Accounting for Allomorphy in Finite State Transducers

Michael Maxwell
University of Maryland, College Park MD 20742 USA
mmaxwell@umd.edu

Abstract

Building morphological parsers with existing finite state toolkits can result in something of a mis-match between the programming language of the toolkit and the linguistic concepts familiar to the average linguist. We illustrate this mismatch with a particular linguistic construct, suppletive allomorphy, and discuss ways to encode suppletive allomorphy in the Stuttgart Finite State tools (sfst). The complexity of the general solution motivates our work in providing an alternative formalism for morphology and phonology, one which can be translated automatically into sfst or other morphological parsing engines.

1 Introduction

While many morphological transducers have been built using finite state tools such as the Xerox Finite State Tool (xfst and lexc, (Beesley and Karttunen, 2003)) and the Stuttgart Finite State Tools (sfst, (Schmid, 2005)), linguists with whom we have worked, who are mostly not computer scientists or even computer programmers, have found such a project daunting. We have therefore created a descriptive mechanism that more closely models views of morphology and phonology that linguists are already familiar with. The resulting formal descriptions are automatically translated into the programming language of a finite state transducer (currently, sfst). For this XML-based descriptive mechanism to work, the constructs of the model must be automatically mappable into the constructs available in the transducer, and this mapping must work for all instances of such constructs.

Many linguistic structures indeed have a fairly straightforward mapping into the formalism of finite state technology. Phonological rules (in rule-based theories of phonology) for example map reasonably well to replace rules in xfst and sfst (although phonological rules expressed in terms of phonological features would not map so easily). In such a case, the XML formalism is essentially syntactic sugar.

However, not all linguistic structures have such a straightforward mapping. Some structures are simply beyond the reach of finite state systems; recursive syntactic structures are an obvious example. Within morphology, full (unlimited) reduplication is another, although xfst provides a work-around for this. In some cases, however, a linguistic structure may be finite state, but still difficult to express in a natural and general way using existing finite state formalisms.

This paper describes one linguistic phenomenon, suppletive allomorphy, which has proven difficult to reliably encode in sfst. In the next section I describe some desiderata for a linguistically based descriptive system. The following section explains how suppletive allomorphy works, and the section after that describes an algorithm we initially tried in our attempts to treat suppletive allomorphy, but which gave wrong results in some cases. The final algorithm works correctly, and we have implemented it to build parsers from grammars encoded in our linguistic formalism.

2 A descriptive system for morphology

Our descriptive mechanism is an XML-based representation of morphology and phonology, which readily accomodates most morphological and rule-based phonological structures (as character or phoneme-based representations, not as feature-based representations). A converter (written in Python) translates this linguistic representation into the sfst code needed to build a parser. This system has been described elsewhere ((Maxwell and David, 2008; ?; ?; ?); see also (Maxwell, accepted)), and we will not go into details here. Suffice to say that the mechanism covers the following sorts of linguistic structures: fusional and agglutinative morphology with both prefixes and suffixes; affix processes such as reduplication[1]; extended

[1] The descriptive mechanism handles most forms of reduplication, although underapplication is a problem, as it is in theoretical linguistics ((Inkelas and Zoll, 2005)). There are some issues with the treatement of reduplication in finite state transducers, some of a theoretical nature (unbounded reduplication is not finite

exponence (morphosyntactic features realized on more than one affix, implying a sort of agreement between affixes); inflection classes; stem and affix allomorphy governed by phonological rules (with rules being optionally sensitive to exception features); suppletive stem and affix allomorphs; suppletive word forms (requiring blocking of hyperregular forms); and dialectal and spelling variation.

For the working linguist who does not consider him or herself to be a programmer, the XML representation offers several advantages over encoding grammars in the programming language of some finite state transducer:

Software independence: The use of XML means that we can create formal grammar specifications which are independent of any particular transducer.

Longevity: Software independence means our formal grammar descriptions will be portable into the next generations of software.

Linguistic basis: By basing our XML schema on linguistically recognized concepts, we ensure that the resulting descriptions are linguistically sound.

Theory agnosticism: At the same time, adhering too closely to a particular linguistic theory would threaten the longevity of data encoded in the schema, and thereby limit the potential audience of users to those linguists who knew (and liked) that theory. Our schema therefore attempts to follow the notion of Basic Linguistic Theory ((Dixon, 2009b; **?**)).

Alternative analyses: It not always possible to provide a schema that is general enough to accomodate a wide variety of theories; instead, the schema must provide optionsdifferent ways to analyze phenomenawhich jointly allow for different theoretical approaches, or different analyses within a single approach.

Ease of use by linguists: A linguistically-based description language allows linguists to construct grammars in a way that should already be familiar to them, making it easier for them to build and maintain parsers. This is particularly evident where a linguistic structure is not straightforwardly mappable into a finite state transducers programming language: our system is adapted to the user, rather than the user having to adapt to the programming language.

These desiderata are used in later sections of this paper to motivate design decisions.

3 Suppletive allomorphy

Suppletive allomorphy is phonologically conditioned allomorphy (that is, it is not driven by inflection classes) for which phonological processes cannot reasonably be posited. Carstairs ()[p. 21]Carstairs1987 and Paster (()paster2009) give examples, of which one is the Turkish passive suffix. This takes the form n after a vowel-final stem, and l elsewhere. While this *can* be expressed using phonological rules, it is not natural to do so, and attempting to do so could easily lead to improper application of the rule elsewhere.

Generative phonologists often avoid the use of allomorphs, assuming instead a single underlying form of each affix, and generating alternative forms of morphemes by the use of phonological rules. Where such rules can be easily defined, and when they handle the allomorphy of a large number of morphemes (for instance stem allomorphy, or suffixal allomorphy due to vowel harmony, as in Turkish), this makes sense. But phonological rules have a drawback: if not written carefully, they can apply to the wrong forms. Working linguists often use allomorphy in situations where they could write a reasonable phonological rule, but have not gotten around to writing it yet, or because the rule would require refinement in order to prevent incorrect application. In fact even generative linguists debate the boundary between suppletive allomorphy and allomorphy that can be attributed to general phonological rules ((Kiparsky, 1996; Bonet and Harbour, 2012).) This situation thus presents an example of our desideratum mentioned in section 2, namely to allow alternative descriptive methods, rather than forcing the linguist to model a phenomenon in a particular way.

Hence, the need for our system to handle suppletive allomorphy has three motivations: some allomorphy cannot reasonably be treated with phonological rules; even where it can, the linguist may simply prefer to use suppletion; and even if the linguist may eventually come up with appropriate phonological rules, rendering the use of suppletion unnecessary, we wish the system to be usable from the beginning of the linguist's work.

Allomorphs by definition appear in mutually exclusive phonological environments. While it is always possible to state each allomorphs environment as mutually exclusive, it is often easier to represent them as a sequence from most specific to least specific; the first allomorph in this list whose environment is satisfied in a particular inflected wordform is chosen. The last allomorph in such a list is usually an elsewhere case, chosen if none of the preceding allomorphs in the list have environments which are satisfied ((Kiparsky, 1973)).

As an example, consider the English noun plural suffix. This suffix has three phonologically conditioned allomorphs: a voiced /z/, a voiceless /s/, and /z/.[2] While these may be derived by phono-

state) and some of a more practical nature (lengthy reduplicated strings can result in slow compilation and large finite state networks).

[2] We will not discuss unusual plurals, such as the en

logical rules (which could also handle the verbal third person singular and the possessive clitic, and perhaps the verbal past tense ed), for illustrative purposes assume the allomorphs are to be represented suppletively:[3]

/z/ : (s—z————) __
/s/ : (p—t—k) __
/z/ : elsewhere
(after b, d, hard g, vowels...; but not after s, z, ...p, t, k)

The order is important. In particular, the elsewhere case must apply last, lest it bleed the application of the other cases. This is represented here as extrinsic ordering. The first and second cases do not need to be ordered in this character-based formulation.[4]

It is not immediately obvious how to express such a list in finite state terms. In general, one of two methods might be used: either the allomorphs could be tested one by one on each wordform, and the first one that matches would be chosen; or each allomorph in the list could be allowed to occur in its phonological environment, and also forbidden to occur in any of the environments of the allomorphs preceding it in the list.

4 Implementing suppletive allomorphy

In this section, I will first discuss the first implementation of suppletive allomorphy that we did, and show why it fails in a way that may not be immediately obvious. Following that, I describe our working implementation.

4.1 Implementation 1: Realizational implementation

Our first implementation was quite simple. The underlying form is represented by the stem plus a unique symbol for each affix; these symbols are then realized by (converted to) the allomorph appropriate to their phonological environment. The unique symbol can be the affixs gloss, which is typically something like ¡PL¿ or ¡1.Sg¿; the angled brackets indicate to sfst that this is a single symbol. Algorithm 1 implements this.

```
Initialization: Initialize Lexicon to contain an
underlying representation wordform of each
cell in the paradigm of each Stem in the
dictionary, where each wordform consists of a
stem and its affixes according to the
morphotactics of the language, and with each
Affix represented by a unique label.
Then:
foreach Affix in Slots do
    foreach Allomorph in Allomorphs of
    Affix do
        if the phonological environment
        specified by Allomorph is satisfied
        then
            replace Affix with Allomorph
        end
    end
end
return Lexicon.
```

Algorithm 1: Realizational implementation

Suppose the lexicon consists of the noun stems {/bot/ boat, /kls/ class, and /tri/ tree}; and suppose that there is one affix, PL, with allomorphs [z /(s—z————)__, s /(p—t—k)__, and z /elsewhere]. Then the set of underlying plural forms will be {/klsPL/, /botPL/, /triPL/}. Applying algorithm 1 to /klsPL/, the environment of the first allomorph /z/ is satisfied, hence /klsPL/ becomes /klsz/. For /botPL/, the environment of the /z/ allomorph is not satisified, but that of the /s/ allomorph is; hence the result is /bots/. Finally, for /triPL/ neither the environment of /z/ nor that of /s/ is satisfied, so the elsewhere case gives /triz/. The results for this simple case are correct.

Unfortunately, this simple implementation will fail if there is more than one order (layer) of affixes, there are allomorphs in at least two of those orders, and the choice of allomorphs in each order depends on the allomorph chosen in another order. For example, consider a hypothetical agglutinating language in which the first order suffix /an/ assimilates in point of articulation to a following consonant, while the second order suffix /za/ assimilates in voicing to a preceding consonant. Assuming consonants in this language are labial, coronal or velar, the first suffix would have allomorphs /am/, /an/, and /a/, while the second suffix would have allomorphs /za/ and /sa/. While it is easy to write the replacement rules for each affix individually that would generate the correct allomorphs, it is not possible to simultaneously condition each on the output of the other.

Does such a hypothetical case occur in reality? I have not been able to find clear examples. However, several facts combine to make it difficult to discount the problem. First, it is true that affixes are usually inwardly phonologically sensitive. That is, the choice of a suffix allomorph more often de-

of *oxen*; nor irregular plurals such as *geese, mice.* I also ignore stems ending in /f/, since in many words this is /v/ in the plural, which then takes the /z/ suffix allomorph: *wife/wives.*

[3] The examples are represented using IPA characters, since the standard orthography does not make the distinctions in a consistent way.

[4] If the allomorph environments were stated intensionally in terms of phonological features, rather than extensionally as lists of phonemes, the first and second allomorphs of this affix would be need to be extrinsically ordered with respect to each other as well.

pends only on the phonemes to its left, and vice versa for a prefix; this has, in fact, been claimed to be a universal ((Bobaljik, 2000)). If this were always the case, we could simply apply the algorithm in such a way as to convert affixes to allomorphs from the inside out.[5] Nevertheless, there are at least alleged cases of outward phonological sensitivity (e.g. (Deal and Wolf, to appear)), so it is not possible to rule out a priori a language in which adjacent affixes interact in this way–and we would prefer not to create an implementation which only works if a particular linguistic theory is correct (the point dubbed theory agnosticism above).

Second, there are certainly cases of phonological processes which operate in an outwardly sensitive manner. For example, assimilation of nasal consonsants to the point of articulation of a following consonant is frequent (cf. (Bonet and Harbour, 2012, footnote 22)). While these cases can often be treated by phonological rules which change one phoneme to another (as opposed to realizing an abstract morpheme as one or another of its allomorphs), such an analysis may not be obviously correct to a linguist: there may for instance be apparent counter-examples,[6] or for one reason or another the linguist may choose not to implement the rule-based analysis. This comes back to the point made in section 2 that we wish to allow the linguist to describe a language as they prefer to analyze it, rather than imposing a theoretical straightjacket based on what is easy to implement.

Thirdly, there are cases in which an affix and a stem interact, both selecting allomorphs of the other. Many Austronesian languages have a prefix which ends in a nasal consonant when followed by a vowel-initial stem. When followed by a consonant-initial stem, the final nasal consonant of the prefix and the initial consonant of the stem may coalesce into a single nasal consonant with the stem consonants point of articulation ((Pater, 1999)). Since both the prefix and the stem are altered, if this is to be expressed by allomorphs (rather than by means of a phonological process), both the prefix and the stem must have mutually conditioning allomorphs. If a similar situation occurs between affixes, then it would not be statable under algorithm 1.

Finally, even if mutual conditioning never occursand it is certainly statistically infrequentthe fact that phonological outward sensitivity (if not necessarily mutual phonological conditioning) sometimes occurs means that a converter from a linguistically based description into sfst code would need to recognize such sensitivity and construct an order of application of the realizational rules that would ensure that for any outwardly sensitive affixes, the allomorphs of affixes appearing immediately outside of the outwardly sensitive affix were chosen before those of the outwardly sensitive affix, reversing the normal inside-to-outside order of the algorithm. While this is possible, it adds considerable complexity to the algorithm. As we show below, it is possible to allow for both inward and outward phonological conditioning at the same time, rendering unnecessary the added complexity of ordering the allomorph constraint checking based on whether the sensitivity is inward or outward.

In sum, while it will usually be possible to construct a grammar in which the order of allomorph selection avoids the need for simultaneous mutual conditioning of two affixes, this adds an unnecessary complexity to the converter, and in some cases this may be impossible. Since our converter must work for all grammars that someone might write, the converter must be able to handle this situation, and therefore algorithm 1 is not sufficient for our purposes.

I note in passing that another way to handle affix allomorphy would be to select the correct allomorph at the point at which the affix is being attached to the base, avoiding the intermediate stage in which the affix is represented by a label on the surface side. Unfortunately, this suffers from an even worse version of the problem that algorithm 1 suffered from: there is no way to handle outward phonological sensitivity in the choice of allomorphs. We must therefore look for another solution.

4.2 Implementation 2: Allomorphy by logic

Our working implementation encodes the allomorph constraints by logic, as it were. Recall that in the ordered list of allomorphs, the first allomorph which can apply to a particular wordform is allowed to apply; any remaining allomorphs cannot appear in that environment. Thus, the first allomorph must appear in environment 1; the second allomorph must appear in environment 2, but must not appear in environment 1, even if environment 1 is a subset of environment 2; and so forth, with the final allomorph typically being an elsewhere case, which is allowed to appear in any wordform so long as the allomorph is not in environment 1, 2, 3....

An algorithm for this, which overcomes the problems of algorithm 1, is given in 2.[7] The sfst code

[5] This assumes, as seems likely, that there is never phonological conditioning between prefixes and suffixes.

[6] The English derivational prefix in has allomorphs im before bilabials, il before /l/, and ir before /r/ (as well as /i/ before velars, although this is not written in the orthography). The derivational prefix un, however, has no analogous allomorphs. A phonological rule that assimilated /n/ to the following consonant would therefore be difficult to write so as to handle both affixes. While this is a case of inward sensitivity, the analogous situation with outward sensitivity cannot be ruled out.

[7] The documentation for xfst and for sfst differ in

generated by this algorithm is straightforward, but in our experience parts of it can be difficult to get right by handwhich of course is our motivation for providing a linguistically based formalism that is translated automatically into the necessary sfst code, freeing up the user to think about the linguistics rather than the somewhat complex sfst code.

As mentioned above, the algorithm has been implemented in Python, and converts the XML-based representation to sfst. When we speak of loops, therefore, we are talking about how the Python code loops over a set of affixes or allomorphs; the sfst code of course does not contain loops. To avoid confusion, we identify the pseudo-code which would be output as sfst statements by enclosing that pseudo-code in a Python function `SFSTOutput`().

```
Initialization: Initialize Lexicon to contain
each wordform consisting of a stem plus all
Allomorphs of all Slots, according to the
morphotactics of the language. Include a label
before and after each Allomorph with the
gloss of its Affix.
Then:
foreach Affix in the set of Slots of each
part of speech do
    foreach Allomorph in the ordered list of
    Allomorphs of Affix do
1           SFSTOutput(Set TemporaryLexicon
            equal to the subset of Lexicon that
            contains Allomorph.)
2           SFSTOutput(Remove from Lexicon all
            instances of Allomorph).
3           SFSTOutput(Remove from Lexicon all
            instances of Affix which appear in the
            environment specified by Allomorph).
4           SFSTOutput(Set TemporaryLexicon
            equal to the subset of
            TemporaryLexicon where Allomorph
            is found in the environment it specifies,
            and then remove all labels of Affix
            from the surface side of
            TemporaryLexicon).
5           SFSTOutput(Add TemporaryLexicon
            back to Lexicon).
    end
end
return SFSTOutput(Lexicon)
```

Algorithm 2: Logic implementation

The initialization is straightforward. At the end of the initialization, the *Lexicon* will contain wordforms of the following type (using the sfst notation):

```
(bot|kls|tri)
   <PL> ({<>}:{\-z}
        |{<>}:{\-s}
        |{<>}:{\-z})
   <>:<PL>
```

We will use this toy lexicon to illustrate the functioning of the algorithm.

The `<PL>` symbol bracketing the allomorphs on the surface side ensures that the algorithm applies allomorph constraints of a given affix only to the allomorphs of that affix, and not to sequences of phonemes which happen to be identical to the allomorph but which belong to another affix, or to a stem. In part this could be accomplished by ensuring that boundary markers separate each affix from other morphemes; this prevents an affix allomorph from being confused with part of a stem, for example.

However, some linguists may prefer not to use boundary markers, which have a checkered history in theoretical phonology. Furthermore, the use of boundary markers does not prevent the allomorph of one affix from being confusable with the allomorph of another affix, or even with a small stem. This potential confusion will cause problems if the phonological conditioning of homophonous allomorphs differs for different affixes. While this may be unusual across languages, it cannot be discounted;[8] and it can certainly happen if two affixes each have zero (null) allomorphs, since those zeroes would be homophonous. We therefore include on the surface side the label (`<PL>`, to distinguish this allomorph from allomorphs of other affixes.

Since this example has only a single affix `<PL>`, the algorithm makes only a single pass through the outer loop.[9]

The first pass through the inner loop is for the allomorph -z. Step 1 uses the following sfst code to copy all paths containing this allomorph into the temporary variable:[10]

```
$TemporaryLexicon$ =
    $Lexicon$
```

their use of the terms upper and lower to designate sides of transducers. To avoid confusion, I adopt the terms lexical and surface.

[8] In English, the possessive clitic (written as apostrophe-s) has a slightly different distribution of allomorphs from the plural noun suffix: the clitic often has a null allomorph when it follows a word-final /s/, particularly if the final syllable of the word is unstressed: Jesus disciples.

[9] A reviewer asked whether the algorithm is capable of handling unbounded affixation. The short answer is no; but then I am not aware of any language with unbounded affixation. That is, agglutinating (and even polysynthetic) languages generally have a limited number of derivational affixes, and a fixed number of slots for inflectional affixes.

[10] I have made simplifications for expository purposes, e.g. word boundary markers are not shown.

```
        || (.*<PL>\-z<PL>.*)
```

Step 2, which removes all paths containing instances of this same allomorph from `$Lexicon$`, looks like this:

```
$Lexicon$ =
        $Lexicon$
        || (!(.*<PL>\-z<PL>.*))
```

At this point, `$TemporaryLexicon$` includes all paths containing instances of the z allomorph, and `$Lexicon$` includes no instances of that allomorph.

Step 3 then removes all instances of *any* allomorph of the `<PL>` suffix from the lexicon, provided those instances appear in the environment where z is expected:[11]

```
$Lexicon$ =
        $Lexicon$
        || (!(.*(s|z||||)<PL>.*<PL>.*))
```

Of course steps 2 and 3 could be combined into a single sfst command.

At this stage, `$TemporaryLexicon$` contains all instances of the z allomorph. Step 4 first removes from `$TemporaryLexicon$` any instances of the z allomorph which do not appear in the correct envrionment,[12] and then removes the `<PL>` tags from the surface side:[13]

```
$TemporaryLexicon$ =
      $TemporaryLexicon$
      || (.*(s|z||||)<PL>\-z<PL>.*))
      || (<PL>:<> ^-> <> __)
```

Finally, `$TemporaryLexicon$` is added back to `$Lexicon$`. `$Lexicon$` now contains instances of the z only where this allomorph immediately follows strident consonants (s, z, , , and); furthermore, these instances are no longer tagged as `<PL>`. `$Lexicon$` also contains paths containing the s and z allomorphs, still tagged with the `<PL>` marker, but neither of these two allomorphs now appears after a strident consonant.

In the second pass through the inner loop, steps 1 and 2 use sfst code which is virtually identical to that of the first pass, save that the allomorph s appears in place of z.

The code in step 3 for removing all instances of `Affix` tagged as `<PL>` in the environment belonging to this second allomorph, shown below, removes instances of the -z allomorph appearing in this environment, but has no effect on the instances of the -z allomorph, since the tags bracketing that allomorph were removed in step 4 in the previous pass.

```
$Lexicon$ =
        $Lexicon$
        || (!(.*(p|t|k)<PL>.*<PL>.*))
```

The code for step 4 is also similar to that in the first pass, but this time applies to the s allomorph and its environment, ensuring that this allomorph is found only in its specified environment; it then removes the `<PL>` tags from the surface side:

```
$TemporaryLexicon$ =
      $TemporaryLexicon$
      || (.*(p|t|k)<PL>\-s<PL>.*))
      || (<PL>:<> ^-> <> __)
```

Again, the contents of `$TemporaryLexicon$` are added back to `$Lexicon$`, which will now contain all paths for which the z and s allomorphs are found in their correct environments (but without the `<PL>` tags on the surface side), as well as paths for the z allomorph (still bracketed by `<PL>` on the surface side) where this allomorph is not found in the environment for the z or s allomorphs.

Finally, on the third pass through the inner loop, steps 1 and 2 again use sfst code which is similar to that of the previous passes, except that the allomorph z appears. Since this is the last allomorph, after step 2 there will be no instances in `$Lexicon$` of this affix tagged by `<PL>` on the surface; and step 3, which removes all instances of the affix with those tags from `$Lexicon$`, is therefore unnecessary (although performing it will do no harm).

Step 4 is supposed to restrict the appearance of the allomorph in `$TemporaryLexicon$` to those paths in which it appears in the appropriate environment. Since this final allomorph is the elsewhere case, the environment is effectively anything, and this step therefore makes no change to `$TemporaryLexicon$`, except to remove the `<PL>` tags from the surface side:

```
$TemporaryLexicon$ =
      $TemporaryLexicon$
      || (.*<PL>\-z<PL>.*))
      || (<PL>:<> ^-> <> __)
```

Again, we add `$TemporaryLexicon$` back to `$Lexicon$`, which will now contain all paths for the three allomorphs, each in all and only its correct environments, and without the `<PL>` tags on the surface.

If there were more affixes, their allomorphs would be constrained to the correct environments during subsequent passes through the outer loop.

In order for this solution to work in agglutinating languages, where there may be more than one

[11] The sfst code shown here assumes that there can be no more than one instance of an affix tagged `<PL>` in any wordform. The converter warns the user if this assumption does not hold.

[12] Since the only instances of the `<PL>` affix contained in `$TemporaryLexicon$` are of the z allomorph, \-z could be replaced by .* in this sfst code.

[13] In sfst, replace rules require an explicit environment, hence the rule requires a regular expression to the left or right of the environment bar; an epsilon, written ¡¿, suffices.

order of prefixes or suffixes, one additional refinement is needed. The sfst code in steps 3 and 4 must overlook any affix labels. For example, suppose we had a hypothetical agglutinating language like this toy English example, but with an additonal order of suffixes between the stem and the plural suffix, containing affixes marking the person of the possessor. Before the application of step 1, the surface side of words in this language might look like `bot<1>{d}<1><PL>{z}<PL>`, where `<1>` is the tag used for one of the first order affixes (intended here to represent a first person possessor). In order for the sfst code in step 1 to see that the phoneme to the left of the `<PL>` suffix is the /d/ of the inner suffix, it must be possible for it to overlook the intervening affix label `<1>`.[14] Fortunately, sfst provides a construct which allows for ignoring specific characters, the insertion operator, written `<<`.[15] The refined code needed in step 4 (using the second pass through the loop as an example) for in this hypothetical agglutinating language would therefore be:

```
$TemporaryLexicon$ =
    $TemporaryLexicon$
    || (.*((p|t|k)<< <1>)<PL>\-s<PL>.*))
    || (<PL>:<> ^-> <> __)
```

Note that all the labels for affixes which have not yet been processed by the inner loop of the algorithm must be ignored. The code to ignore these affix labels is added by our converter in the relevant places.

5 Conclusion

We have described a treatment of allomorphs that accounts for the fact that they are ordered, with the first allomorph whose phonological environment is satisfied in any particular word taking priority over the remaining allomorphs. This treatment has been implemented in our converter from XML to sfst. This enables linguists to think in terms of concepts they are familiar with, without worrying about how to reliably encode them in sfst.

While one might refer to this XML-based description language as ”syntactic sugar,” we have found it to be an essential nutrient for grammar writing. The reason is that some constructs–such as the allomorph constraints discussed in this paper–are very hard to get right in practice. By allowing the user to think of them in much simpler terms–and in particular, in linguistic terms–we have made it easier for the linguists to get their grammars right, or at least to think of linguistic problems, rather than programming language problems.

In fact, what we have constructed is a linguistically based higher order programming language; and the converter from our XML representation is analogous to a compiler for a programming language, except that instead of outputting assembly language code, it outputs sfst code. Another similarity between the converter and many programming language compilers is the need for some degree of optimization in the output code. We have found that long sfst commands tend to make the sfst-compiler program very slow. Accordingly, the converter breaks up long commands into shorter ones by assigning the output of intermediate steps to variables. For example, the left and right environments of phonological rules and allomorph constraints are compiled separately into FSAs and stored as intermediate variables, before the entire phonological rule or allomorph constraint is compiled and applied.[16] Again, the fact that the optimization is done by the converter frees up the linguist from having to deal with this.

In addition to the converter from XML to sfst, we are developing other tools to make it easier for linguists to build morphological transducers. One of these tools is a debugger, which in its present form displays the derivation from underlying to surface form, including the application of phonological rules. The debugger does not yet display the steps described in this paper for the application of the allomorphs constraints; that is future work.

References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology.* University of Chicago Press, Chicago.

Jonathan David Bobaljik. 2000. The ins and outs of contextual allomorphy. In *University of Maryland Working Papers in Linguistics*, volume 10, page 3571. University of Maryland.

Eullia Bonet and Daniel Harbour. 2012. Contextual allomorphy. In Jochen Trommer, editor, *The Morphology and Phonology of Exponence*, volume 41 of *Oxford Studies in Theoretical Linguistics.* Oxford University Press.

Andrew Carstairs-McCarthy. 1987. *Allomorphy in Inflexion.* Croom Helm.

[14] If suffixes are processed left-to-right, the ¡1¿ labels on the surface side could have been removed prior to this processing step. However, this problem would still arise for phonological conditioning by suffixes to the right of the plural suffix: another instance of the outward sensitivity problem.

[15] This same mechanism is used in our system to allow morpheme boundary markers to be ignored.

[16] Differently from programming language compiler optimization, the optimization performed by our converter is intended to speed up the subsequent sfst compilation step, not the run-time performance.

Andrew Damerell Carstairs. 1987. *Allomorphy in inflexion*. Croom Helm linguistics series. Croom Helm, London.

Anne David and Michael Maxwell. 2008. Joint grammar development by linguists and computer scientists. In *IJCNLP* (DBL, 2008), pages 27–34.

2008. *Third International Joint Conference on Natural Language Processing, IJCNLP 2008, Hyderabad, India, January 7-12, 2008*. The Association for Computer Linguistics.

Amy Rose Deal and Matthew Wolf, (to appear). *Outwards-sensitive phonologically-conditioned allomorphy in Nez Perce*. Oxford University Press.

R. M. W. Dixon. 2009a. *Basic Linguistic Theory*, volume 2: Grammatical Topics. Oxford University Press.

R.M.W. Dixon. 2009b. *Basic Linguistic Theory*, volume 1: Methodology. Oxford University Press.

Mans Hulden. 2009. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, page 2932. Association for Computational Linguistics.

Sharon Inkelas and Cheryl Zoll. 2005. *Reduplication: Doubling in Morphology*. Cambridge Studies in Linguistics. Cambridge University Press.

Paul Kiparsky. 1973. elsewhere in phonology. In Stephen R. Anderson, editor, *A Festschrift for Morris Halle*, pages 93–106. Holt, New York.

Paul Kiparsky. 1996. Allomorphy or morphophonology? In Rajendra Singh, editor, *Trubetzkoy's Orphan: Proceedings of the Montreal Roundtable Morphonology: Contemporary Responses*, pages 13–31. Benjamins, Amsterdam.

Michael Maxwell and Anne David. 2008. Interoperable grammars. In Jonathan Webster, Nancy Ide, and Alex Chengyu Fang, editors, *First International Conference on Global Interoperability for Language Resources (ICGL 2008)*, pages 155–162, Hong Kong.

Michael Maxwell. 2010. Standardization as a means to sustainability. In *Workshop on Language Resources: From Storyboard to Sustainability and LR Lifecycle Management*, pages 30–33. LREC 2010.

Michael Maxwell. 2012. Electronic grammars and reproducible research. In Sebastian Nordoff and Karl-Ludwig G. Poggeman, editors, *Electronic Grammaticography*, pages 207–235. University of Hawaii Press.

Michael Maxwell. (accepted). Grammar debugging. Workshop on Systems and Frameworks for Computational Morphology.

Mary E. Paster. 2006. *Phonological Conditions on Affixation*. Ph.D. thesis, University of California, Berkeley.

Mary E. Paster. 2009. Explaining phonological conditions on affixation: Evidence from suppletive allomorphy and affix ordering 1. 2(1):18–37.

Joe Pater. 1999. Austronesian nasal substitution and other NC effects. In Ren Kager, Harry van der Hulst, and Wim Zonneveld, editors, *The Prosody Morphology Interface*, pages 310–343. Cambridge University Press.

Helmut Schmid. 2005. A programming language for finite state transducers. In Anssi Yli-Jyr, Lauri Karttunen, and Juhani Karhumki, editors, *Proceedings of the 5th International Workshop on Finite State Methods in Natural Language Processing (FSMNLP 2005)*.