Chapter 6

Testing and Debugging

6.1 The Challenge of Testing and Debugging

The testing of any sizable natural-language processing system is notoriously difficult. A full-scale morphological analyzer will typically grow to contain tens of thousands of baseforms and millions of surface forms. An analyzer that handles productive compounding will in fact handle an infinite number of surface forms. When you edit your lexicons or rules and recompile the system, how can you tell what was added, lost or perhaps broken in the process?

Fortunately, the finite-state calculus provides many powerful ways to test a system against large corpora, wordlists, other lexicons, Lexical Grammars, and previous versions of the system itself. Networks can also be examined in various ways, in particular to print lists of their alphabets or labels, to flush out typical programming errors. This chapter describes the idioms for testing and debugging finite-state morphology systems using the Finite-State Calculus itself.

6.2 Testing on Real Corpora

6.2.1 Hand-Testing vs. Automated Testing

In the days before computers, or at least before computational linguistics became practical, linguists with formal instincts and training would often produce elaborate grammars on paper. While much excellent linguistics was done in this way, *faute de mieux*, all testing had to be done by hand; and rare is the human being with the patience and concentration necessary to run even a hundred examples reliably through a complicated grammar, let alone thousands. It is not surprising that most paper grammars are poorly tested and inaccurate.

With **Xerox** finite-state compilers and interfaces, grammars are compiled and combined into highly efficient Lexical Transducers that typically analyze real text at rates of thousands of words per second. However, the testing facilities built into the development tools themselves are quite limited, intended mainly for quick

manual checking of a few words here and there during development. The **apply up** and **apply down** (page 86) commands in **xfst** can handle single tokens, and even accept input from a wordlist file that has one token per line; but neither tool offers a way to output to file, or to pipe the output of morphological analysis to a subsequent process like a disambiguator or parser.

For serious testing, we need to automate the process, applying our Lexical Transducers to corpora containing hundreds, thousands or even millions of words, and outputting successful analyses and FAILURES (not-found words) to various output files for later review. By CORPUS, plural CORPORA, we mean any files containing normal running text.

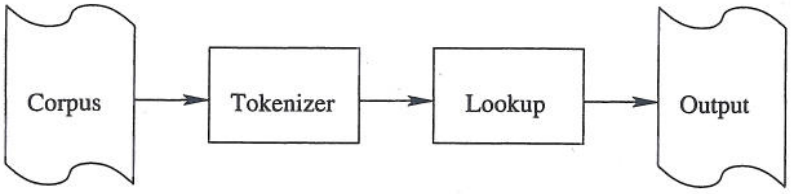

Figure 6.1: An Overview of Automated Testing

The testing procedure, shown in overview in Figure 6.1, is to pipe the corpus to a TOKENIZER utility that breaks it up into individual TOKENS (one to a line), which are typically orthographical words, and which are in turn piped to a LOOKUP utility that applies pre-compiled finite-state transducers to try to analyze them. The software available with this book includes command-line utilities called **tokenize** and **lookup** which we will examine more closely in Chapter 9. The use of **tokenize**, in particular, requires the definition of a special language-specific tokenizing transducer that we don't want to deal with just yet.

> Whereas **lexc**, **xfst** and **twolc** are DEVELOPMENT TOOLS for creating finite-state networks, **tokenize** and **lookup** are RUNTIME APPLICATIONS that apply your transducers to do serious testing and practical natural-language processing.

In this chapter, we will provide some useful testing idioms that can be copied and used without necessarily understanding all the details and options.

6.2.2 Testing for Failures

Morphological analyzers are never complete or perfect, and we need various ways to test them and find the problems.

Collecting Words that Aren't Analyzed

When testing your morphological analyzer on a large corpus, it is usually practical to collect and look only at the failures, that is, at the input words for which the analyzer fails to find any analysis at all. In this section, we will present some useful idioms for collecting and sorting such failures or "sins of omission". In Section 6.2.3 we will look at more positive testing that discovers "sins of commission" and partial analysis failures.

Failures can result from mistakes in your **xfst** rules, or from mistakes in your **lexc** morphotactic description; other failures result from the simple fact that you haven't yet added the appropriate baseforms to the **lexc** lexicon. After the basic outlines of your analyzer are written and working, subsequent development becomes a never-ending round of addition and refinement of your rules and lexicons, trying to make the language accepted by the Lexical Transducer correspond to the target natural language as closely as possible.

Assuming that your corpus is stored in file `mycorpus.txt` and that your morphological analyzer is stored in `mylanguage.fst`, the following testing idiom generates an output file of not-found words called `failures.all`.[1]

```
unix> cat mycorpus.txt | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup mylanguage.fst | \
grep '+?' | \
gawk '{print $1}' > failures.all
```

This somewhat intimidating command uses the standard Unix command-line utilities **cat**, **tr**, **grep** and **gawk**; and it requires that the **lookup** utility, included in the **Xerox** software licensed with this book, be installed in a directory that is in your search path. Consult a local guru if you need help installing programs or modifying the command to work in your particular operating system.

Without going into too much detail, the command operates as follows:

1. The text in the corpus is piped to the **tr** utility.
2. **tr** translates all non-alphabetic characters into newlines, squeezing multiple newlines into a single newline, and pipes its output to **lookup**
3. **lookup** loads and uses the pre-compiled network stored in the binary file `mylanguage.fst` to look up each word, piping its output (in its own idiosyncratic format) to **grep**
4. **grep** acts as a filter, searching for and outputting only those lines containing `+?`, which is what **lookup** outputs as the pseudo-solution for a word when no solution is found

[1]The backslashes in the example precede newlines and effectively tell the operating system to treat the command as if it were all typed on a single long line. You can literally type the command on a single line if you wish, but in that case the backslashes should not be typed.

5. and finally **gawk** prints out the first field of the **lookup** output, which is the not-found word itself.

The result is output to a file, `failures.all`, containing all the words from `mycorpus.txt` for which no solution at all was found by `mylanguage.fst`.

Finite-state morphological analyzers are so fast that it is completely practical to test repeatedly against corpora containing millions of words. But even if you are saving only the failures, this can still result in unmanageably large output files, usually including many duplicates. Developers need some way to collapse the duplicates and identify which failures are the most important to fix first.

The simplest solution is to sort the collected failures by frequency of occurrence. If the failures are stored in `failures.all`, the following Unix idiom will produce `failures.sorted`, with the most frequent failures sorted to the top of the file.

```
cat failures.all | sort | uniq -c | sort -rnb > failures.sorted
```

The first `sort` will arrange all the not-found words alphabetically, which will have the effect of putting all multiple copies of the same word together in the output. If you are analyzing an English corpus and the word *canton* is missing three times, and the word *dahlia* is missing four times, the output of this first sort would look like this:

```
...
canton
canton
canton
dahlia
dahlia
dahlia
dahlia
...
```

These results are then piped to the **uniq** utility with the **-c** flag, which collapses multiple adjacent lines containing the same word into a single line with a count of the original number of lines, e.g.

```
3       canton
4       dahlia
```

These results are piped in turn to the **sort** utility, but this time with the flags **-rnb**, which cause the lines to be sorted in reverse numerical order, based on the first numerical field.

```
4       dahlia
3       canton
```

The most frequent not-found words end up at the top of the `failures.sorted` file. You can then examine the output file manually and concentrate on the top 100 or so most frequently missed words, either adding the necessary baseforms to the lexicon or fixing the **lexc** description or the rules that prevent the forms from being analyzed. If the not-found words contain misspellings, then you may simply want to correct the test corpus. The addition of one new baseform will often resolve multiple missing surface words, so after fixing the top 100 missing words the whole test is usually best repeated to find the next 100 most important missing words. Retesting the whole corpus at reasonable intervals is also a valuable form of REGRESSION TESTING, highlighting errors that you inadvertently introduce while trying to fix your lexicons and rules.

> Use the testing idioms to run even huge corpora through your morphological analyzer, looking for failures. Re-running the corpus at reasonable intervals is a useful form of regression testing.

If you know what you're doing, such commands are infinitely customizable to meet your particular needs. For example, the idiom shown above uses the **tr** utility to replace all non-alphabetic characters with the newline character, where alphabetic characters are designated by the built-in `:alpha:` notation, which is locale-specific. If necessary or desired, you can also specify the set of alphabetic letters explicitly by enumeration, as in the following example, which would be suitable for a language where the alphabet contains **A** to **Z**, **Ñ**, **Ä**, **Ï**, **Ü** (all in both cases) and the apostrophe **'**.

```
cat corpus | \
tr -sc "[A-Z]ÑÄÏÜ[a-z]'ñäïü" "[\n*]" | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' > failures_file
```

The essence of tokenization is simply to insert a newline character between tokens, and that can also be done using Perl or a similar scripting language. The Perl script in Figure 6.2, which requires Perl version 5.005_03 or higher, does a reasonable job of tokenization for English text, and any Perl hacker can customize it for a particular language. Unlike the **tr**-based tokenizers shown above, this solution separates and retains the punctuation characters as tokens. The script reads from standard input and writes to the standard output; so if it is stored in `tokenize.pl`

```
#!/usr/bin/perl -w
# requires Perl version 5.005_03 or higher
# you may need to edit the path above to point
#   to the Perl executable in your environment

while (<STDIN>) {
  chop ;
  s/^\s+// ;
  s/\s+$// ;
  /^$/ && next ;    # ignore blank lines

  # separate left punc(s) from start of words
  s/^(['"]+)(?=\w)/$1\n/g ;
  s/(?<=\W)(['"]+)(?=\w)/$1\n/g ;

  # separate right punc(s) from end of words
  s/(?<=\w)(['".,;:?!]+)(?=\W)/\n$1/g ;
  s/(?<=\w)(['".,;:?!]+)$/\n$1/g ;

  # separate strings of multiple punc
  s/(?<=[^\w\s])([^\w\s])/\n$1/g ;

  # break on whitespace
  s/\s+/\n/g ;
  print $_, "\n" ;
}
```

Figure 6.2: A Perl Script that Tokenizes English Text. This script reads from standard input, writes to standard output, and preserves punctuation marks as tokens.

and made an executable file,[2] it can replace **tr** in the pipe.

```
cat corpus | \
tokenize.pl | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' > failures_file
```

The idioms for failure collection and sorting can of course be collapsed together into a single command, which is best typed into a file, edited to match your filenames, and run using the Unix utility **source**.

[2]The Unix command to make it executable is `chmod 755 tokenize.pl`. See your local guru for details.

```
cat corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup lexical_transducer | \
grep '+?' | \
gawk '{print $1}' | \
sort | uniq -c | sort -rnb > sorted_failures
```

Because the sorting allows the linguist to concentrate on the most important missing words, this method is practical even with huge corpora, and new texts can be added to the test corpus at any time. Eventually, after many rounds of testing and lexical work, when the system is finding solutions for over 97% of the corpus words, the most frequently missed words will tend to be typographical errors, foreign words, proper names, and various rare nouns and adjectives, with most of the missing words occurring only once or twice even in a very large corpus. After this point, in any natural-language-processing system, it becomes increasingly difficult to improve the coverage significantly.

Testing for Failures with a Guesser

Another way to test for failures is to construct both a normal morphological analyzer and an auxiliary analyzer called a GUESSER. Guessers are built around a definition of phonologically possible roots, rather than enumerated known roots, and they are particularly valuable for testing in the early stages of a project, when the dictionary of known roots is small. The use of guessers, which requires a deeper understanding of the **lookup** application than we can approach here, is explained in Section 9.5.4, page 444.

6.2.3 Positive Testing

```
Lexical:   spit+Verb+PresPart
Surface:   spiting
```

Figure 6.3: A Sin of Commission, Analyzing *spiting* Erroneously as the Present Participle of *spit*.

As useful as the saving, sorting and repairing of failures is, it does not help to discover cases where the Lexical Transducer analyzes words incorrectly, i.e. overrecognition or SINS OF COMMISSION, or where the analyzer finds some but not all of the valid analyses for a word, which we might term incomplete analysis or SINS OF PARTIAL OMISSION.

A typical sin of commission would be the English *spiting* being analyzed incorrectly as the present participle of the verb *spit* as in Figure 6.3. In reality, *spiting*

is a form of the verb *spite*; the verb *spit* would properly have the present participle *spitting*, with two *t*s. The incorrect analysis of *spiting* as a form of *spit* will need to be fixed by the linguist, but the biggest problem is not usually fixing such problems but finding them in the first place. When any natural-language processing system contains tens of thousands of baseforms and analyzes and generates millions of surface forms, finding a *spit:spiting* problem is like searching for the proverbial needle in a haystack.

Similarly, sins of partial omission or incomplete analysis occur when a surface word is ambiguous, and the system does not find all of its valid readings. Consider the English words *table* and *tables*, which have both noun and verb readings. If the lexicon contains the noun baseform *table* but happens not to contain the verb baseform *table*, then even in an otherwise perfect system only the noun solutions will be found:

```
Lexical:    table+Noun+Sg          table+Noun+Pl
Surface:    table                  tables
```

The saving and sorting of failures will of course not identify such cases where a correct but incomplete analysis is being performed.

While there is no perfect solution to the problem of flushing out sins of commission and partial omission, one useful and highly recommended daily exercise is the following:

1. During development, pick out a new paragraph or two of text every day and type it or store it in a separate file. The text should come from a variety of sources, for example the latest magazines, newspapers, a book you happen to be reading, a message that someone posted on the Internet, etc. In general, collect as much new text as you can find for the large test corpus, but choose just one or two random paragraphs of text for this daily exercise.

2. Run the paragraphs through the tokenizer and **lookup**, and then examine *all* the output by hand to see if the analyses are both correct and complete.

3. Fix the lexicons and rules as appropriate.

A useful idiom for such positive testing is the following, where the input file contains a smallish text.

```
cat small_corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
lookup lexical_transducer > output_file
```

The output file can then be examined manually using any text editor. If you have the luxury of independent testers, make it part of their job to do such POSITIVE TESTING of real running text every day, entering manageably small texts and looking at absolutely everything that the system returns. It is a necessary balance to the NEGATIVE TESTING that looks only for failures.

```
The senator promised to   table the bill
DET NOUN    VERB     PREP NOUN  DET NOUN
```

Figure 6.4: A Typical Tagging Error Based on Incomplete Morphological Analysis

Inevitably, sins of partial omission will continue to present themselves for months and years to come. Sometimes they become obvious only after a part-of-speech disambiguator ("tagger") is written, and the tagger is found to make mistakes as in Figure 6.4. Here we have a sentence using *table* as a verb, but the tagger is labeling it incorrectly as a *NOUN*. When tracking down such problems, it is often discovered that the morphological analysis simply lacks the verb baseform *table* and so is presenting the tagger with an unambiguous noun solution.

6.3 Checking the Alphabet

6.3.1 The Gap in Symbol-Parsing Assumptions

During development, a Lexical Transducer will often be found to contain mistakes in its alphabet, either missing symbols or strange, undesired symbols that are introduced by errors in the various source files. You should "check the alphabet" regularly to find such mistakes, which come from the following sources:

1. **xfst** makes the assumption that any string of characters written together, e.g. *abc*, represents a single multicharacter symbol. **lexc**, for good lexicographical reasons, makes the the opposite "exploding" assumption, i.e. that *abc* represents three separate symbols concatenated together; the only way to override the **lexc** exploding assumption is to declare written sequences of symbols as `Multichar_Symbols` to be kept together.

2. **xfst** treats a number of characters as special, including + (for Kleene Plus), * (for Kleene Star), | (alternation), & (intersection), ^ (iteration), etc. In **lexc**, on the other hand, such symbols are not generally special, unless they occur inside angle brackets or in `Definitions`, e.g. `< a b* c >`, where the formalism and so the special characters of regular expressions apply.

Even though there are good reasons for the different assumptions, the linguist who switches back and forth between **lexc** and regular expressions in **xfst** will often have trouble keeping the different assumptions straight. Even within **lexc** files, the linguist has to juggle different assumptions inside and outside of angle brackets. These conflicting assumptions about how to parse typed strings into symbols are the root cause of most alphabet problems in finite-state systems.

```
LEXICON Adjectives
dark            Adj ;
quick           Adj ;
heavy           Adj ;

LEXICON Adj
+Adj:0          # ;
```

Figure 6.5: +Adj Intended to be a Multicharacter Tag

6.3.2 Failure to Declare Multichar_Symbols in lexc

The most common alphabet problem in **lexc** comes from the failure to declare all the intended Multichar_Symbols. If you write LEXICONs as in Figure 6.5, intending +Adj to be a multicharacter symbol, but failing to declare it in the `Multichar_Symbols` statement, **lexc** will happily and silently explode the string +Adj into four separate symbols, leading to the compilation of a **lexc** transducer having paths like the following:

```
Upper:      d a r k + A d j
Lower:      d a r k 0 0 0 0
```

If you analyze the string "dark" using this transducer (e.g. using **apply up** in **xfst**), the result string will look correct, i.e. it will be printed out superficially as "dark+Adj", but it will in fact be wrong if you intended +Adj to be a single symbol. Note that the plus sign + is not special in **lexc**, except when it appears in a regular expression inside angle brackets; **lexc** will not treat the sequence **+Adj** as a multicharacter symbol unless it is declared. Failure to declare all the intended multicharacter symbols in **lexc** is a common mistake that results in multiple normal symbols where you intended a single multicharacter symbol.

6.3.3 Inadvertent Declarations of Multicharacter Symbols in xfst

```
xfst[]:  read regex [{talk} | {walk}] %+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
```

Figure 6.6: An **xfst** Regular Expression that Implicitly Declares *ed* and *ing* as Multicharacter Symbols. Such cases are almost always errors and can cause your Lexical Transducer to act mysteriously.

The opposite problem occurs in **xfst** and in **lexc** `Declarations` and regular expressions between angle brackets, when the linguist forgets that strings of characters written together are automatically parsed as multicharacter symbols. An **xfst** regular expression like the one in Figure 6.6 implicitly declares *ed* and *ing* as multicharacter symbols simply because the letters are written together without any separating spaces. **xfst** replace rules are also regular expressions, and the rule in Figure 6.7 implicitly declares *amus* as a multicharacter symbol.

```
xfst[]: define Rule1 %+Verb
%+Pres %+Indic %+Act %+1P %+Pl -> amus ;  ! error
```

Figure 6.7: An **xfst** Replace Rule that Inadvertently Declares the Multicharacter Symbol *amus*. Such mistakes are common and should be guarded against. The correct rule would have a spaced-out `a m u s` or curly-braced `{amus}` on the right-hand side.

In the **lexc** example shown in Figure 6.8, the sequences *ed* and *ing* appearing in regular expressions, inside **lexc** angle brackets, are similarly treated as single symbols and are automatically added to the alphabet of the resulting network. The grammars in both Figure 6.6 and Figure 6.8 will result in transducer symbol pairings like the following

```
Upper:        t a l k +Verb +PresPart
Lower:        t a l k 0     ing
```

where, again, *ing* is being stored and manipulated as a single symbol. Having multicharacter symbols like *ed*, *ing* and *amus* in your Lexical Transducer is almost always a Bad Thing, especially if subsequently composed rules are looking for the symbol sequences `[ e d ]`, `[ i n g ]` and `[ a m u s ]` that just aren't there.

6.3.4 When to Suspect an Alphabet Problem

You should *always* suspect alphabet problems. Alphabet problems are so common that experienced developers learn to check for them regularly, and idioms for this purpose are presented in the next section. The overt sign of an alphabet problem is when it looks like a rule should fire, but doesn't, or when it looks like a word should be analyzed, but isn't. The underlying problem is often that the alphabet is corrupted, causing your input strings (words) to be divided into symbols in unexpected ways.

For example, consider the **xfst** session in Figure 6.9 that builds a transducer for five regular English verbs and then tests it using **apply up**. The regular expres-

```
Multichar_Symbols +Verb +Bare +3PSg  +Past +PresPart

LEXICON Root
        Verbs ;

LEXICON Verbs
talk    V ;
walk    V ;

LEXICON V
+Verb:0 VEndings ;

! bad definition of verb endings

LEXICON VEndings
< %+Bare:0  >   # ;
< %+3PSg:s  >   # ;
< %+Past:ed >   # ;       ! Error
< %+PresPart:ing >  # ;   ! Error
```

Figure 6.8: Unintentional Declaration of the Multicharacter Symbols *ed* and *ing* inside **lexc** Angle-Bracket Entries. Such declarations are almost always errors. The presence of such unintended multicharacter symbols in the alphabet of a network can often lead to mysterious problems in analysis and the application of rules.

sion that defines the network inadvertently declares *ed* and *ing* as multicharacter symbols, adding them to the alphabet of the network.

The application routines (such as **apply up** and **apply down**) need to parse input strings into symbols, including multicharacter symbols, before matching them against transducer paths. This symbol tokenization is always done relative to the sigma of the network being applied, and in cases of ambiguity they always give precedence to longer symbol matches. Thus if **e**, **d** and **ed** are in the alphabet, and the input is a string like "`edits`", the input string is broken up into symbols as *ed-i-t-s* and is not found because no strings in the language begin with the multicharacter symbol *ed*. (See Section 3.6.3, page 187 for more on this topic.)

Another not immediately obvious problem with the transducer in Figure 6.9 is that "`snorting`" is analyzed, but probably not as intended. Because *ing* is a multicharacter symbol in the alphabet, the input string "`snorting`" is broken up into symbols as *s-n-o-r-t-ing*; it is found because the lower-side language contains this string with the symbol *ing* on the end. Beginning developers often think that they can get away with a few multicharacter symbols of this type. However, when the verb stems *ring* and *ping* are later added to the lexicon, the presence of the

```
xfst[n]: clear stack
xfst[0]: read regex [{talk}|{walk}|{bark}|{snort}|{edit}]
%+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
xfst[1]: apply up talk
talk+Verb+Bare
xfst[1]: apply up walks
walk+Verb+3PSg
xfst[1]: apply up barked
bark+Verb+Past
xfst[1]: apply up snorting
snort+Verb+PresPart
xfst[1]: apply up edit
xfst[1]: apply up edits
xfst[1]: apply up edited
xfst[1]: apply up editing
```

Figure 6.9: A Typical Alphabet Problem. Because *ed* is inadvertently declared as a multicharacter symbol, input strings like "`edit`", which contain *ed*, cannot be found. When a word looks like it should be analyzed, but mysteriously isn't found, it is often a clue that the alphabet of the network is corrupted with unintended multicharacter symbols.

multicharacter symbol *ing* in the alphabet would prevent any form of "`ping`" or "`ring`" from being analyzed. Inappropriate multicharacter symbols usually come back to haunt you.

The other classic manifestation of an alphabet problem is when a rule looks like it should apply to and modify a network, but doesn't. Suppose that invoking **print random-lower** in **xfst** shows that the lower side of a network contains strings such as "`ringort`", and that the rule

```
g -> 0 || n _ [o | a]
```

is subsequently applied to the lower side, with the intention of effecting the following mapping:

```
Upper:  ringort
Lower:  rinort
```

If it definitely looks like a rule should fire, but doesn't, then it may be the case that the input string, printed out by default as "`ringort`", might in fact consist of the symbols

```
r ing o r t
```

In such a case, the rule cannot apply because there is no **g** symbol in the string.

It is almost always a mistake to declare multicharacter symbols like *ed* and *ing* that look like concatenations of normal alphabetic characters.[3] Following the **Xerox** convention, which includes a punctuation mark like the plus sign (+) or the circumflex (^) or surrounding square brackets in every multicharacter symbol can help avoid and correct alphabet errors.

```
xfst[n]: clear stack
xfst[0]: load stack MyLanguage.fst
xfst[1]: upper-side net
xfst[1]: print labels

xfst[n]: clear stack
xfst[0]: load stack MyLanguage.fst
xfst[1]: lower-side net
xfst[1]: print labels
```

Figure 6.10: Idioms to Examine the Labels on the Upper and Lower Side of a Transducer. These simple tests should be performed regularly to check the alphabet for missing and extraneous multicharacter symbols caused by errors in the source files.

6.3.5 Idioms for Checking the Alphabet

Learn to suspect and test regularly for alphabet problems. In particular, if it appears that a rule should be applying, but doesn't, or that an input string should be accepted, but isn't, the very first thing to do is to "check the alphabet".

Printing Out the Sigma and Labels

The way to test a whole network for alphabet problems is simply to load it onto the top of an **xfst** stack and print out the labels.

```
xfst[0]: load stack MyLanguage.fst
xfst[1]: print labels
```

The result, which shows all the symbol *pairs* in the system, can be a bit confusing, so printing the SIGMA, i.e. the alphabet of symbol characters, is usually more readable.

[3] Justifiable exceptions may be appropriate for orthographies where ngrams represent single phonemes and cannot be confused with sequences of separate letters. For example, if a particular orthography uses **p**, **t** and **k** to represent unaspirated consonants and **ph**, **th** and **kh** to represent their aspirated counterparts, and if **h** is not used elsewhere in the orthography; then **ph**, **th** and **kh** are unambiguous and could be safely declared as multicharacter symbols.

```
xfst[0]: load stack MyLanguage.fst
xfst[1]: print sigma
```

To better localize alphabet problems, it is often preferable to look at the labels of the upper and lower sides separately as in Figure 6.10. Just examine the list of labels printed out; if you spot anything bizarre, like surface *ing* or *ed* being treated as single symbols, isolate the words in which they appear by composing simple filters.

If the network is found to contain an unexpected multicharacter symbol like *ing* on the lower side, the idiom in Figure 6.11 will isolate those words in which it appears, which can help you locate the error in your source files. Note that the idiom composes `$[ing]`, which denotes the language of all strings containing the multicharacter symbol *ing*, on the lower side of the network, where the surface strings are visible.

```
xfst[0]: read regex  @"MyLanguage.fst" .o. $[ing] ;
xfst[1]: print random-lower
xfst[1]: print random-upper
```

Figure 6.11: Idiom to Isolate Words Containing a Suspicious Multicharacter Symbol *ing* on the Lower Side of the Network

Once the offending strings have been identified, the fixes are usually pretty obvious: If *ing* is being treated, unexpectedly, as a multicharacter symbol, then fix the rule or other regular expression that is implicitly declaring it. Conversely, if you expect to see a particular multicharacter lexical symbol like `+Adj`, and it doesn't appear in the alphabet of the upper-side language, then you should go back to your **lexc** description and declare it.

If you're not sure what to look for, identifying undeclared multicharacter symbols is slightly more difficult. If you follow the **Xerox** conventions for spelling multicharacter symbols and features, and if you see a separate plus sign (+) or circumflex (^) or square bracket in the alphabet of the upper-side language, then this is often a tell-tale sign that you have failed to declare either a tag like `+Foo` or `[Foo]`, or a feature like `^FEAT`. To isolate and identify which multicharacter symbols and tags are undefined, compose a little filter on *top* of `MyLanguage.fst` that matches all and only strings containing the unexpected separate punctuation marks. This idiom is illustrated in Figure 6.12; the random strings printed out should provide enough information to help you track down which multicharacter symbols were not declared.

```
xfst[0]: read regex  $[ %^ | %+ | %[ | %] ]
.o.
@"MyLanguage.fst" ;
xfst[1]: print random-upper
```

Figure 6.12: An Idiom to Help Find Undeclared Multicharacter Symbols. If you include a plus sign or a circumflex or square brackets as part of the spelling of every multicharacter symbol, then finding any of them as a separate symbol on the upper side of a network may be the sign of an intended multicharacter symbol that was not properly declared in **lexc**.

It is only a **Xerox** convention to include some kind of punctuation, e.g. a plus sign or circumflex, in the spelling of every multicharacter symbol. Another convention that has been used is to spell all tags with surrounding square brackets, e.g. **[Noun]**. Such conventions make multicharacter symbols stand out visually and can help to discover intended multicharacter symbols that were not properly declared.

Setting print-space ON

When working interactively with networks in **xfst**, developers often invoke **print random-upper** and **print random-lower** to see what the strings in a network look like. By default, these **print** routines print out connected strings to which the network could successfully be applied, but they do not show how those strings are tokenized into symbols. When **print random-lower** is invoked as in the following example, no alphabet problem is apparent.

```
xfst[n]: clear stack
xfst[0]: read regex [{talk}|{walk}|{bark}|{snort}|{edit}]
%+Verb:0
[ %+Bare:0 | %+3PSg:s | %+Past:ed | %+PresPart:ing ] ;
xfst[1]: print random-lower
edits
edits
snorted
snort
bark
walk
barks
```

```
edit
snorting
walks
barked
walk
walking
talking
walked
xfst[1]:
```

If an alphabet problem is even slightly suspected, a good first step is to set the **xfst** interface variable **print-space** to **ON**. This causes the **print** routines to print a space between symbols, showing clearly which sequences are being tokenized as multicharacter symbols.

```
xfst[1]: set print-space ON
variable print-space = ON
xfst[1]: print random-lower
t a l k
b a r k ed
t a l k
e d i t ed
w a l k s
w a l k s
e d i t s
s n o r t
b a r k
w a l k
s n o r t s
b a r k
w a l k ed
t a l k s
e d i t ing
```

With **print-space** set to **ON**, the suspicious multicharacter symbols *ing* and *ed* are hard to miss. For more information on the use of **print-space**, see Sections 7.4.2, 3.6.2 and 3.8.1.

6.4 Testing with Subtraction

A regular relation, encoded by a finite-state transducer, is a mapping between two regular languages, where (following the usual **Xerox** conventions) the upper language is typically a set of strings that consist of a baseform and multicharacter-symbol tags, and the lower language is a set of surface strings. Using the finite-state calculus, and particularly the subtraction operation in **xfst**, we can compare these languages against other languages that we or others have defined.

It should be remembered that regular relations (and the transducers that encode them) are not closed under subtraction. In practical terms, this means that you cannot generally subtract one transducer (relation) from another. Regular *languages* are, however, closed under subtraction, and the following examples and idioms require extracting the upper-side or lower-side language from a transducer before subtraction is performed.

6.4.1 Testing the Lexical Language

All developers are strongly urged to plan out their analyses as much as possible before starting the coding. Alternatively, developers should feel free to experiment for a while and then restart the work after they have a better idea of the directions to take.

An important part of planning is to choose a reasonable tagset and to decide on the physical order in which co-occurring tags will appear. These choices can and should be formalized in a LEXICAL GRAMMAR that describes all possible legally constructed lexical strings. Lexical Grammars can be written in some cases as single monolithic regular expressions to be compiled by **read regex** inside **xfst**, but it is usually better to define them using **xfst** scripts, which are executed with the **source** command. For a review of **xfst** scripts, see Section 3.3.3, page 117.

Let us assume that we want to define an upper-side language where the strings consist of a baseform followed by multicharacter-symbol tags. A regular expression that matches all possible baseforms will look something like the following:

```
[ a | b | c | d | e | f | g | h | i | j | k |
  l | m | n | o | p | q | r | s | t | u | v |
  w | x | y | z |
  A | B | C | D | E | F | G | H | I | J | K |
  L | M | N | O | P | Q | R | S | T | U | V |
  W | X | Y | Z ]+
```

This rather crude expression, which matches any string of one or more letters from an alphabet, should be modified, of course, to contain all and only the letters that can validly appear in the baseforms of your language. Following, i.e. concatenated after, these pseudo-baseforms are the legal strings of tags, described in a regular expression something like the following:

```
[
%+Noun ([%+Aug | %+Dim]) [%+Masc | %+Fem] [%+Sg | %+Pl]
|
%+Adj ([%+Comp | %+Sup]) [%+Masc | %+Fem] [%+Sg | %+Pl]
|
```

```
%+Verb [%+Past | %+Pres | %+Fut] [%+1P | %+2P | %+3P]
                          [%+Sg | %+Pl]
]
```

In this illustration, we assume that nouns are optionally marked as augmentative or diminutive, obligatorily marked as masculine or feminine, and obligatorily marked as singular or plural via tags that appear in that order. Similar distinctions and orders are defined for adjectives and verbs. Of course, the tags, tag orders and overall tagging patterns will differ for each language, and there may be prefix tags appearing before the baseform. A real Lexical Grammar will be far more complex than the example above, perhaps extending to several pages.

```
clear stack
define InitialC [p|t|k|b|d|g|s|z|r|l|n|m|w|y] ;
define CodaC    [p|t|k|s|r|l|n|m|w|y] ;
define Vowel    [a|e|i|o|u] ;
define Syllable InitialC Vowel (CodaC) ;
define Baseform Syllable^{1,3} ;
define Nouns    %+Noun ([%+Aug | %+Dim])
[%+Masc | %+Fem] [%+Sg | %+Pl] ;
define Adjs     %+Adj ([%+Comp | %+Sup])
[%+Masc | %+Fem] [%+Sg | %+Pl] ;
define Verbs    %+Verb [%+Past | %+Pres | %+Fut]
[%+1P | %+2P | %+3P] [%+Sg | %+Pl] ;
define XolaLexGram   Baseform [Nouns | Adjs | Verbs] ;
```

Figure 6.13: The **xfst** Script Xola-lexgram.script Defining a Lexical Grammar for an Imaginary Language, Including Baseforms with Constraints on Syllables, Possible Initial Consonants, and Possible Coda Consonants

The definition or "modeling" of pseudo-baseforms may also be much more constrained and sophisticated, reflecting known phonological constraints in the language. For example, Figure 6.13 shows a Lexical Grammar for the mythical Xola language, in the form of an **xfst** script, wherein the baseforms consist of one to three syllables of form CV or CVC, and where there are constraints on which consonants can appear in initial and coda position.

The object of writing and maintaining a Lexical Grammar is to encourage you to define and conform to a consistent format for analysis strings; such consistency is absolutely essential to anyone, including yourself, who needs to use the Lexical Transducer to do generation.

The Lexical Grammar is also a powerful tool for testing and debugging. Assume that your Lexical Transducer is already compiled and stored in file `MyLanguage.fst` and that your Lexical Grammar is written as a regular expression in

```
xfst[n]: clear stack
xfst[0]: read regex < MyLanguage-lexgram.regex
xfst[1]: define LexGram
xfst[0]: read regex [@"MyLanguage.fst"].u - LexGram ;
xfst[1]: print size
xfst[1]: print words
```

Figure 6.14: An Idiom to Check the Upper-Side Language Using a Lexical Grammar Defined in a Regular-Expression File. Note that the Lexical Grammar is subtracted from the extracted upper-side language of `MyLanguage.fst`. If the upper-side language is fully covered by the Lexical Grammar, then the network left on The Stack will denote the empty language.

file `MyLanguage-lexgram.regex`. The idiom in Figure 6.14 subtracts the Lexical Grammar language from the upper-side language of `MyLanguage.fst`, leaving on The Stack a network containing all the upper-side strings of `MyLanguage.fst` that do not conform, for whatever reason, to the Lexical Grammar as it is defined in `MyLanguage-lexgram.regex`. If the upper-side language is fully covered by the Lexical Grammar, then the result of the subtraction will be the empty language.

```
xfst[n]: clear stack
xfst[0]: source Xola-lexgram.script
xfst[0]: read regex [@"Xola.fst"].u - XolaLexGram ;
xfst[1]: print size
xfst[1]: print words
```

Figure 6.15: Idiom to Check the Lexical Language Using a Lexical Grammar Defined in an **xfst** Script File. In this case, the script (see Figure 6.13) computes the lexical-grammar network and defines the variable XolaLexGram. This XolaLexGram is then subtracted from the upper-side language of `Xola.fst`. If all is well, the network left on The Stack will be empty.

If you have defined your Lexical Grammar as an **xfst** script, as in Figure 6.13, then the idiom shown in Figure 6.15 will perform the appropriate check. Let us assume that the lexical-grammar script is stored in `Xola-lexgram.script` and that the Lexical Transducer to be tested is in `Xola.fst`. Running the script in Figure 6.13, using the usual **source** utility in **xfst**, causes the variable XolaLex-

Gram to be defined.

However it is defined, the Lexical Grammar should produce a network that encodes a simple language, not a transducer. When that language is subtracted from the upper-side language of the Lexical Transducer, the result should be the empty language. In practice, this exercise often flushes out a set of lexical strings that do not conform, for whatever reason, to the Lexical Grammar. Any non-conforming strings should be examined manually, and any necessary corrections should be done. It may, of course, be necessary to correct your **lexc** and **xfst** files, or it may be necessary to augment and correct your Lexical Grammar, especially after you introduce new parts of speech or any changes in the tags.

The Lexical Grammar will naturally grow and need modification during development. Don't be afraid to change it as necessary, but do use it throughout development to keep your lexical-level strings as consistent and beautiful as possible. Finally, put commented examples inside the Lexical Grammar source file; this file, without modification, should serve as an important part of the final documentation of your system. Anyone wanting to use your system for generation will have to know exactly how the upper-side strings are spelled.

> Do not try to subtract in the other direction, computing the Lexical Grammar minus the upper side of your lexical transducer. Because your Lexical Grammar accepts strings based on any possible base-form, its language covers a huge or even infinite number of strings, and the difference will also be huge or infinite in size.

6.4.2 Testing the Surface Language

Just as the lexical language of a transducer can be extracted and compared against a language defined in a Lexical Grammar, the surface language of a Lexical Transducer can also be extracted and compared against other suitable languages. These comparison languages can come from external wordlists, external lexicons, and previous versions of your own system. Comparison against previous versions of your own system is known as regression testing.

Testing Against Wordlists

A wordlist—a plain list of surface words, usually extracted from a corpus—is not often of huge value in computational linguistics. But many such lists have been prepared, often to support primitive spelling checkers, and they can often be acquired free from the Internet. A plain wordlist is seldom worth buying, but where one is available, it can be used profitably to test the surface side of your Lexical Transducer.

Wordlists can be viewed as attempts, very primitive and inadequate attempts, to model a natural language by simply enumerating all the possible strings. You can automatically create your own wordlist from any corpus, simply by tokenizing it, one word to a line, and **sort**-ing it and **uniq**-ing to remove duplicates.

```
cat corpus | \
tr -sc "[:alpha:]" "[\n*]" | \
sort | uniq > wordlist
```

One way to test our Lexical Transducers on a wordlist is to simply pipe the wordlist to the **lookup** utility, which expects its input to have one token to a line. The failures can be collected and manually reviewed as usual, and your lexicons and rules can be modified, if appropriate, to accept them.

```
cat wordlist | \
lookup lexical_transducer | \
grep '+?' | gawk '{print $1}' > failures
```

However, we can get more from a wordlist, faster, by compiling it into a network and using subtraction to compare it against the surface language of our Lexical Transducer. Consider a plain wordlist that starts like this:

```
aardvark
about
apple
day
diaper
dynasty
easy
finances
grapes
grovel
...
```

We could tediously edit the wordlist into a suitable **lexc** file or even a regular expression for compilation into a network, but the need to compile a network from a plain wordlist is common enough that **xfst** offers the **read text** utility for exactly that purpose. **read text** takes the name of a wordlist file, containing one word per line, explodes the strings into separate symbols just like **lexc**, builds a network that accepts all and only the strings in the wordlist, and pushes that network on the **xfst** stack.

The first way, and sometimes the only practical way, to use the wordlist language is to subtract from it the surface language of your Lexical Transducer. An **xfst** idiom to do this is shown in Figure 6.16. We assume again that our Lexical Transducer is in file `MyLanguage.fst` and that the wordlist is in the file `wordlist.txt`.

```
xfst[n]: clear stack
xfst[0]: read text < wordlist.txt
xfst[1]: define WordList
xfst[0]: read regex WordList - [@"MyLanguage.fst"].l ;
xfst[1]: write text > words-missing.txt
```

Figure 6.16: Idiom to Subtract the Lower-Side Language of a Network from a Wordlist Language. The remaining language, which can be printed out as a wordlist file, is the set of words in the wordlist that are not analyzed by the transducer.

However it is computed, the difference language will contain a set of surface words that are in the wordlist but are not in the surface language of the Lexical Transducer. You can use the **write text** utility to write these words out to file. These words should be reviewed carefully for possible addition to your Lexical Transducer. Because publicly available wordlists come from who-knows-where and were built from who-knows-what corpora, they are notorious for containing garbage.

Depending on the size of the wordlist—the larger the better—you can also perform the reverse subtraction and get manageable results, as shown in Figure 6.17. This subtraction will yield a list of all words that are in your surface language but which were not in the wordlist. This *may* flush out some bad strings that have crept into your surface language via errors of various sorts. But if the wordlist language is too small, or if your surface language is exceptionally large (especially if you have a compounding language with an infinite number of strings), then this second test becomes impractical.

```
xfst[n]: clear stack
xfst[0]: read text < wordlist.txt
xfst[1]: define WordList
xfst[0]: read regex [@"MyLanguage.fst"].l
- WordList ;
xfst[1]: print size
xfst[1]: write text > words-extra.txt
```

Figure 6.17: Idiom to Subtract the Wordlist Language from the Surface Language of a Lexical Transducer. For this test to be practical, the lower-side language of `Mylanguage.fst` should not be infinite or too much larger than the language of `wordlist.txt`.

A corpus of running text is generally more valuable for testing than a plain wordlist. With a corpus, you can sort the not-found words by frequency of occurrence, allowing you to concentrate on fixing the most important gaps in your system; and you can search the corpus to see the context of mysterious alleged words. With a wordlist, all the not-found words are just disembodied strings of characters, with no information as to frequency or context.

Testing Against Previous Versions

Morphological analyzers get big and complex, to the point where it is humanly impossible to predict all the implications of adding, deleting and changing rules or lexical entries. Luckily, the finite-state calculus gives us powerful ways of comparing the coverage of different versions of our own systems, which is the essence of regression testing.

For testing purposes and for safety, you should keep backups and work within a version-control system (see Chapter 5). From time to time, the system will reach a relatively stable state, and at such times you should take a snapshot of the entire status of your source files. You should *always* take a snapshot when any kind of contractual milestone is reached or a delivery is made. The version-control system will then allow you to restore that state at a later time.

Here's a typical scenario: Assume that your system reached a relatively stable state, and that you took a snapshot of it as Version1. A week or a month later, you have added a few lexical items, added three rules, changed several other rules, and you wonder if you've broken anything in the process.

Assume that the new Lexical Transducer is named `new.fst`, and that you saved a copy of the previous Version1 transducer as `old.fst`. If you forgot to save a copy, use the version-control system to reconstruct it; that's what a version-control system is for. Figure 6.18 shows an **xfst** idiom for seeing which surface words have been lost going from `old.fst` to `new.fst`.

Creating the output file may not be necessary. Recall that you can use **print size** to see how many words were lost, if any. If the number is not zero, then you might want to look at a few of the words using **print random-upper** or **print random-lower**; or, if the number is very small, use **print words** to see them all displayed on the screen.

To see which surface words were added in going from `old.fst` to `new.fst`, simply do the reverse subtraction as shown in Figure 6.19.

Testing against your own previous system is often the only practical way to catch regressions (degradations) and introduced errors. After performing the subtractions, if all the words in the file `words-lost.txt` are in fact bad words, then the system has improved. And if all the words in `words-added.txt` are good

```
xfst[n]: clear stack
xfst[0]: read regex [@"old.fst"].l - [@"new.fst"].l ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > words-lost.txt
```

Figure 6.18: Regression Testing, Comparing Two Versions to Find Lost Surface Words

```
xfst[n]: clear stack
xfst[0]: read regex [@"new.fst"].l - [@"old.fst"].l ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > words-added.txt
```

Figure 6.19: Regression Testing, Comparing Two Versions to Look for Added Words

words, then again the system has improved. But if you've lost any good words, or gained any bad words, you need to fix the current files and rerun the tests until you see all improvement and no degradation. At that point, if your new system is generally stable, it is a good idea to take a new snapshot and save the current Lexical Transducer as `old.fst` for the next round of regression testing.

> Learn to use your local version control system, even if it seems like irritating overhead. The ability to resurrect the previous stable system and compare it against your new system is often the only practical way to do regression testing.

Testing Against External Lexicons

Occasionally you can buy or find externally produced LEXICONS that can also be used for testing against your system's surface language, where by lexicon we mean an on-line resource that is more informative than a bare wordlist, such as a list of baseforms with category and feature information. The more auxiliary information a lexicon contains, the more potentially valuable it is. Such externally produced lexicons can be a gold mine for testing and development.

- If you can extract or generate a list of surface words from an external lexicon, that list can be used like any other wordlist for testing against your system's surface language.
- If you can establish the equivalence between the external lexicon's categories and your own, you may be able to extract large numbers of entries from the external lexicon, edit them with macros, and add them semi-automatically to your own **lexc** lexicons.
- If the external lexicon contains part-of-speech and feature information, you should be able to construct testing scripts that check to see if your system analyzes the words and assigns equivalent tags and features.

Comparing your system against another carefully produced lexicon is one of the few effective ways to find sins of partial omission, e.g. when your system analyzes *table* as a noun but not also as a verb. Although any lexicon will have mistakes and omissions, the chances of two independently produced lexicons having the same mistakes and errors are relatively low.

```
xfst[n]: clear stack
xfst[0]: define MyLanguageNouns [ $[%+Noun]
.o.
@"MyLanguage.fst"].l  ;
xfst[0]: read text < nounlist.txt
xfst[1]: define NounList
xfst[1]: read regex NounList - MyLanguageNouns ;
xfst[1]: print size
xfst[1]: print random-lower
xfst[1]: write text > nouns-to-add.txt
```

Figure 6.20: Extracting Nouns from a Lexical Transducer for Comparison to a Wordlist of Alleged Nouns Extracted from Another Dictionary. We assume here that our Lexical Transducer marks nouns on the upper side with the tag +Noun. The subtraction of MyLanguageNouns from NounList will leave on The Stack a network of alleged nouns that are not analyzed as nouns by the transducer.

For example, a wordlist of words that are identified in the external lexicon as nouns can be compared profitably against the set of surface nouns recognized by your system. Figure 6.20 shows the idiom for extracting the set of surface nouns (as a language) from your system, which we'll assume is stored in `MyLanguage.fst`. Let's also assume that the list of alleged nouns extracted somehow from the external lexicon is called `nounlist.txt`. The idiom effectively identifies a set of words, alleged to be nouns in another dictionary, that are not analyzed

as nouns by your Lexical Transducer. The left-over words should be considered carefully for addition, as nouns, to your lexicons.

If the noun wordlist is comparable in size to the noun coverage of your system, then the reverse subtraction may also be possible and interesting, identifying alleged nouns analyzed by your system that are not included in the external list of nouns. This is the way to find if your transducer is generating any ill-formed nouns.

Typically, both the external wordlist and your analyzer will be in error in various ways, but the probability that both will err in exactly the same way is low. This is what makes cross-wordlist or cross-lexicon testing so valuable; it subtracts one haystack from another, leaving relatively little hay and a manageable assortment of needles to examine.

The more finely an external lexicon is coded, the better it is for testing against our finite-state systems. Two separate Spanish noun lists, for example, one of feminine nouns and one of masculine nouns, are more valuable than one list of undifferentiated Spanish nouns. The testing of a list of feminine Spanish nouns against a full Spanish lexical transducer can be done exactly like the MyLanguageNouns test in Figure 6.20, except that one starts by extracting just the feminine nouns from `Spanish.fst`, as in Figure 6.21. Simple comparison, via subtraction, of what your system calls feminine Spanish nouns from what another lexicon calls feminine Spanish nouns can easily highlight gaps and mis-codings.

```
xfst[n]: clear stack
xfst[1]: define FemNounList [ $[%+Noun ?* %+Fem]
.o.
@"MyLanguage.fst"].l ;
```

Figure 6.21: Extracting Feminine Nouns Only. This example assumes that the tags are on the upper side and that the `+Noun` tags precede the `+Fem` tags.

A decent lexicon with precise subcodings can often be split automatically, e.g. with Perl scripts, into dozens of separate wordlists for comparison against the words and analyses in your own system. Acquire any good lexicons and labeled wordlists that you can, everything from lists of people, cities and companies to full-scale lexicons from other NLP projects. A list of German compound nouns is valuable; a list of Spanish verb infinitives, with each verb marked for its conjugation class, is a gift; and a detailed lexicon from another serious natural-language-processing project can be priceless. A carefully constructed lexicon of baseforms, with detailed marking of inflectional possibilities, conjugation types and derivational possibilities can often be converted semi-automatically into real **lexc** format. Such lexicons, usually jealously guarded by their owners, can be extremely valu-

able for accelerating a development project, saving you person-years of tedious lexicographical development.

Unfortunately, lexicons vary tremendously in quality, coverage, format and terminology; and they are notoriously difficult to acquire for commercial purposes. Each case is unique, usually requiring some detective work to interpret and find the information you need, and you may need to depend heavily on local computer gurus to write conversion scripts. But while it is never easy, "mining" an independently produced lexicon of good quality is one of the most valuable exercises you can perform for testing.