Chapter 5

Planning and Managing Finite-State Projects

Writing a finite-state system such as a morphological analyzer demands the same planning and discipline as for any large software-development project. Every experienced software developer can tell horror stories about inadequate hardware, code being lost and corrupted, and the nightmare of maintaining poorly designed and undocumented systems.

From the software-engineering point of view, the finite-state developer faces a bewildering array of choices in how the project is divided into modules and how the work should be divided up among **lexc**, **xfst**, and perhaps even **twolc**.[1] From the strategic point of view, it may appear that choices must be made concerning orthography, dialect, register and even political correctness; but there are techniques that allow a single flexible core network to be modified via finite-state operations to serve multiple ends. Finally, the expert use of finite-state operations like composition, union and priority union can simplify grammars and allow cleaner handling of grammatical irregularity.

This chapter will discuss hardware and software infrastructure, the kinds of software and linguistic planning that should be done *before* any serious coding, and some expert idioms that keep your grammatical descriptions simpler while at the same time making them maximally flexible.

5.1 Infrastructure

5.1.1 Operating System

The **Xerox** Finite-State Tools licensed with this book are compiled for the Solaris, Linux (Intel-based) , Windows (NT, 2000, Me, XP) and Macintosh OS X operating systems.

[1]The **twolc** language and compiler are documented on the website `http://www.fsmbook.com/`.

and dangerous editing. In general, think of version control as a way to save software versions that you might want to return to.

- When you are testing a new stable version of your overall system, you should always compare it against the previous stable version of the same system. If you faithfully took a snapshot of the previous version, CVS can restore it for your tests.

- If you have delivered a version of your project to any other organization or development team, they may come back to you weeks or months later with bug reports and requests for changes. You may need to restore a copy of the version that was delivered in order to confirm the bugs.

Experienced software developers use and appreciate version control, even if it does impose some overhead in checking out and checking in files.

Makefiles and System Charts

Typical lexical transducers are made from multiple **lexc** and **xfst** source files, and the resulting networks are unioned, composed and intersected in various ways to make the final network. While recompiling and re-assembling a whole system can theoretically be done by hand, in practice the only way to ensure reliability is to put all the commands in a MAKEFILE.

Makefiles Makefiles are organized in terms of goals; in our case the top-level goal will be the production of the final lexical transducer or transducers. A makefile typically contains many subgoals that must be produced on the way to the final goal, including the appropriate compilation of all the component files and calculation of intermediate results. In a properly written makefile, each goal is associated with all of the subgoals or source files on which it depends, and the operating system's `make` utility is intelligent enough to recompute a goal only if one of its subgoals or source files has changed. After you edit and save any source file of your system, you should be able simply to enter the system command `make` to have all (and only) the affected parts of your overall system recompiled. See the website[6] for some examples of makefiles for finite-state systems.

System Charts While a properly written makefile will automate the recompilation and re-assembly of your lexical transducers, experience has shown that finite-state developers all too easily lose track of the overall organization of their system modules unless they are visualized graphically. This problem is particularly acute if the developer can't write, or perhaps even read, the makefile himself or herself. It is therefore highly recommended that the components and operations of the makefile be plotted graphically as boxes on a chart and that the chart be taped prominently

[6] `http://www.fsmbook.com/`

to the wall for easy reference during development. Such a chart should be as large as necessary, showing explicitly how the system modules are compiled, unioned, intersected and composed together.

The wall chart is especially valuable when tracing bugs, and, of course, when understanding and modifying the makefile itself. It is up to the developer to keep the makefile and the wall chart in sync.

5.2 Linguistic Planning

lexc and **xfst** are tools that allow the developer to state formal linguistic facts in notations that are already familiar to formal linguists. Thus **lexc** gives us a framework for specifying morphotactic facts in terms of sublexicons, entries and continuation classes; and **xfst** rewrite rules provide a way to specify facts about phonological or orthographical alternations. But these tools are only frameworks for stating facts; they cannot tell you what the facts are. The discipline that examines natural languages, discovering their rules and formalizing them, is FORMAL LINGUISTICS.

5.2.1 Formal Linguistics

When we refer to *linguists* in this book, we mean *formal* linguists, typically people who have been trained at the university level in formal linguistic description. In common parlance, a linguist is anyone who speaks multiple languages, i.e. a polyglot, which is something quite different. The ability to speak multiple languages, though admirable, doesn't make one a formal linguist any more than having a heart makes one a cardiologist.

A great deal of good formal linguistics was done before computerized tools were available, and a great deal of bad formal linguistics can be done even with the best computational tools. It is important to realize that coding in **lexc** and **xfst** is not linguistics per se and that the formalization of bad linguistic models simply results in bad programs. Formal linguistics is what it always has been, the formal study and understanding of linguistic structures; and that study and understanding should come first. After the formal linguistics is done, after a model is worked out, we implement and test the model using the finite-state tools.

The lesson is this: study the language and do some old-fashioned pure linguistic modeling before jumping into the coding. Your programs will never be better than the linguistic model behind them.

5.2.2 Programming Decisions

Experienced finite-state developers understand that most programming projects can be divided up in many ways, and this is especially the case when one has a choice of multiple tools, **lexc** and **xfst**,[7] whose capabilities overlap. A little planning, in

[7] Alternation rules can also be written in the **twolc** format, but **xfst** replace rules have largely supplanted them at **Xerox**.

the form of software engineering, is always recommended.

Tool Choice

lexc and **xfst** are notations that compile into finite-state networks, so a developer must often choose which tool to use to define a particular network. **lexc** is convenient for defining natural-language lexicons and morphotactics; it is optimized to handle huge lexical unions efficiently, and the continuation classes provide a convenient notation for encoding general compounding and looping. **lexc** also provides convenient notations for handling irregularities, including the gross irregularities called SUPPLETIONS. Anything that can be encoded in **lexc** can theoretically be encoded in **xfst**, using the regular-expression compiler, but with some practical limitations:

1. **xfst** is *not* optimized, as **lexc** is, to handle huge unions efficiently, and
2. regular expressions that encode general compounding and looping can be very hard to write and to read

For these reasons, **lexc** is the preferred choice for defining lexicons and morphotactics.

Alternation rules, which also compile into networks, can be written in either **twolc** or **xfst**. At **Xerox**, **twolc** has largely fallen out of use, with most developers preferring to write cascades of replace rules in **xfst**.

Modularity

Depending on your language, you may want to keep verbs, nouns, adjectives, etc. in separate **lexc** lexicons and compile them into separate networks. The corresponding verb, noun, adjective, and other networks may subsequently be unioned together before alternation rules are composed, or you may need to write and apply at least some rules that are specific to particular categories. For languages with productive compounding, where adjective, noun and verb roots attach to each other, all the compounding categories will have to be kept together in the same **lexc** source file.

For alternation rules, you may want to organize your rules into several layers, creating various intermediate languages between the lexical and surface levels. You will almost certainly want to apply little cleanup transducers to the lexical side, and perhaps to the surface side, of your final lexical transducer. It is impossible to lay down rigid rules, because languages differ so much, but some decent advance planning can save much rewriting and reorganization of the system during development.

Whenever possible, formal linguistic studies and descriptions should be presented to other linguists for comment before you jump into the coding. Part of the plan should include a wall chart (see page 282), a graphic display of the various components of the system and how they will be combined together.

Choosing Lexical Baseforms

By convention, **Xerox** lexical transducers have lexical (upper-side) strings that consist of BASEFORMS and multicharacter-symbol tags. The baseform itself is the headword used conventionally when looking up the surface words in a standard printed (or perhaps online) dictionary. For example, a Spanish verb lemma consists of perhaps 300 different surface forms, but only the infinitive form, the conventional baseform, appears as a headword in dictionaries. Thus *canto* ("I sing") is analyzed in the **Xerox** Spanish morphological analyzer as a form of *cantar* ("to sing").

```
Upper:    cantar+Verb+PresInd+1P+Sg
Lower:    canto
```

There is nothing necessary or obvious about using the infinitive to represent the whole lemma; it is only a lexicographical convention of Spanish, and the conventions change from language to language. Latin, for example, has a cognate infinitive *cantare*, but the headword in standard dictionaries is *canto*, the first-person singular present-indicative active form. In a typical **Xerox** Latin analyzer, we would therefore expect *cantare* and all other surface forms of the lemma to be analyzed as forms of *canto*.

```
Upper:    canto+Verb+Inf
Lower:    cantare
```

In analyzing other languages, where lexicographical conventions are mixed or non-existent, it is ultimately up to the linguist to decide what the baseform will be. Some dictionaries, such as those for Esperanto and Sanskrit, follow the admirable convention of using stems rather than whole surface words as the keys in dictionaries. Dictionaries for Semitic languages are usually organized under very abstract root keys, which are typically triplets of consonants that aren't even pronounceable by themselves.

The overall goal is to choose baseforms that will be of maximum familiarity and usefulness in future projects. Where a lexical tradition already exists, writing a **Xerox**-style system that returns the conventional keywords will facilitate looking up definitions and translations in existing lexicons. For a language that has no good tradition for choosing baseforms, it is up to you to choose them. In any case, you should have a solid view of what the baseforms will be before you start coding with tools like **lexc** and **xfst**.

Multicharacter-Symbol Tags and Tag Orders

Choosing Tags See the website[8] for guidelines on choosing appropriate analysis tags for your language. Pure linguistic planning should clarify which distinctions

[8] `http://www.fsmbook.com`

are real in the morphology and, therefore, which tags will be necessary to represent those distinctions. As much as possible, the choice of a preliminary set of tags should be done before coding starts.

Tag String Orders Where multiple tags follow the baseform, the linguist must define the order in which tags will appear. Failure to define standard combinations and orders of tags will make it impossible for anyone to use your system for generation. Tag orders should be documented in a Lexical Grammar (see Section 6.4.1).

The Lexical Language is Defined by the Linguist

A transducer encodes a relation between an upper-side language and a lower-side language, where a language is a set of strings. When defining a transducer that performs morphological analysis for a natural language like French, the surface language is usually defined for the linguist by the official orthography. The upper-side or lexical language, however, must be defined by the linguist according to his or her needs and tastes.

The **Xerox** convention of defining lexical strings that consist of a baseform followed by tags may not be suitable in all cases, and indeed this convention is not always followed even at **Xerox**. Here are some cases that suggest other ways to design the lexical language:

1. If the language makes productive use of prefixation rather than suffixation, then it may be much more practical to define a lexical language where the tags are prefixed to the baseform.

2. If the language is Semitic, involving surface stems that are complex interdigitated constructs, then it may be convenient to define a lexical language that separates and labels the component root and pattern (or root, template and vocalization) on the lexical side, depending on your theory.

3. If the goal of the system is pedagogical, then it may be convenient to define a lexical language that contains the (morpho)phonological content of the affixes, not just of the baseform.

There are no hard-and-fast rules for the design of a lexical language. Expert developers understand that the lexical language is defined by the developer, and they put considerable planning into making it as useful as possible for the application or applications at hand.

5.2.3 Planning vs. Discovery

Plan Before You Code is always a good maxim, but no one can plan everything, and there comes a point where one must simply sit down and start writing **lexc** and

xfst. The rigor of formalizing your models in these programming languages will inevitably highlight possibilities and gaps that you didn't even imagine. You will find your own initial intuitions to be unreliable; and even the printed descriptions of your language will soon prove to be inaccurate and incomplete, intended as informal guidance to thinking humans rather than formal descriptions for a computer program. This is never a surprise to formal linguists, but it can come as a shock to those trained only in traditional schoolbook grammar.

Finite-state morphological analyzers can typically process thousands of words per second, analyzing huge corpora while you eat lunch, and quickly revealing the errors and omissions of your grammars and dictionaries. Building and testing a morphological analyzer can therefore be an important part of the linguistic investigation itself.

As problems arise, be prepared to refine your linguistic theories and rewrite parts of your system. Linguistic development is an endless round of observation, theorizing, formalizing and testing; and the goal, for a lexical transducer, is to create a system that correctly analyzes and generates a language that looks as much like the real natural language as possible.

> The ultimate goals of any morphological analyzer are to accept and correctly analyze all valid words, and not to accept any invalid words.

5.2.4 Planning for Flexibility

First-time users of the **Xerox** Finite-State Tools usually have a single final application in mind, such as a spelling-checker, part-of-speech disambiguator, parser or even an automatic machine-translation system, and they tend to make design choices that limit the potential uses of the system. Experienced developers understand that a single lexical transducer can and should be the basis for multiple future applications, and they build this flexibility into the rules and lexicons at many levels. A sound understanding of the full power of the finite-state calculus, and especially the union and composition algorithms, is the prerequisite to planning and maintaining flexibility.

Thinking about Multiple Final Applications

The first task of finite-state developers is usually to create a morphological analyzer, i.e. a Lexical Transducer, and such systems are a key component of many other applications. For example, the extracted lower side of a lexical transducer can serve as a spelling checker, and morphological analysis is a vital step before part-of-speech tagging, parsing, and all the various NLP (Natural-Language Processing) systems that depend on parsing. **Xerox** tokenizers and HMM (Hidden

Markov Model) taggers use a modified lexical transducer as a vital component.

Even within a particular application, such as a spelling checker, one might imagine versions that accept vulgar words, and others that do not. Some morphological analyzers might accept and analyze words from Brazilian Portuguese and Continental Portuguese, and others might specialize in one or the other. In some languages, spelling reforms are adopted from time to time, changing the surface language that needs to be analyzed and generated. The keys to flexibility are the following:

- Avoid making decisions that limit flexibility. Instead of choosing Option A or Option B, try to create a core system that supports both.
- Plant the seeds of flexibility, typically in the form of feature markings, in your source files.
- Try to maintain a single set of source files, especially dictionary files.
- Use the finite-state calculus, and particularly union and composition, to generate multiple network variations from the single set of source files.

One Core, Many Modifications

In supporting flexibility, the main thing you want to avoid is maintaining multiple copies of core files such as lexicons and alternation rules. You do not, for example, want to maintain separate dictionaries for American English and British English that differ only in small details. It is humanly impossible to maintain multiple large files in parallel.

Dialects Suppose, for example, that you are building a Portuguese system and that there are two major dialects: Brazilian and Continental (i.e. Portuguese as spoken and written in Portugal). While most written words in the language are common to the two dialects, there are some idiosyncratic and even some systematic differences. For example, the man's name *Antônio* in Brazil is spelled *António*, reflecting a significantly different pronunciation of the accented vowel, in Portugal. The two verb forms spelled *cantamos* and *cantámos* in Portugal are both spelled *cantamos* in Brazil. In addition, the rather productive diminutive suffix *-ito/-ita* used in Portugal is found only in a few fossilized examples, such as *cabrito* ("kid" or "young goat") in Brazil, where the productive diminutive suffix is *-inho/-inha*. (The *-inho* and *-inha* diminutives are also used in Portugal.) Finally, there are some gross terminological differences such as *papa-formigas* ("ant-eater") in Portugal vs. *tamanduá*, borrowed from the Tupi language, in Brazil.

In such cases, one should not choose one dialect or the other, and one should definitely not maintain two separate lexicons. Instead, the proper finite-state approach is to go into the common source lexicon and include a distinctive feature

marking, e.g. a multicharacter-symbol tag like ^C, on the lexical side of all morphemes that are exclusively used in Continental Portuguese. A parallel feature mark, like ^B, can be placed on the lexical side of all morphemes that are exclusively used in Brazilian Portuguese.

```
! Marking dialectal vocabulary with multicharacter
! 'features'

Multichar_Symbols ^C ^B

LEXICON Root
        Nouns ;

LEXICON Nouns
gato        Nm ;    ! common word for 'cat'
cachorro    Nm ;    ! common word for 'dog'

papa-formigas   NmC ;   ! Continental for 'ant-eater'
tamanduá        NmB ;   ! Brazilian for 'ant-eater'

eléctrico   NmC ;   ! Continental for 'streetcar'
bonde       NmB ;   ! Brazilian for 'streetcar'

LEXICON NmC
^C:0    Nm ;

LEXICON NmB
^B:0    Nm ;
```

```
[] <- [ %^C | %^B ]
.o.
CommonCoreSystem
```

Figure 5.1: Leave Both Brazilian and Continental Words

A non-final Common Core version of the lexical transducer will then have a majority of lexical strings containing no ^B or ^C feature, some strings marked with ^B, and some with ^C. To create a version of the system that accepts both Brazilian and Continental words, all one needs to do is to start with the Common Core network and map both ^C and ^B upward to the empty string as shown in Figure 5.1.

Similarly, to create a purely Brazilian system that contains no exclusively Continental words, we map ^B to the empty string and ^C to the null language (which

eliminates all the Continental paths) as in Figure 5.2.[9]

```
~$[] <- %^C
.o.
[]  <- %^B
.o.
CommonCoreSystem
```

Figure 5.2: Keep Brazilian Words, Kill Continental Words

Finally, to create a purely Continental system that contains no exclusively Brazilian words, we do the opposite, as shown in Figure 5.3. The order of the compositions in Figures 5.2 and 5.3 is not significant for these examples.

```
[] <- %^C
.o.
~$[] <- %^B
.o.
CommonCoreSystem
```

Figure 5.3: Kill Brazilian Words, Keep Continental Words

Spelling Reforms Spelling reforms, like dialectal differences, are problems that plague all developers in natural-language processing. Beginning developers too often feel that they have to make a choice, to accept only the old orthography or only the new orthography. The better solution, of course, is to accept both, or either, in variants of the same core system. Spelling reforms are often announced and then abandoned, or new and old forms may exist side-by-side for years; in any case, large corpora of text in the old orthography may exist forever and still need to be processed. Creative flexibility is highly encouraged when dealing with spelling reforms.

The basic mechanism is the same as for dialectal differences. One feature, such as `^N` (for "new") is placed on the upper side of new word spellings, and something like `^O` (for "old") is placed on spellings destined someday to disappear. Then you use simple composition, as in the dialect examples above, to allow or delete new and old orthographical forms in variations of the final lexical transducer.

[9]The null language is that language that contains no strings at all, not even the empty string. The null language can be notated in an infinite number ways, including `~$[]`, `~[?*]`, `[a - a]`, etc.

> In any apparent choice between two desirable alternatives, always try to have it three ways: either one, or the other, or both.

Spelling differences can straddle the line between dialects and spelling reforms. In Brazil, there is a traditional distinction between the spellings *qui* representing /ki/ and *qüi* representing /kwi/; also between *que* (/ke/) and *qüe* (/kwe/), *gui* (/gi/) and *güi* (/gwi/), *gue* (/ge/) and *güe* (/gwe/). A Brazilian or student of Brazilian could therefore look at the written words *quinze* ("fifteen") and *tranqüilo* ("tranquil") and know how they are pronounced. The pronunciation of the words is the same in Portugal, but the *ü* letter is never used in the orthography, making written *quinze* and *tranquilo* phonetically ambiguous, at least to a foreigner trying to learn the language. The solution in such cases is always to adopt the spelling convention with the maximum number of distinctions for the common core lexicon, using the spelling *tranqüilo*, with the *ü*, as the common baseform. It's always easy to collapse the orthographical *ü* to plain *u* for Continental Portuguese orthography, as in Figure 5.4, but adding distinctions post hoc requires going in and editing the core lexicons.

```
u <- ü
.o.
CommonCoreSystem
.o.
ü -> u
```

Figure 5.4: Collapsing all *ü*s to *u*s for Continental Portuguese

Recently, Brazil and Portugal have been trying to make their orthographies more similar, and one element of a 1986 proposal was for Brazil to abandon the use of *ü*. This proposal has not been accepted, at least not completely, and the debate continues between the linguistic conservatives and the liberals. Many Brazilians now consider the use of *ü* optional in most common words where natives know the pronunciation. To create a version of the analyzer that handles such optional usage of *ü*, one could simply compose the optional rule `[ü (->) u]` on the lower side of the CommonCoreSystem. If *ü* were ever officially abandoned in Brazil, then the composition in Figure 5.4 would serve for the new Brazilian orthography as well. Yet we still would not want to edit the source files; we may still, for many years to come, want to be able to create versions of our system that accept only traditional or transitional Brazilian orthography.

Vulgar, Slang, Substandard Words As with dialect words, it is sometimes desirable to produce variations of a system that contain no vulgar, slang, substandard or other politically incorrect words. This may be the case, for example, in a spelling checker that returns suggested corrections. Such dangerous words should also be marked with features in the core lexicons, perhaps

```
^V      for Vulgar
^S      for Slang
^D      for Substandard/Deprecated
```

Composition can once again be used to create many customized final versions without any re-editing of the source files.

```
StrictSpanishCoreNetwork
.o.
[ á (->) a ] .o. [ Á (->) A ] .o. [ é (->) e ]
.o.
[ É (->) E ] .o. [ í (->) i ] .o. [ Í (->) I ]
.o.
[ ó (->) o ] .o. [ Ó (->) O ] .o. [ ú (->) u ]
.o.
[ Ú (->) U ] .o. [ ü (->) u ] .o. [ Ü (->) U ]
```

Figure 5.5: Allowing Degraded Spanish Spelling. These trivial rules, composed on the lower side of a strict network that analyzes properly spelled Spanish words, allow the same words to be recognized with or without proper accents on the vowels. Note that these are optional rules, written with the parenthesized right arrow. Both the strict and the relaxed versions of the system may prove useful for different applications.

Handling Degraded Spelling In some applications, such as parsing email messages or text retrieval, it is often desirable to create a version of a morphological analyzer that accepts words even when they are incorrectly accented. The trick is first to create a strict core system that has a lower-side language consisting of only properly spelled words, and then use composition to create versions that accept degraded spellings. For example, it is very common for Spanish and Portuguese email writers to dispense with accents, degrading letters like *é* to *e* and *ô* to *o*; but properly accented letters might still appear. The best way to handle such random deaccentuation is to create a permissive version of your basic strict system by composing a RELAXING GRAMMAR on the bottom of the common core system. The relaxing grammar for Spanish in Figure 5.5 optionally allows accented letters to be unaccented on the new surface level.

```
StrictGermanCoreNetwork
.o.
[ ü (->) u e ] .o. [ ö (->) o e ]
.o.
[ ä (->) a e ] .o. [ ß (->) s s ]
```

Figure 5.6: Allowing Relaxed German Spelling. These trivial rules, composed on the lower side of a strict network that analyzes properly spelled German words, allow the resulting network to accept conventional respellings.

In another useful but technically trivial example, German spelling has long tolerated the convention that *ü* can be spelled *ue*, *ö* can be spelled *oe*, *ä* can be spelled *ae*, and *ß* can be spelled *ss* if the correct letters are not available. Starting with a strict German transducer, with *ü*, *ö*, *ä* and *ß* on the lower side, the rules shown in Figure 5.6 will create a more permissive version.

Specialized Extra Words Often one customer will want special strings, such as proprietary part numbers, email addresses, internal telephone numbers or even common misspellings, added to the morphology system; but other customers may not need or want such words at all. The solution in such cases is to build a common core system of words that everyone is likely to want, and then write separate grammars of specialized words. Separate networks compiled from these separate grammars can then be selectively unioned with the strict core system to make customized versions for particular customers.

To take a concrete Spanish example, the plural word for "countries" is formally *países*, with an acute accent on the *i*, but even some otherwise careful newspapers omit the accent in this one case. If the correctly spelled word analyzes as

```
país+Noun+Masc+Pl
```

then the strict core system can be loosened up to accept this one misspelled word by performing the following trivial union.

```
StrictSpanishCoreNetwork |
[ [ {país} %+Noun %+Masc %+Pl ] .x. {paises} ]
```

Other applications, including strict spelling checkers, will need the stricter core network.

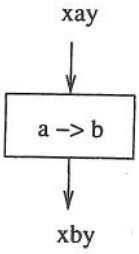
5.3 Composition is Our Friend

As is clear from the examples shown above, composition is a powerful tool for customizing lexical transducers, allowing a single core system to serve multiple

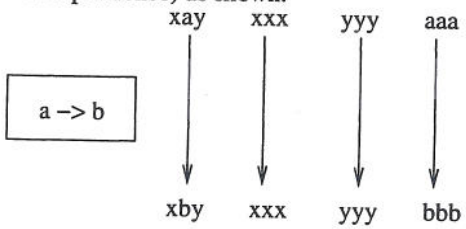
ends. In this section, we present a summary of useful things you can do with transducers and composition.

5.3.1 Modifying Strings

We usually picture transducers as little abstract machines that map one string into another string or strings. For example, from a generation point of view, consider the following mapping

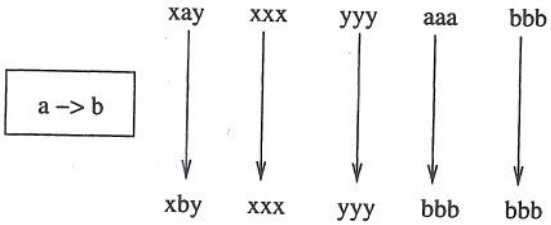


which changes "xay" (perhaps a word in our dictionary) into "xby". In such cases, we usually do not think of the rule as changing the size of the original lexical language, but just as modifying one or more strings in some way. If we have the following four strings in the lexical language, they would be mapped into surface strings (in vertical correspondence) as shown.



Note that where a rule doesn't apply, as in the cases of "xxx" and "yyy", the rule simply maps the input to the output with no change.

In some cases, by accident, rules can effectively collapse two strings in the mapping. If, for example, the lexical language also contains "bbb", we would see the following:



where the strings generated from lexical "aaa" and lexical "bbb" both appear on the surface as "bbb". The resulting surface "bbb" is therefore ambiguous, and the lower language has one fewer distinct strings than the upper language. Such ambiguities are often seen in natural language. Consider the Spanish noun *canto*

meaning "song" or "chant". It happens that *canto* is also "I sing", one of the many conjugated forms of the verb *cantar* ("to sing"). Suppose that the initial **lexc** lexicon specifies the following two string pairs

```
Upper:    canto+Noun+Masc+Sg
Lower:    canto+Noun+Masc+Sg

Upper:    cantar+V1+PresInd+1P+Sg
Lower:    cantar+V1+PresInd+1P+Sg
```

and that we define the following cascade of rules to be composed on the bottom of the lexicon.

```
[ a r -> [] || _ %+V1 ]
.o.
[ %+V1 %+PresInd %+1P %+Sg -> o ]
.o.
[ %+V1 %+PresInd %+2P %+Sg -> a s ]
.o.
[ %+V1 %+PresInd %+3P %+Sg -> a ]
.o.
...              ! many more rules
.o.
[ %+Noun %+Masc %+Sg -> [] ] ;
```

If you trace the generation by hand (or using **xfst**), you will find that both surface strings end up as "`canto`".[10]

When used in the way just described, the application (composition) of rules to the lexicon tends not to modify the size of the resulting languages very much. The assumption is that the lexicon is generating valid abstract strings, and that it is the job of the rules simply to modify them to produce valid surface strings, some of which will be ambiguous.

5.3.2 Extracting Subsets of a Transducer

Composition can be used to extract subsets of paths from a transducer or language, and this is often done during testing. Assuming we have a large transducer in the file `Esperanto.fst` that covers nouns, adjectives and verbs, and assuming that the tags `+Noun`, `+Adj` and `+Verb` mark the appropriate words on the upper side, we can read in `Esperanto.fst` and isolate just the nouns with the following regular expression:

```
xfst[0]: read regex
$[%+Noun]
.o.
@"Esperanto.fst"  ;
```

[10] Beware: you should test such bare cascades of rules using **apply down**; **apply up** may yield odd results or even segmentation faults until a restricting lexicon itself is composed on top of the rules.

The regular-expression notation `@"Esperanto.fst"` tells the **xfst regex** compiler to read in the network stored in the binary file `Esperanto.fst`. The double quotes are necessary if the filename contains punctuation characters, as in this case. The expression `$[%+Noun]` denotes the language consisting of all and only strings that contain the symbol +Noun. When composed on the top of `Esperanto.fst`, it will match all the noun strings (because they have a +Noun symbol in them) and will fail to match any of the verb and adjective strings (precisely because they lack a +Noun symbol). The result of the composition is that subset of `Esperanto.fst` that contains just noun strings; all the adjectives and verbs and anything else without a +Noun symbol on the upper side are deleted in the composition.

Conversely, we can filter out the nouns, leaving everything else, by composing the complement of `$[%+Noun]`, i.e. `~$[%+Noun]`, on top of the lexicon:

```
xfst[0]: read regex
~$[%+Noun]
.o.
@"Esperanto.fst"  ;
```

The expression `~$[%+Noun]` compiles into a network that accepts all and only strings which do not contain +Noun.

Multiple levels of regular expressions or rules can be composed on either side of a transducer as appropriate, but it is up to the developer to keep track of what kind of strings are on each side. If you have a transducer saved as `myfile.fst`, with the surface strings on the lower side, as usual, you can limit it to contain only those paths with surface strings that end in *ly* with the following composition:

```
xfst[0]: read regex
@"myfile.fst"
.o.
[ ?* l y ] ;
```

Notice that such a surface filter needs to be composed on the lower side of the lexicon transducer, where the surface strings are visible. You can limit the same grammar to cover only surface strings that begin with *re*:

```
xfst[0]: read regex
@"myfile.fst"
.o.
[ r e ?* ] ;
```

> Note that in general regular expressions, including examples such as `[ ?* l y ]` and `[ r e ?* ]`, the beginning and the end of the denoted strings are implicitly specified. `[ ?* l y ]` denotes the language of all strings that end with *ly*; `[ r e ?* ]` denotes the language of all strings that begin with *re*. The `.#.` notation is *not* appropriate here; `.#.` can appear only in restriction contexts and in replace-rule contexts (see pages 46 and 136).

You can extract all the words that, on the surface, begin with *re* and end with *ly* with the following:

```
xfst[0]: read regex
@"myfile.fst"
.o.
[ r e ?* l y ] ;
```

You can limit the system to contain only adjectives ending (on the surface) with *ly* with the following:

```
xfst[0]: read regex
$[%+Adj]
.o.
@"myfile.fst"
.o.
[ ?* l y ] ;
```

If `myfile.fst` has good coverage of English adjectives, this final exercise will isolate the paths that have adjectives like *friendly*, *comely*, *homely* and *cowardly* on the lower side.

Note that tags, by **Xerox** convention, are found on the upper side of a lexical transducer, so the restriction `$[%+Adj]` must be composed on the top of `myfile.fst`, while the *ly* requirement is a restriction on surface strings, and so that restriction must be composed on the bottom where the surface strings are. As always, it is vitally important to keep straight which side is up and which side is down, and what the strings look like at each level, when performing composition.

5.3.3 Filtering out Morphotactic Overgeneration

It is also possible, and often useful, to use composition to restrict an overgenerating lexicon. Many morphotactic restrictions found in real language are awkward to impose within **lexc** (or any other formalism for specifying finite-state networks), and in such cases it may be useful to define, on purpose, an initial lexicon that overgenerates. As a simple example, consider the Esperanto Animal Nouns, where readers were urged (see Section 6, page 219) not to worry about restricting multiple

occurrences of the feminine (+Fem) suffix *in* , the augmentative (+Aug) suffix *eg* or the diminutive (+Dim) suffix *et*.

Here's one purposely overgenerating **lexc** description:

```
Multichar_Symbols  +Noun +Aug +Dim +Fem +Sg +Pl +Acc

LEXICON Root
        Nouns ;

LEXICON Nouns
hund    N ;             ! dog
kat     N ;             ! cat
elefant N ;             ! elephant

LEXICON N
+Aug:eg         N ;     ! looping site!
+Dim:et         N ;
+Fem:in         N ;
                NSuff ; ! escape from the loop

LEXICON NSuff
+Noun:o Number ;

LEXICON Number
+Sg:0   Case ;
+Pl:j   Case ;

LEXICON Case
        # ;
+Acc:n  # ;
```

Despite the instructions, many readers do worry a lot about the fact that this grammar will happily overgenerate strings like

```
hundegego
hundininoj
elefantineginetojn
```

and they try to incorporate in the **lexc** description various restrictions that may or not reflect realities in Esperanto grammar. In any case, let us assume for the purposes of this example that the three suffixes *et*, *eg* and *in* can all co-occur in the same word, in any order, but that a maximum of only one of each can appear in a valid word. Let us also assume that we have compiled our overgenerating **lexc** grammar and stored the results in a file named `esp-lex.fst`. If you actually compile this grammar, **lexc** will inform you that it contains 18 states, 23 arcs and is CIRCULAR; this means that the grammar produces an infinite number of strings. The circularity of course results from the loop in `LEXICON N`.

Here is a step-by-step reasoning to a solution that removes the overgeneration:

- We want to restrict our lexicon to contain only those string pairs where the string on the lexical side contains zero or one appearances of +Dim, zero or one appearances of +Aug, and zero or one appearances of +Fem. We will let these three suffixes occur in any order, but we never want to see two or more of the same suffix in a single string.

- Another way to look at the problem is to realize that any lexical string that contains two or more +Aug symbols is bad. We can formalize these bad strings as

```
$[%+Aug ?* %+Aug]    ! some bad lexical strings
```

  Read this as "all strings that contain one +Aug followed at any distance by another +Aug". This is a finite-state characterization of what we do not want: having two or more +Aug symbols in the same word.

- Similarly, a lexical string that contains two or more +Dim symbols, or two or more +Fem symbols, is also bad.

```
$[%+Dim ?* %+Dim]    ! more bad lexical strings

$[%+Fem ?* %+Fem]    ! yet more bad lexical strings
```

- A single expression that matches all the bad strings is the following:

```
[ $[%+Aug ?* %+Aug]
| $[%+Dim ?* %+Dim]
| $[%+Fem ?* %+Fem] ]
```

  We want to remove every string pair whose lexical side is in this language from the overgenerating transducer created by **lexc**. We cannot subtract transducers, because subtraction is not a valid operation on transducers, but we can accomplish our goal by creative composition.

- The complement of the bad strings denotes a language consisting of good strings:

```
~[ $[%+Aug ?* %+Aug]
 | $[%+Dim ?* %+Dim]
 | $[%+Fem ?* %+Fem] ]
```

  This expression will match all the lexical strings of our transducer *except* the bad strings we want to eliminate.

- And when we compose this net *on top of* the overgenerating lexicon (remember that the tags are on the top), the composition will impose the restrictions we desire. The unmatched strings, which are bad, are simply eliminated in the composition.

```
xfst[0]: read regex
~[ $[%+Aug ?* %+Aug]
 | $[%+Dim ?* %+Dim]
 | $[%+Fem ?* %+Fem] ]
.o.
@"esp-lex.fst" ;
```

When you compile this expression, **xfst** informs you that the result contains 34 states, 48 arcs, and 192 words. As the original `esp-lex.fst` was circular, it contained an infinite number of words. By imposing the restriction, we have reduced the number of words covered and, in doing so, have increased the memory (the number of states and arcs) required to store the network. Imposing restrictions on a transducer often results, counter-intuitively, in increased memory. In the worst cases, restrictions composed on a transducer can cause it to "blow up" in size.

The use of composed filters to eliminate overgeneration is often much cleaner than trying to impose the same restrictions in **lexc**. For an alternative expert method to limit morphotactic overgeneration, while keeping networks small, see Chapter 7 on Flag Diacritics.

5.4 Priority Union for Handling Irregular Forms

5.4.1 Irregular Forms

When writing morphological analyzers for natural languages, one is often faced with a finite number of idiosyncratic irregular forms that simply do not follow the usual rules. English noun plurals provide convenient examples, and we shall see that it is sometimes convenient to add exceptional plurals to the productive plurals in a transducer, while other times it is convenient to override productive, but unacceptable, plurals with irregular ones. The adding of new paths to a transducer is easily accomplished using the union algorithm, which is already familiar to us. The overriding of paths in one transducer with paths from another is accomplished elegantly using the PRIORITY UNION algorithm, which will be illustrated below.

Looking at the data, we note that some English nouns like *money* have a normal productive plural plus one or more exceptional plurals as shown in Table 5.1. In these cases the final morphological analyzer must recognize the irregular forms as well as the regular ones. We will refer to such irregular forms as EXTRA PLURALS.

Noun	Regular	Irregular
money	moneys	monies
octopus	octopuses	octopi
bus	buses	busses
cherub	cherubs	cherubim
fish	fishes	fish

Table 5.1: Some English Nouns with Extra Irregular Plurals

Noun	Regular	Irregular
sheep	*sheeps	sheep
deer	*deers	deer
kibbutz	*kibbutzes	kibbutzim
automaton	*automatons	automata

Table 5.2: Some Irregular English Plurals that Should Override the Regular Forms

```
Multichar_Symbols +Noun +Sg +Pl

LEXICON Root
        Nouns ;

LEXICON Nouns
dog         Nreg ;       ! regular nouns, take -s plural
cat         Nreg ;
dish        Nreg ;       ! regular noun, takes -es plural
                         ! (handled by productive rule)

money       NregIes ;    ! noun takes -s plural plus -ies
octopus     NregI ;      ! noun takes -s plural plus -i
bus         NregSes ;    ! noun takes -es plural plus -ses
cherub      NregIm ;     ! noun takes -s plural plus -im
sheep       Nsame ;      ! no change in the plural
deer        Nsame ;
kibbutz     Nim ;        ! -im plural only
automaton   Na ;         ! -a plural only
fish        NregSame     ! plurals fish and fishes
```

Figure 5.7: Trying to Handle Irregular Plurals in **lexc** with Customized Continuation Classes. This approach is awkward and not generally recommended.

Some nouns like *sheep* and *deer* do not change in the plural, and other nouns like *kibbutz* have been half-borrowed into English, keeping only their native plurals, which are irregular from the English point of view. In such cases, illustrated in Table 5.2, the final morphological analyzer should not recognize the regular plurals but only the irregular ones. We will refer to these irregular forms as OVERRIDING PLURALS.

We will now examine three ways that these irregular plurals can be handled, concluding with the technique of using priority union to override undesired forms with desired forms. This new technique is especially appropriate and elegant when the exceptions are, as with these English plurals, finite and genuinely idiosyncratic.

5.4.2 Handling Irregularities in lexc and Rules

It is possible, but usually awkward, to handle irregular English plurals by encoding them specially in **lexc** and perhaps even writing extra alternation rules for them. Thus one could imagine writing a **lexc** grammar like the one in Figure 5.7, where each of the irregular nouns is given a customized continuation class. For example, the continuation class Nsame for *sheep* and *deer* is easily defined, as is the class

Nim for *kibbutz*.

```
LEXICON Nreg
+Noun+Sg:0      # ;
+Noun+Pl:+s     # ;   ! here literal + and s

LEXICON Nsame
+Noun+Sg:0      # ;   ! deer
+Noun+Pl:0      # ;   ! deer

LEXICON Nim
+Noun+Sg:0      # ;   ! kibbutz
+Noun+Pl:im     # ;   ! kibbutzim
```

The more interesting NregIm class for *cherub* could lead to a sublexicon that in turn leads to the normal single and plural endings and then adds the extra *-im* ending. A similar solution is possible for NregSes (for *bus*) and NregSame (for *fish*). We assume here that lower-side words like *bus+s* and *fish+s* are mapped subsequently to *buses* and *fishes* by productive alternation rules.

```
LEXICON NregSes
            Nreg ;
+Noun+Pl:ses    # ;

LEXICON NregIm
            Nreg ;   ! for regular endings: cherub/cherubs
+Noun+Pl:im     # ;   ! for the extra -im plural: cherubim

LEXICON NregSame
            Nreg ;   ! for the regular endings: fish/fishes
+Noun+Pl:0      # ;   ! for fish as plural
```

Other cases are harder. For example, the NregIes class for *money* will need to lead to the regular endings and to *-ies*, but simply adding *-ies* will yield **moneyies*, and alternation rules will need to be applied to yield the final correct form *monies*.

```
LEXICON NregIes
            Nreg ;    ! money/moneys
+Noun+Pl:ies    # ;   ! *moneyies
                      ! needing rules to map it to monies
```

For finite exceptional cases like these English plurals, trying to handle them in **lexc** leads to a proliferation of continuation classes, and one must often write ad hoc rules to handle a single root or a small handful of obviously exceptional roots. In general, one would prefer to keep the **lexc** grammar simpler and to reserve the writing of rules for more productive phenomena.

```
Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
        Nouns ;

LEXICON Nouns
cat     Nreg ;
dog     Nreg ;
dish    Nreg ;
money   Nreg ;
octopus Nreg ;
cherub  Nreg ;
fish    Nreg ;
```

Figure 5.8: A **lexc** Grammar that Generates only Regular Plurals

5.4.3 Handling Exceptions via Union and Composition

Handling Extra Plurals

Recall that we made a distinction in Section 5.4.1 between *extra* plurals, which need to be recognized in addition to the regular plurals, and *overriding* plurals, which need to override or supplant the regular plurals.

In the finite-state world, the notion of adding-to is easily accomplished via the union algorithm, so we'll start with the handling of extra plurals. We can start with a **lexc** grammar that undergenerates, producing only the regular plurals for nouns like *money*, *cherub*, *octopus* and *fish* as shown in Figure 5.8. All that remains is to union in the extra irregular plurals, which are easily defined by hand; sometimes this is referred to as adding forms by "brute force". This can be done in **lexc** itself as shown in Figure 5.9.

Recall that in **lexc**, the entries in any LEXICON are simply unioned together, so this grammar will union together all the paths built by following the Nreg continuation and all the extra hand-defined plurals. The proliferation of continuation classes is avoided, as is the writing of ad hoc alternation rules.

There are many ways to accomplish the same task. In **lexc**, the extra plurals could be grouped into a separate sublexicon, as in Figure 5.10. The irregular forms could also be defined, again by hand, in a completely separate **lexc** program, compiled into a separate network, and then the main network and the network of extra noun plurals could simply be unioned together in **xfst**. The extra plurals could also be defined directly in **xfst** itself and then unioned with the undergenerating lexicon from **lexc**. The **xfst** script in Figure 5.11 assumes that the **lexc** grammar has been compiled and that the network is stored in the file `lex.fst`.

```
Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
        Nouns ;
LEXICON Nouns
cat     Nreg ;
dog     Nreg ;
dish    Nreg ;
money   Nreg ;
money+Noun+Pl:monies    # ;  ! extra plural
octopus Nreg ;
octopus+Noun+Pl:octopi  # ;  ! extra plural
cherub  Nreg ;
cherub+Noun+Pl:cherubim # ;  ! extra plural
fish    Nreg ;
fish+Noun+Pl:fish       # ;  ! extra plural
```

Figure 5.9: Unioning in Irregular Extra Plurals by Brute Force

```
Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
        Nouns ;
        ExtraIrregularPlurals ;
LEXICON Nouns
cat     Nreg ;
dog     Nreg ;
dish    Nreg ;
money   Nreg ;
octopus Nreg ;
cherub  Nreg ;
fish    Nreg ;

LEXICON ExtraIrregularPlurals
money+Noun+Pl:monies    # ;
octopus+Noun+Pl:octopi  # ;
cherub+Noun+Pl:cherubim # ;
fish+Noun+Pl:fish       # ;
```

Figure 5.10: Unioning in Extra Plurals in a Separate Sublexicon

```
clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %+Noun:0  %+Pl:0
;

read regex   ExtraIrregularPlurals |  @"lex.fst" ;
save stack augmentedlex.fst
```

Figure 5.11: Unioning in Extra Plurals in **xfst**. The network that handles only regular plurals is stored in `lex.fst`. This **xfst** script unions the extra plurals with `lex.fst` and saves the result as `augmentedlex.fst`.

Handling Overriding Plurals with Composition, then Union

For handling the overriding plurals, we can

1. First define a **lexc** grammar that overgenerates regular plurals
2. Use composition to filter out the overgenerated plurals from the network, and then
3. Use union to add the overriding plurals

As we shall see in the next section, this idiom is helpfully encapsulated in a single finite-state algorithm called priority union.

To illustrate the idiom step by step, assume that we start with the **lexc** grammar in Figure 5.12 that overgenerates regular plurals for *sheep*, *deer*, *kibbutz* and *automaton*. It also undergenerates by not handling the irregular plurals for these roots and for words like *money* that have both regular and irregular plurals.

In a separate **xfst** grammar, we can define a set of overriding plurals in addition to the set of extra plurals. The **xfst** script in Figure 5.13 assumes that the network compiled from the **lexc** grammar is stored in the file `lex.fst`. Follow the comments in the script to see how the incorrect regular plurals are first removed by an upper-side filtering composition before the irregular plurals are unioned into the result.

Handling Overriding Plurals via Priority Union

As illustrated in the previous section, irregular plurals can override unwanted regular plurals via an idiom that combines upper-side filtering and union. This idiom is

```
Multichar_Symbols  +Noun +Sg +Pl

LEXICON Root
        Nouns ;

LEXICON Nouns
cat     Nreg ;
dog     Nreg ;
dish    Nreg ;
money   Nreg ;  ! undergenerates; no monies pl.
octopus Nreg ;  !                 no octopi pl.
cherub  Nreg ;  !                 no cherubim pl.
fish    Nreg ;  !                 no fish pl.

sheep         Nreg ;  ! overgenerates *sheeps
deer          Nreg ;  !               *deers
kibbutz       Nreg ;  !               *kibbutzes
automaton     Nreg ;  !               *automatons
```

Figure 5.12: A **lexc** Grammar that Both Undergenerates and Overgenerates Plurals

so useful that it has been packaged in a single finite-state algorithm called PRIORITY UNION, or more precisely in this case, *upper-side* priority union. In a regular expression, upper-side priority union is notated with the operator `.P.` as in the **xfst** script in Figure 5.14. Note that the .P. operation is ordered, with the left-side network operand having the priority. For the operation `L .P. R`, the result is a union of `L` and `R`, except that whenever `L` and `R` have the same string on the upper side, the path in `L` overrides (has precedence over) the path in `R`.

For completeness, the regular-expression language also includes the operator `.p.`, for lower-side priority union, but the `.P.` operator is more useful in practice given the **Xerox** convention of putting baseforms and tags on the upper side of a transducer.

5.5 Conclusions

5.5.1 Take Software Engineering Seriously

The building of a finite-state morphological analyzer is a significant software project that deserves careful planning. Developers should make sure that they have adequate hardware, up-to-date versions of the finite-state software, a version-control system, backups, and the ability to automate complex compilations using makefiles. This may require cooperation among linguists and computer scientists.

```
clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %+Noun:0  %+Pl:0
;

define OverridingIrregularPlurals [ {sheep}
| {deer}
| {kibbutz}:{kibbutzim}
| {automaton}:{automata}
]  %+Noun:0  %+Pl:0
;

! now set Filter to the upper-side language of
! OverridingIrregularPlurals; it contains strings like
! sheep+Noun+Pl and automaton+Noun+Pl

define Filter  OverridingIrregularPlurals.u ;

! read lex.fst from file, and compose on top of it
! the complement of the Filter.  This filters out
! the paths that have sheep+Noun+Pl, deer+Noun+Pl,
! kibbutz+Noun+Pl and automaton+Noun+Pl on the
! upper side.  The operation therefore removes the
! overgenerated regular plural paths.

define FilteredLex  ~Filter .o. @"lex.fst" ;

! now simply union in the extra plurals and
! the overriding plurals to create the
! final result

read regex
FilteredLex |
ExtraIrregularPlurals |
OverridingIrregularPlurals ;

save stack augmentedlex.fst
```

Figure 5.13: Overriding via a Composed Filter and Union in an **xfst** Script

```
clear stack

define ExtraIrregularPlurals [ {money}:{monies}
| {octopus}:{octopi}
| {cherub}:{cherubim}
| {fish}
] %+Noun:0  %+Pl:0
;

define OverridingIrregularPlurals [ {sheep}
| {deer}
| {kibbutz}:{kibbutzim}
| {automaton}:{automata}
]  %+Noun:0  %+Pl:0
;

! use of upper-side priority union .P.
! overrides bad plurals with good plurals

read regex
[ OverridingIrregularPlurals .P. @"lex.fst" ] |
ExtraIrregularPlurals ;

save stack augmentedlex.fst
```

Figure 5.14: Simplified Overriding Using the Upper-Side Priority-Union Operator .P.

5.5.2 Take Linguistic Planning Seriously

The bulk of linguistic planning and modeling should be done before the coding starts, although the rigor of writing an automatic morphological analyzer is also a discovery process that can lead to many insights. The overall plan of the system, showing the division into lexicon and rule modules, and the way that various transducers combine together into the final lexical transducer(s), should be formalized in makefiles and visualized in wall charts for easy reference. From the beginning, developers should have a clearly defined strategy for handling irregular forms and for supporting multiple final applications.

Handling Irregular Forms

When making decisions concerning the handling of morphological phenomena, the principle of CREATIVE LAZINESS should apply. For morphotactic and morphophonological phenomena that are truly productive, it is best to handle them

using **lexc** and alternation rules. Writing rules is the "lazy" way to handle lots of similar examples that would be tedious to handle by brute force. But for finite, idiosyncratic irregularities like the English plurals shown above, it is often cleaner and easier to create an initial lexicon transducer that overgenerates and undergenerates, and then fix it using union, composition, and perhaps priority union. Handling a very finite number of exceptions via brute force is easier and lazier than writing a lot of ad hoc rules, most of which are, in any case, suspicious from any phonological point of view. These tricks facilitate the lexicography and keep the grammars smaller and easier to maintain.

Customization and Multiple Use

Expert finite-state developers plan from the beginning for flexibility and adaptability. Starting with a common core network, which may by itself be useless for any particular application, the goal is to be able to modify that core network via finite-state operations, notably composition, union and priority union, to create a multitude of variant systems that support

- Multiple final applications, such as morphological analyzers, automatic dictionary lookup systems, disambiguators, parsers, machine-translation systems, etc. (see page 287)
- Multiple dialects of the same language (see page 288)
- Multiple orthographies (e.g. after a spelling reform, see page 290)
- Multiple registers, including or excluding vulgar, slang and substandard words (see page 291)
- Multiple levels of strictness, e.g. properly accented vs. deaccented Spanish (see page 292)

The key in this customization technique, as it is in the filtering out of morphotactic generation (Section 5.3.3) is to understand that a finite-state network is an object that can be modified by finite-state operations, without having to return and edit the source files that defined it in the first place.