomorphs it can appear with. For example, if we were analyzing English and wanted to splice in new adjectives, we would not know in advance what allomorph of the *in-* prefix they would take. Therefore the constraints on *in-* must be reapplied after each new adjective is added.

In more detail: the xfst statements for the English *in-* prefix might look like this:

```
define ContSym
  ["#MoreADJs"
  |"#MoreNs"
  ];
define HomNumbers
   [%^H1|%^H2];
read regex {#il#}%^H1 =>
  _ [ l /HomNumbers
    |ContSym] ;
read regex {#ir#}%^H1 =>
  _ [ r /HomNumbers
    |ContSym];
read regex {#im#}%^H1 =>
  _ [[Labial]/HomNumbers
    | ContSym];
read regex
  ~[[ ?* {#in#}%^H1 l
    ?*]/HomNumbers];
read regex
  ~[[ ?*  {#in#}%^H1 r
    ?*  ]/HomNumbers];
read regex
  ~[[ ?*  {#in#}%^H1
  [Labial]?*]/HomNumbers];
```

This allows the *il-* allomorph to appear before an *l*, or before any place-holding variable, and similarly for the other allomorphs.

When a real morpheme (or allomorph) is added, the constraints on allomorphs of co-occurring morphemes must be imposed. For example, on adding the root *possible*, it is necessary to ensure that only the *im-* allomorph appears with it. Since phonological processes such can apply over long distances, this could mean the potentially time consuming application of all the constraints to the entire lexicon.

Speeding up this process of adding a new morpheme involves the following steps:

1. Selecting the subset of morphemes which can co-occur with the new morpheme;
2. Adding the new morpheme to that subset to create a mini-lexicon;
3. Imposing the constraints on just the mini-lexicon; and
4. Merging the mini-lexicon back in.

Fortunately, these steps are each fast.

Given that derivational affixes can change the part of speech of the word, it might seem that we could not know in advance which morphemes can co-occur with a given stem (step (1)). However, the grammar itself can tell us this. We first tell xfst to create a network containing only placeholder variables. If `ContSym` is a variable bound to a list of the continuation placeholders, and `Lex` is bound to the current lexicon, then the following commands

```
read regex
   ContSym+ .o. Lex ;
define PlaceHoldersOnly;
```

will result in such a network, i.e. "words" consisting only of placeholders.

We then eliminate all paths except those not containing the class of the morpheme to be added. For example, the following command would display 'words' containing noun roots:

```
read regex [?* "#MoreNs" ?*]
  .o.
  PlaceHoldersOnly;
```

Applying this to a network for a hypothetical agglutinative language marking case, gender and number as suffixes on nouns, the command '`print words`' might output the following:[12]

```
#MoreNs
#MoreNs#MoreNUM
#MoreNs#MoreGENDER
#MoreNs#MORE-
   GENDER#MoreNUM
#MoreNs#MoreCASE
#MoreNs#MORE-
   CASE#MoreNUM
#MoreNs#MORE-
   CASE#MoreGENDER
#MoreNs#MoreCASE#More-
   GENDER#MoreNUM
```

Next, we eliminate the instances of the given class, *unless* words of this class can co-occur with themselves. For example, to see the words which co-occur with noun roots, we eliminate the noun root placeholder itself, *unless* the language allows compound nouns. The following com-

[12] The set of words can be cyclic, e.g. a nominalizer can attach to a verb, followed by the attachment of a verbalizer. Fortunately, `print words` is smart enough to output a finite list of words. As will be seen in a moment, the result is still sufficient to capture all co-occurring morpheme classes.

mand does this, deleting the first instance of "#MoreNs" in each word:

```
read regex
 RestrictedPlaceholders
 .o.
 ["#MoreNs" -> 0 ||
   .#. [?-"#MoreNs"]*_];
```

Applied to the above output, this gives:

```
#MoreNUM
#MoreGENDER
#MoreGENDER#MoreNUM
#MoreCASE
#MoreCASE#MoreNUM
#MoreCASE#MoreGENDER
#MoreCASE#MoreGENDER#MoreNUM
```

Finally, the following command prints out one instance of each placeholder which can co-occur with the specified class:

```
print labels;
```

Given the above sample data, this outputs:

```
#MoreGENDER
#MoreCASE
#MoreNUM
```

The above steps can be done in a single xfst command, without intermediate variables.

Once the grammar has told us which morphemes co-occur with a given morpheme, there are three ways to reduce what needs to be computed on adding a new morpheme:

1. Reduce the number of morphemes whose allomorphy co-occurrence constraints need to be checked against a new morpheme.

2. Reduce the number of allomorphy constraints that need to be checked.

3. Reduce the application of phonological rules.

I will consider these points in turn.

Point (1) results straightforwardly from knowing the classes of morphemes which can (morphosyntactically and morphotactically) co-occur with the new morpheme. The relevant classes can be stored in un-compiled or semi-compiled form, to be used as needed.

Point (2) implies that we can extract the subset of allomorphy constraints relevant to a particular morpheme class. Recall that morphemes and their classes are defined in lexc files, while allomorphy constraints are given in xfst files. However, if both are extracted from a dictionary in some other form (such as an XML file), filtering constraints by class is straightforward.

As for point (3), in most cases it is not possible to reduce the number of phonological rules which need to be applied, since phonological rules by definition apply without regard to individual morphemes (apart from strata or exceptional rule marking). However, the techniques described above reduce the number of morphemes to which those rules must be re-applied. That is, the phonological rules need only be applied to the underlying forms of words resulting from the known small set of morphemes selected under point (1) above.

A further complexity arises when a new allomorph is added to a morpheme already in the lexicon; this situation will be addressed below.

Summarizing thus far, the addition of new morphemes can be sped up by determining which morphemes can co-occur with the new morpheme. Just those morphemes are then combined with the new morpheme to create a mini-lexicon, to which the appropriate constraints and (all) phonological rules can be applied. The resulting lexicon is then unioned back in with the main lexicon, a fast operation.

Both the time and memory needed for naïve additions, and the savings in time and memory obtained by the methodology outlined here, depend heavily on the size of the lexicon, and even more so on the nature of the grammar constraints and rules in a particular language. Hence it would be fruitless to give exact numbers here. Suffice to say that With substantial lexicons, the process of incremental additions described here can be an order of magnitude faster than recompiling the entire lexicon, as shown by tests with sample data. (The lexicons tested were in the range of several thousand morphemes.) But since the constraints operative in a particular language are generally unknown in advance, the methodology described here results in significant savings in time and memory.

Incrementally Deleting a Morpheme

Deleting a morpheme is straightforward. The following xfst command removes the morpheme glossed 'dog' (and all its allomorphs) from a lexicon bound to the variable LEX:

```
read regex
  ~$[{dog}] .o. LEX;
```

Incrementally Changing Allomorphs

Returning now to the issue of modifying the allomorphs of an already loaded morpheme: this can be done by first deleting the morpheme, taking with it all its existing allomorphs, and then adding the morpheme and both its old and new allomorphs and constraints back in. The reason for deleting the old allomorphs is that some of them may be restricted to not occur in the environment of the new allomorph. If the new allomorph were simply added in without changing the old allomorphs, there might (incorrectly) be environments where both allomorphs could occur. Removing the old allomorphs prevents this.

The deletion of the existing morpheme must of course be done to the entire lexicon, but this is fast; the addition of the new morpheme, with all its allomorphs, is done as described earlier.

5 Conclusion

The Xerox Finite State tools are optimized for run-time efficiency, which can conflict with a field linguist's need for compile-time efficiency. I have described a work-around which allows rapid incremental changes. The work-around consists of extracting the set of morphemes which can co-occur with a given morpheme, imposing constraints and rules on only that subset plus the new morpheme, then adding the constrained subset back into the larger lexicon. This also allows the rapid deletion and modification of existing morphemes and their allomorphs.

Acknowledgements

Much of the work described here was completed under the auspices of SIL International, whose support is gratefully acknowledged. Thanks to Ken Beesley for suggesting the use of the 'substitute defined' command to add morphemes.

References

Beesley, Kenneth R.; and Lauri Karttunen. Forthcoming. *Finite State Morphology: Xerox Tools and Techniques*. Stanford: CSLI.

Carstairs, Andrew. 1990. "Phonologically Conditioned Suppletion". In *Contemporary Morphology*, eds. Wolfgang U. Dressler; Hans C. Luschutzky; Oskar E. Pfeiffer; and John R. Rennison, x, 320. Berlin: Mouton de Gruyter.

Chomsky, Noam; and Morris Halle. 1968. *The Sound Pattern of English*. New York: Harper & Row. [Reprinted in 1991, MIT Press]

Goldsmith, John. 2001. "Unsupervised Learning of the Morphology of a Natural Language". *Computational Linguistics* 27:153-198.

Maxwell, Michael B. 1996. "Two Theories of Morphology, One Implementation". *1996 General CARLA Conference*, 203-230, Waxhaw, NC.

Maxwell, Michael B.; Gary Simons; and Larry S. Hayashi. 2002. "A Morphological Glossing Assistant". *Proceedings of the International LREC Workshop on Resources and Tools in Field Linguistics*, Las Palmas, Spain.

Oflazer, Kemal; Sergei Nirenburg; and Marjorie McShane. 2001. "Bootstrapping Morphological Analyzers by Combining Human Elicitation and Machine Learning". *Computational Linguistics* 27:59-85.

SigPhon ed. 2002. *Workshop on Morphological and Phonological Learning*. New Brunswick, NJ: ACL.

SIL. 2000. *The Linguist's Shoebox: Tutorial and User's Guide*. Waxhaw, North Carolina: SIL.

Simons, Gary; and Larry Versaw. 1992. *How to Use IT: A Guide to Interlinear Text Processing*. Dallas: Summer Institute of Linguistics.