Chapter 3

The xfst Interface

3.1 Introduction

3.1.1 What is xfst?

This chapter is a hands-on tutorial on **xfst**, a general-purpose interactive utility for creating and manipulating finite-state networks. You should have the **xfst** application installed and running in front of you as you work your way through the examples and exercises. We will review the various regular-expression notations, already introduced in Chapter 2, while learning about the **xfst** interface and its various commands.

xfst provides a compiler for regular expressions, creating new finite-state networks according to your specifications. You can also, from within **xfst**, invoke the **lexc** compiler on **lexc** source files (to be presented in Chapter 4), and there are auxiliary compilers for compiling wordlists and other text-like formats. Where necessary, **xfst** gives you direct access to the algorithms of the Finite-State Calculus, including concatenation, union, intersection, composition, complementation, etc.

3.1.2 Getting Started

We introduce **xfst** with a brief tour that shows how the application is launched, queried and exited in a UNIX-like operating system.

Invoking xfst

The basic way to invoke **xfst** is simply to enter `xfst` at the operating system's command-line prompt. "Entering" means typing the command followed by the Enter or Return key on your keyboard.

```
unix> xfst
```

```
usage (alias: compose):
      compose net

replaces the stack with the composition of
all networks currently on the stack. If the
stack consists of two networks, defined as
A and B, with A on top of the stack, the
result is equivalent to compiling the regular
expression [A .o. B].
```

Figure 3.1: **xfst** Help Message for Command **compose net**

xfst will respond with a welcome banner and an **xfst** prompt, and it then waits for you to enter an **xfst** command after the prompt.

```
Copyright © Xerox Corporation 1997-2002
Xerox Finite-State Tool, version 7.9.0

Enter "help" to list all commands available
or "help help" for further help.

xfst[0]:
```

The zero in the prompt (`xfst[0]`) will be explained later, as will command-line flag options available when invoking **xfst**.

Help

If you enter `help` or a question mark (`?`) at the **xfst** prompt, **xfst** will display a long menu of available commands (Table 3.1), organized into subclasses. Don't be intimidated by the wealth of choices; you will need only a few commands to get started.

```
xfst[0]: help
```

For short documentation on a particular command, e.g. **compose net**, enter `help` followed by the name of that command.

```
xfst[0]: help compose net
```

xfst will respond with a useful summary (Figure 3.1) of the usage and semantics of the command.

```
COMMAND CLASS: System commands

alias, quit, set, show, source, system

COMMAND CLASS: Input/output and stack commands

clear stack, define, list, load defined, load stack, pop stack,
push defined, read lexc, read prolog, read regex,
read spaced-text, read text, rotate stack, save defined,
save stack, turn stack, undefine, unlist, write prolog,
write spaced-text, write text,


COMMAND CLASS: Display commands

apply up, apply down, apropos, echo, help, inspect net,
print aliases, print defined, print directory,
print file-info, print flags, print labels,
print label-tally, print lists, print longest-string,
print longest-string-size, print lower-words, print name,
print net, print random-lower,print random-upper,
print random-words, print sigma, print sigma-tally,
print sigma-word-tally, print size, print stack,
print upper-words, print words, write properties

COMMAND CLASS: Tests of network properties

test equivalent, test lower-bounded, test lower-universal,
test non-null, test null, test overlap, test sublanguage,
test upper-bounded, test upper-universal

COMMAND CLASS: Operations on networks

add properties, cleanup net, compact sigma,
compile-replace lower, compile-replace upper, complete net,
compose net, concatenate net, crossproduct net,
determinize net, edit properties, eliminate flag,
epsilon-remove net, intersect net, invert net, label net,
lower-side net, minimize net, minus net, name net, negate net,
one-plus net, prune net, read properties, reverse net,
shuffle net, sigma net, sort net, substitute defined,
substitute label, substitute symbol, sub-
string net, union net,
upper-side net, zero-plus net
```

Table 3.1: The Command Menu from **xfst**

```
compose net :
    compose the finite-state networks in the stack
regex : compose net
    regular expression [A .o. B]
```

Figure 3.2: Response from **apropos compose**

Apropos

If you don't know the exact format of the command you need, the **apropos** command can be very useful. For example to find information about commands that involve the compose operation, enter `apropos compose` and **xfst** will return a list of relevant commands and switches as shown in Figure 3.2.

```
xfst[0]: apropos compose
```

The Read-Eval-Print Loop

The **xfst** interface is a read-eval-print loop; it reads your command typed manually at the prompt, executes it, and then displays a response and a new prompt. As we shall see later, it is also possible to direct **xfst** to perform a series of commands that were previously edited and stored in a file called a SCRIPT (see Section 3.3.3).

Exiting from xfst

To exit from the **xfst** interface and return to the host operating system, simply invoke the **exit** command or the **quit** command.

```
xfst[0]: exit
```

```
xfst[0]: quit
```

3.2 Compiling Regular Expressions

3.2.1 xfst Regular Expressions

xfst includes a compiler for the **Xerox** regular-expression metalanguage, which includes many useful operators that are not available in other implementations. Traditional textbook notations have been modified where necessary to allow them to be typed as linear strings of ASCII characters. Readers who are already acquainted with regular expressions and want to get started quickly should review the syntax already presented in Section 2.3.1 and consult the online regular-expression summaries available from `http://www.fsmbook.com/`.

3.2.2 Basic Regular-Expression Compilation and Testing

This section is intended for those who are already fairly comfortable with the **Xerox** regular-expression notation and want to jump into compilation and testing. Later we will review, more slowly and incrementally, the various regular-expression operators and **xfst** commands available.

define

There are two **xfst** commands, **define** and **read regex**, that invoke the regular-expression compiler. The schema for the **define** command is the following:

```
xfst[0]: define variable regular-expression ;
```

The variable name is chosen by the user. The regular expression must start on the same line, but it can be arbitrarily complicated and can extend over multiple lines. The regular expression must be terminated with a semicolon followed by a newline.

The effect of such a command, e.g.

```
xfst[0]: define MyVar [ d o g | c a t | h o r s e ] ;
```

is to invoke the compiler on the regular expression, create a network, and assign that network to the indicated variable, here **MyVar**. Once defined in this way, the variable is stored in a symbol table and can be used in subsequent regular expressions.

> The **define** command in fact comes in two easily confusable variants. When you call **define** to compile a regular expression, as in the examples just shown, that regular expression must begin on the same line as the **define** command. The other variant of the **define** command will be shown below.

read regex

The other compiler command, **read regex**, similarly reads and compiles a regular expression, but the resulting network is added or PUSHED onto a built-in STACK.

```
xfst[0]: read regex regular-expression ;
```

We will have much more to say about The Stack below, but for now it suffices to know that many **xfst** operations refer by default to the top network on The Stack, or they POP their arguments from The Stack and push the result back onto The Stack.

The **xfst** prompt includes a counter indicating how many networks are currently stored on The Stack.

If we perform the following command:

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
10 states, 11 arcs, 3 paths.
xfst[1]:
```

and have not made any mistakes in typing the regular expression, **xfst** will respond with a message indicating the number of states, arcs and paths in the resulting network. The digit **1** printed in the new prompt indicates that there is one network saved on The Stack.

> Because they can extend over multiple lines, regular expressions inside **define** and **read regex** commands must be explicitly terminated with a semicolon followed by a newline. Other commands not containing regular expressions cannot extend over multiple lines and are not terminated with a semicolon.

Basic Network Testing

Once we have, on the top of The Stack, a network that encodes a language, and if the language is finite, we can then use the **print words** command to have the language of the network enumerated on the terminal.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
10 states, 11 arcs, 3 paths.
xfst[1]: print words
dog
cat
horse
```

Other useful commands are **apply up**, which performs ANALYSIS or LOOKUP, and **apply down**, which performs GENERATION, also known as LOOKDOWN; both **apply up** and **apply down** use the network on the top of The Stack. In the formal terminology of networks, **apply up** "applies the network in an upward direction" to an input string, matching the input against the lower side of the network and returning any related strings on the upper side of the network. The following example looks up the word "dog" and returns "dog" as the response, indicating success. Note that it is not necessary to recompile the regular expression if the network is already compiled and on the top of The Stack.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: apply up dog
dog
xfst[1]:
```

Any attempt to look up a word not in the language encoded by the network will result in failure, and **xfst** will simply display a new prompt.

```
xfst[1]: apply up elephant
xfst[1]:
```

If you are hand-testing a number of input words, you may soon tire of retyping **apply up** over and over again. It happens that the **apply up** command, like most commands in **xfst**, has an abbreviation, in this case **up**.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: up dog
dog
xfst[1]:
```

In addition, there is an APPLY UP MODE that you invoke when you enter **apply up**, or just **up**, without an argument. The new prompt becomes `apply up>`, and you simply enter input strings one at a time at the prompt. To exit this apply-up mode and return to the **xfst** prompt, enter `END;`, all in capital (upper-case) letters, and including the semicolon. On a Unix-like system, you can also escape apply-up mode by entering Control-D. In a Windows system, you can type Control-Z.

```
xfst[0]: read regex [ d o g | c a t | h o r s e ] ;
xfst[1]: apply up
apply up> dog
dog
apply up> cat
cat
apply up> elephant
apply up> END;
xfst[1]:
```

Finally, if you have a list of input words stored in a file, with one word per line,[1] you can analyze the whole file by entering **apply up**, followed by a left angle bracket (<) and the name of the file (or, in general, the pathname of the file). As usual, **apply up** will apply the network on the top of The Stack.

```
xfst[1]:  apply up < filename
```

Basic Pushing and Popping

There are a number of commands for manipulating The Stack itself, but for now we'll introduce just two. Use the **pop stack** command to pop the top network off The Stack and discard it. The counter in the prompt will decrement by one.

```
xfst[1]: pop stack
xfst[0]:
```

[1]Each whole line in the file, including any spaces, is treated as a word or token to be analyzed. Extraneous spaces on the beginning or end of tokens will cause lookup to fail.

Use the **clear stack** command to pop all the networks off of The Stack, leaving it empty. The counter in the new prompt will be 0.

```
xfst[4]: clear stack
xfst[0]:
```

> Be sure to distinguish commands from regular expressions. You type commands, including the **help**, **apropos**, **define**, **read regex**, **pop stack**, **clear stack**, **apply up** and **apply down** commands, to the **xfst** interface. Only two of these commands, **define** and **read regex**, can *include* a regular expression as part of the command. You must therefore learn the language of commands for running **xfst** and the metalanguage of regular expressions, which has its own syntax.

3.2.3 Basic Regular Expressions Denoting Languages

Through the rest of this chapter we will review the regular-expression metalanguage progressively, presenting examples and introducing useful **xfst** commands as we go. We start with the basic SYMBOL notations shown in Table 3.2. Every basic alphabetic symbol is, by itself, a valid regular expression. Case is significant everywhere in **xfst**, so **z** is a separate symbol from **Z**.

The simplest regular expression is therefore just a symbol like `a`, which by itself denotes the language consisting of the single string "a". You compile such an expression using the **read regex** command or the **define** command. Like all regular expressions in **xfst**, it must be terminated with a semicolon and a newline.

```
xfst[0]: clear stack
xfst[0]: read regex  a ;
2 states, 1 arc, 1 path.
xfst[1]: print words
a
```

xfst responds with a short message indicating the number of bytes, states, arcs and paths in the resulting network, and then displays a new prompt. The **1** in the new prompt indicates that there is one network stored on The Stack.

In addition to basic symbols, **xfst** regular expressions also allow MULTICHARACTER SYMBOLS, which have multicharacter print names but which are stored and manipulated in networks exactly like simple alphabetic symbols. Multicharacter symbols are not declared in **xfst** but are simply written in regular expressions with no spaces between the letters of the name, as shown in Table 3.3. Any multicharacter symbol is, by itself, a valid regular expression.

a	Alphabetic characters like a, b, c, etc.
"a"	A double-quoted normal alphabetic letter like "a" is equivalent to a.
"+"	A literal plus sign symbol; i.e. not a Kleene star.
%+	A literal plus sign; alternate notation.
"\o" "\oo" "\ooo"	A symbol expressed as an octal value, where **o** is a digit 0-7, e.g. "\123".
"\xHH"	A symbol expressed as a hexadecimal (hex) value, where **H** is 0-9, a-f or A-F, e.g. "\xA3".
"\uHHHH"	A 16-bit Unicode character, e.g. "\u0633".
"\n"	Newline symbol in escape notation as per Unix convention. Also \t (tab), \b (backspace), \r (carriage return), \f (formfeed), \v (vertical tab), and \a (alert).

Table 3.2: Regular-Expression Notations for Basic Symbols

cat	Compiled as the single multicharacter symbol **cat**. Users are advised *not* to define multicharacter symbols like **cat** because they are too easy to confuse with the concatenation of three separate symbols **c**, **a** and **t**.
"+Noun"	Compiled as the single multicharacter symbol +Noun. The surrounding double quotes cause the plus sign to be treated as a literal letter, i.e. not the Kleene plus. The use of one or more punctuation symbols in the spelling of a multicharacter symbol is a recommended convention.
%+Noun	Another way to notate the multicharacter symbol +Noun. The percent sign literalizes the following plus sign.
%^HIGH	The multicharacter symbol ^HIGH, starting with a literal circumflex.
%[HIGH%]	The multicharacter symbol [HIGH], starting with a literal left square bracket and ending with a literal right square bracket.
"[HIGH]"	Another way to notate the multicharacter symbol [HIGH].

Table 3.3: Notations for Multicharacter Symbols

A trivial regular expression consisting of a single multicharacter symbol is shown here:

```
xfst[0]: read regex "+Noun" ;
2 states, 1 arc, 1 path.
xfst[1]: print words
+Noun
```

?	Denotes ANY symbol.
0	Denotes the empty (zero-length) string, also called epsilon.
[]	Denotes the empty string; equivalent to 0.

Table 3.4: Special Symbol-Like Notations

Three special symbol-like notations are listed in Table 3.4; we will encounter more later when dealing with restrictions and replace rules. **xfst** uses `0` and `[]` to denote the empty string because the traditionally used epsilon symbol (ϵ) is not available on ASCII keyboards. In regular expressions the question mark denotes any single symbol, including all possible non-alphabetic and multicharacter symbols.

Note that the backslash notations for symbols, e.g. **\142**, **\x63** and **\u0633**, and the special control characters like **\n** and **\t**, can appear only inside double-quoted strings.

The finite-state grouping, iteration and optionality operators are shown in Table 3.5. In this table, A represents an arbitrarily complex regular expression. Consider the regular expression `a+`, which denotes the infinite language of all strings consisting of one or more **a**s: "a", "aa", "aaa", "aaaa", etc. Such an expression can be compiled and tested like any other; **apply up** will succeed for strings like "aa" and "aaaaaaa" but fail for any string that is not composed exclusively of one or more **a** symbols.

```
xfst[0]: read regex  a+ ;
xfst[1]: apply up a
a
xfst[1]: apply up aa
aa
xfst[1]: apply up aaaaaaaaaa
aaaaaaaaaa
xfst[1]: apply up aaaab
xfst[1]:
```

`[]`	Grouping brackets. `[A]` is equivalent to `A`.
`A*`	The Kleene star. Denotes zero or more concatenations of `A` with itself.
`A+`	The Kleene plus. Denotes one or more concatenations of `A` with itself. `A+` is equivalent to `[A A*]`.
`(A)`	Optional. `(A)` is equivalent to `[A|0]`.
`A^n`	Where *n* is an integer, denotes n concatenations of `A` with itself.
`A^{n,m}`	Where *n* and *m* are integers, denotes n to m concatenations of `A` with itself.
`A^<n`	Where *n* is an integer, denotes fewer than n concatenations of `A` with itself.
`A^>n`	Where *n* is an integer, denotes greater than n concatenations of `A` with itself.

Table 3.5: Grouping, Iteration and Optionality. In this table, `A` represents an arbitrarily complex regular expression.

The language denoted by `a*` includes all the words in `a+` plus the empty string (epsilon). Optionality is indicated by surrounding an expression with parentheses, as in `(a)`. The numbered iteration operators have many flavors, the simplest being exemplified by `a^5`, which denotes the language consisting of the single string "`aaaaa`".

> Do not confuse square brackets, used for grouping, with parentheses, which denote optionality.

The basic traditional finite-state operations include union, intersection, minus (subtraction), complementation (negation) and concatenation. Table 3.6 shows how these are notated in **xfst** regular expressions. In the table, `A` and `B` denote arbitrarily complex regular expressions.

Note in particular the operation of concatenation, which has no explicit operator. When symbols like **a**, **b**, **c** and **+Noun** are to be concatenated together, they should be typed one after another, separated by whitespace. The expression `[c a t]`, for example, denotes the string "`cat`", which consists of the three separate symbols **c**, **a**, and **t**.

Compile and test, using **apply up**, the regular expression `[a b]*`, which denotes the infinite language of strings containing zero or more concatenations of "`ab`". Don't neglect to put a space between the **a** and the **b** in the regular expres-

`A	B`	The union of A and B.
`A B`	The concatenation of B after A. The operands are separated by white space; there is no explicit concatenation operator.
`{cat}`	Compiled as a concatenation of the three symbols **c**, **a** and **t**. Surrounding a string of symbols with curly braces "explodes" them into separate symbols, here equivalent to `[c a t]`.
`A & B`	The intersection of A and B. Here A and B must denote languages, not relations.
`A - B`	The subtraction of B from A. Here A and B must denote languages, not relations.
`~A`	The language complement of A, i.e. `[?* - A]`. Here A must denote a language, not a relation.

Table 3.6: Regular-Expression Notations for Basic Finite-State Operations. In this table, A and B stand for arbitrarily complex regular expressions.

sion or you will inadvertently be defining a new multicharacter symbol **ab**, which is not what you want here.

Spacing is required when concatenating *symbols*, to avoid implicit declaration of unwanted multicharacter symbols. When one or both of the operands to be concatenated are themselves delimited, then no separating white space is necessary; for example

```
[l o o k]  [i n g]
```

and

```
[l o o k][i n g]
```

are equivalent expressions, both denoting the concatenation of the language containing the string "`look`" with the language containing the string "`ing`", to denote a new language containing the string "`looking`". Similarly, no spacing is necessary in the following union expression, because the vertical bars themselves separate the symbols being unioned.

```
[a|e|i|o|u]
```

The curly braces in expressions like `{dog}` are sometimes called "explosion" braces because they tell the compiler to "explode" the contained string into separate symbols rather than treating them as the name of a multicharacter symbol; the regular expression `{dog}` is equivalent to `[d o g]`. In practice it is often

more convenient to type and read {elephant} than [e l e p h a n t], especially if there are many such strings to type.

> Warning: It is all too easy to inadvertently write a multicharacter symbol, e.g. cat, in a regular expression when you really intended to write c a t or {cat}, indicating the concatenation of three separate symbols **c**, **a** and **t**. By **Xerox** convention, multicharacter symbols include a non-alphabetic character like the plus sign, e.g. +Noun, or the circumflex, e.g. ^HIGH, or they surround a name in punctuation, e.g. [Noun] or [Verb], to help them stand out and to avoid visual confusion with simple concatenations of alphabetic characters. Use the command **print sigma** to spot any unwanted multicharacter symbols in the alphabet.

From these regular-expression notations and others to come, you will note that almost all the ASCII punctuation characters have been pressed into service as operators. The punctuation characters are therefore special characters in **xfst**. To notate literal plus signs, asterisks, vertical bars, etc. they must be preceded with the literalizing percent sign, as in %+, %* and %|, or included inside double quotes, as in "+", "*" and "|".

Table 3.7 illustrates expressions that denote the universal language and the empty language. The regular expression ?* denotes the universal language, which contains all possible strings, of any length, including the empty string. The empty language, which contains no strings at all, not even the empty string, can be notated many ways, including ~[?*] (i.e. the complement of the universal language), [a - a], [b - b], etc.

?*	Denotes the universal language, which contains all possible strings, including the empty string (epsilon).
~[?*]	One of an infinite number of ways to denote the empty language, which contains no strings at all.
[a - a]	Another way to denote the empty language.

Table 3.7: The Universal Language and the Empty Language

The union operator | constructs a regular language or relation that contains all the strings (or ordered string pairs) of the operands; e.g. a | b denotes the language that contains the string "a" and "b". Square brackets can be used freely to group expressions, e.g. [c a t | d o g] - [d o g], either to make expressions easier for people to read or to force the compiler to perform certain

operations before others. By convention, concatenation has higher precedence than union. The precedence of the finite-state operators is described below on page 202.

Using these operators, which are already familiar to most programmers, we can start to write more interesting regular expressions and experiment with the compiled networks. Follow along with the following examples.

```
xfst[0]: read regex  d o g | c a t | h o r s e ;
10 states, 11 arcs, 3 paths.
xfst[1]:
```

This regular expression denotes a language containing just the three words "dog", "cat" and "horse". Once the corresponding network is compiled and pushed on the top of The Stack, we can cause **xfst** to enumerate the words of the language with the command **print words**.

```
xfst[1]:  print words
dog
cat
horse
xfst[1]:
```

The following expression subtracts one language of strings from another. Compile it and use **print words** to see the words in the resulting language.

```
xfst[0]:  read regex [ d o g | c a t | r a t |
e l e p h a n t ] - [ d o g | r a t ] ;
10 states, 10 arcs, 2 paths.
xfst[1]: print words
cat
elephant
```

Also compile and test the expression

```
(r e)[[m a k e] | [c o m p i l e]]
```

which denotes a language containing the words "make", "remake", "compile" and "recompile". Remember to separate the symbols with spaces to avoid creating unwanted multicharacter symbols.

Similarly, use **read regex** to compile the following regular expression, which denotes an intersection of two small languages, and then enumerate the words of the language using **print words**. In the **read regex** command, remember to terminate the regular expression with a semicolon and a newline.

```
[ d o g | c a t | r a t | e l e p h a n t | p i g ]
& [ c a t | a a r d v a r k | p i g ]
```

You should find that the resulting language contains just two words, "cat" and "pig". If you get tired of spacing out all the letters, try the alternative notation using curly braces.

```
[ {dog} | {cat} | {rat} | {elephant} | {pig} ]
& [ {cat} | {aardvark} | {pig} ]
```

> Try to remember *not* to type just plain dog, without curly braces, if you really intend [d o g] or {dog}. Plain dog will be compiled as a single multicharacter symbol, which is quite different from a concatenation of three separate symbols. Unwanted multicharacter symbols can come back to haunt you later with mysterious problems.

The intersection operator & operates only on networks encoding languages (not on transducers) and constructs a language that contains all the strings common to the operands.

~A	The language complement of A, i.e. [?* - A]. Here A must denote a language, not a relation.
\A	The symbol complement of A, i.e. [? - A]. Thus \a contains "b", "c", "d", etc., but not "a" or the empty string or any string of length greater than one.
\?	Another notation for the null language.

Table 3.8: Language Complement vs. Symbol Complement

The two complement operators, shown in Table 3.8, deserve some special attention because learners very often confuse them. Where A is an arbitrarily complicated regular expression denoting a language, the *language* complement, e.g. ~A, denotes the universal language (all possible strings, of any length) minus all the strings in A; ~A is therefore equivalent to [?* - A]. If language A does not contain the empty string, then ~A will contain the empty string, as well as all other possible strings not in A.

In contrast, the *symbol* complement, e.g. \A, denotes the language of all *single-symbol* strings, minus the strings in A. The most typical uses of the symbol complement operator involve a language A which denotes either a single-symbol string or a union of single-symbol strings. Thus we typically see practical examples like \z, which denotes the language of all single-symbol strings "a", "b", "c", "d", etc., except for the string "z"; or \[a|b|c], which denotes the language of all single-symbol strings except for the strings "a", "b" and "c".[2] From the matching

[2]Programmers familiar with regular expressions in **Perl** and **emacs** may compare the **xfst** \z notation to the equivalent **Perl/emacs** [^z] notation, and \[a|b|c] to the **Perl/emacs** [^abc].

point of view, an expression like \a must match a single symbol and can never match the empty string or a string of length greater than one.

The semantics and compilation of the complement operators are discussed in detail in Sections 2.3.1 and 2.3.4 and will not be repeated here. If A denotes a finite language, then ~A denotes an infinite language that cannot be enumerated. **xfst** calls such networks "Circular", indicating that while they contain a finite number of states and arcs, the arcs include loops that result in the network containing an infinite number of possible paths.

```
xfst[0]: read regex ~[c a t] ;
5 states, 20 arcs, Circular.
xfst[1]:
```

In this example, the language being denoted is the complement of the language containing the single word "cat", i.e. the universal language minus the language containing just "cat". This is therefore an infinite language, and the network that encodes it is circular. The **print words** command will print out a few examples but has special logic that keeps it from trying to display an infinite number of words. The **print random-words** command will print a random selection of words from the language.

More useful in such cases is the **apply up** command that looks up (analyzes) an input word using the network on top of The Stack. In the case of ~[c a t], **apply up** will fail for the input word "cat" and succeed for all other words, including "cats", "dog", "hippopotamus", "monkey" and "wihuiehguwe". When **apply up** succeeds, one or more strings are returned; when **apply up** fails, it returns nothing, and **xfst** simply displays a new prompt.

```
xfst[0]: read regex ~[c a t] ;
5 states, 20 arcs, Circular.
xfst[1]: apply up cat
xfst[1]: apply up cats
cats
xfst[1]: apply up dog
dog
xfst[1]: apply up hippopotamus
hippopotamus
xfst[1]: apply up monkey
monkey
xfst[1]: apply up wihuiehguwe
wihuiehguwe
```

The CONTAIN and IGNORE operations, defined in Section 2.3.1, are summarized in Table 3.9. Try compiling and testing the regular expression $a, which denotes the language of all strings that contain at least one **a**, including "a", "ab", "zzzzza", "alphabet", etc. Also compile and test a+/b, which denotes the language of all strings consisting of one or more **a**s, i.e. "a", "aa", "aaa", etc., plus all these strings with any number of **b** symbols appearing as noise: "ab", "bbbab", "bababab", "aabbbbab", etc.

`$A`	Denotes the language of all strings (or the relation of all ordered string pairs) that contain `A`, i.e. `[?* A ?*]`.
`$.A`	Denotes the language of all strings that contain exactly one occurrence of a string from `A`.
`$?A`	Denotes the language of all strings that contain at most one occurrence of a string from `A`.
`A/B`	Denotes the language `A`, ignoring interspersed "noise" strings from `B`. In other words, `A/B` denotes the set of all strings that would be in `A` if all substrings from `B`, considered as a kind of "noise", were removed.
`A./.B`	Denotes the language `A`, ignoring *internally* interspersed "noise" strings from `B`.

Table 3.9: Contain and Ignore

3.2.4 Basic Stack Operations

Understanding The Stack

Before continuing with our review of regular-expression operators, let's look more closely at the built-in stack used by **xfst** to store and manipulate networks. The Stack is a last-in, first-out (LIFO) data structure that stores networks, and when **xfst** is launched, The Stack is empty. Networks can be PUSHED onto the top of The Stack, where they are added on top of any previously pushed networks. Networks can also be POPPED or taken off the top of The Stack.

> Last-in, first-out (LIFO) stacks are often compared to the spring-loaded pop-up stacks of plates found in many cafeterias. When you take a plate, you take it from the very top of The Stack; and any newly washed plates are added or "pushed" back onto The Stack from the top. Plates in the middle or at the bottom of The Stack are inaccessible until the plates above them have been "popped" off.

Many **xfst** commands refer to The Stack in some way, usually popping their arguments one at a time from the top of The Stack, computing a result, and then pushing the result back onto The Stack. Other commands, like **print words**, **print net**, **apply up** and **apply down**, refer by default to the topmost network on The Stack without actually popping or modifying it. Users of **xfst** must constantly be aware of what is on The Stack. Luckily, there are a number of commands to help visualize and manipulate The Stack.

Pushing and Popping

We have already seen that the **read regex** command reads a regular expression, compiles it into a finite-state network, and pushes the result onto The Stack. For the user's information, the number in the **xfst** prompt indicates at each step how many networks are currently on The Stack. In the following example session, three networks are compiled and pushed successively on The Stack.

```
xfst[0]: clear stack
xfst[0]: read regex a ;
2 states, 1 arc, 1 path.
xfst[1]: read regex c a t ;
4 states, 3 arcs, 1 path.
xfst[2]: read regex [ {dog} | {cat} | {elephant} ] ;
12 states, 13 arcs, 3 paths.
xfst[3]:
```

In this case, the resulting stack, pictured in Figure 3.3, will have the last-pushed network, the one corresponding to `[ {dog} | {cat} | {elephant} ]`, on the top, with the network for `{cat}` underneath it, and the network for `a` on the very bottom. Try typing in the same commands yourself, recreating The Stack state shown in Figure 3.3. Continuing the same session, if we invoke **print words** it will refer by default to the top network on The Stack, the last network to be pushed. The other networks below the top are temporarily inaccessible.

```
xfst[3]: print words
elephant
dog
cat
```

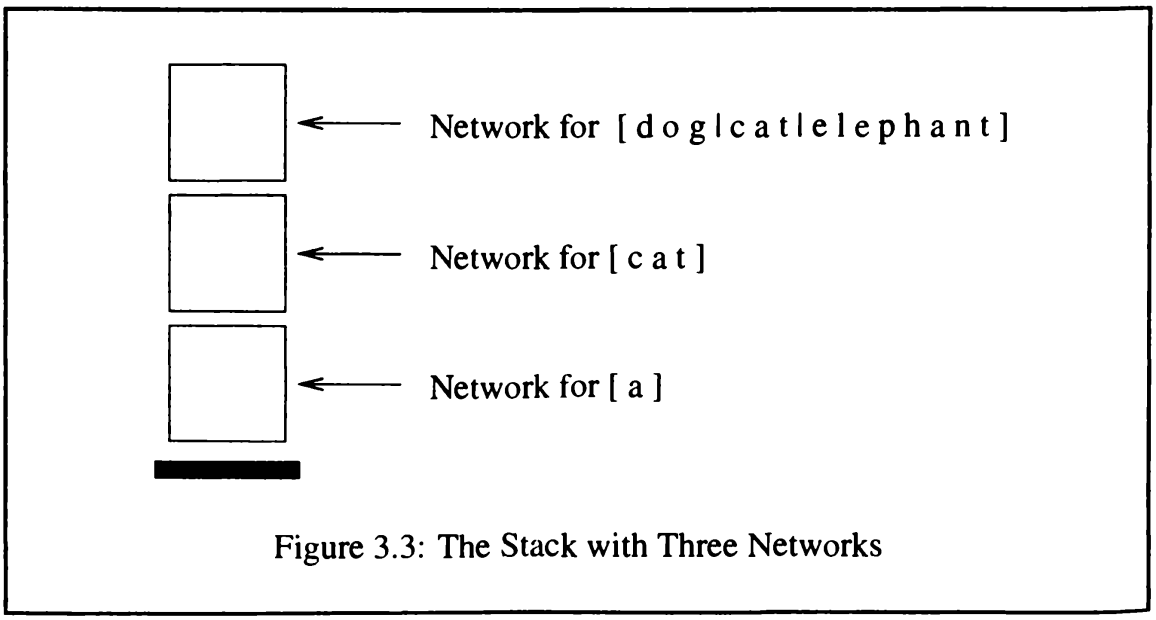


Figure 3.3: The Stack with Three Networks

To see what is on The Stack at any time, invoke the **print stack** command:

```
xfst[3]: print stack
0: 12 states, 13 arcs, 3 paths.
1: 4 states, 3 arcs, 1 path.
2: 2 states, 1 arc, 1 path.
xfst[3]:
```

print stack displays the size of each network on The Stack, starting at the top network, which it numbers 0.

To see detailed information about a particular network, including its sigma alphabet (see Section 3.2.4) and a listing of states and arcs, use the **print net** command. If it is invoked without an argument, **print net** by default displays information about the top network on The Stack.

```
xfst[3]:  print net
Sigma: a c d e g h l n o p t
Size: 11
Net: EE3E0
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   c -> s1, d -> s2, e -> s3.
s1:   a -> s4.
s2:   o -> s5.
s3:   l -> s6.
s4:   t -> fs7.
s5:   g -> fs7.
s6:   e -> s8.
fs7:  (no arcs)
s8:   p -> s9.
s9:   h -> s10.
s10:  a -> s11.
s11:  n -> s4.
```

In this display, the notation `s0` indicates the non-final start state that is numbered 0. A notation like `fs7`, with an initial **f**, indicates a final state, numbered 7. The notation `s3:  l -> s6` indicates that there is an arc labeled **l** from state 3 to state 6. The sigma alphabet is a list of all the single symbols that occur either on the upper or lower side of arcs. The arity is 1 for networks encoding simple languages (or identity relations on such languages) and 2 for networks (transducers) encoding non-identity relations.

If you call the **define** command and set a variable to a network value, you can also indicate that variable as an argument to **print net** or **print words**. Try the following:

```
xfst[3]: define XXX ( r e ) [ l o c k | c o r k ]
[ i n g | e d | s | 0 ] ;
13 states, 17 arcs, 16 paths.
xfst[3]: print net XXX
```

```
Sigma: c d e g i k l n o r s
Size: 11
Net: EE438
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   c -> s1, l -> s2, r -> s3.
s1:   o -> s4.
s2:   o -> s5.
s3:   e -> s6.
s4:   r -> s7.
s5:   c -> s7.
s6:   c -> s1, l -> s2.
s7:   k -> fs8.
fs8:  e -> s9, i -> s10, s -> fs11.
s9:   d -> fs11.
s10:  n -> s12.
fs11: (no arcs)
s12:  g -> fs11.
xfst[3]: print words XXX
lock
locking
locks
locked
cork
corking
corks
corked
relock
relocking
relocks
relocked
recork
recorking
recorks
recorked
```

To pop the top network off of The Stack and discard it, use the **pop stack** command. The counter in the prompt will decrement by one, and the next network will then become the new top network, becoming visible to commands like **print words** and **print net**.

```
xfst[3]: pop stack
xfst[2]: print words
cat
xfst[2]: print net
Sigma: a c t
Size: 3
Net: EE388
```

```
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   c -> s1.
s1:   a -> s2.
s2:   t -> fs3.
fs3:  (no arcs)
xfst[2]:
```

To pop all of the networks off of The Stack, leaving it completely empty, use the **clear stack** command. The counter in the new **xfst** prompt will then be zero.

```
xfst[2]: clear stack
xfst[0]:
```

> Play with The Stack until you become comfortable with it. Formally speaking, The Stack of **xfst** is a LIFO (Last-In, First Out) storage mechanism that is familiar to most computer scientists but a bit of a challenge to those from other backgrounds. Failure to understand the operation of The Stack is a common source of confusion and frustration when learning and using **xfst**.

Pushing and Popping Defined Variables

We have already introduced the **define** command in one variant that calls the regular-expression compiler. For example,

```
xfst[0]: define Var [ {dog} | {cat} | {snake} ] ;
```

compiles the indicated regular expression and stores the network value in the indicated variable, here Var, without affecting The Stack. The other variant of the **define** command, however, does not include a regular expression (or a terminating semicolon); instead it pops the top network off of The Stack and assigns that value to the indicated variable. The following sequence of operations is equivalent, in the end, to the **define** statement just above.

```
xfst[0]: read regex [ {dog} | {cat} | {snake} ] ;
xfst[1]: define Var
xfst[0]:
```

Note that the counter in the prompt decrements after the call to **define**, which pops The Stack to get the value to assign to `Var`.

> **xfst** commands that end with regular expressions, like **read regex** and some **define** commands, must have those regular expressions terminated with a semicolon and a newline. **xfst** commands that do not contain a regular expression are not terminated with a semicolon.

When a **define** command includes a regular expression to be compiled (see page 85), that regular expression must begin on the same line as the command name; otherwise **xfst** cannot distinguish it from the **define** variant that pops its argument off of The Stack. It is a very common error to write

```
xfst[0]: define myvariable
[ d o g | c a t | b i r d ] ;
```

expecting the regular expression to be compiled and the variable to be set to the network value. To invoke the regular expression compiler with **define**, at least one character of the regular expression to be compiled must appear on the same line as the **define** statement, e.g.

```
xfst[0]: define myvariable [
d o g | c a t | b i r d ] ;
```

After a network value has been stored in a defined variable, it may later become necessary to push it back on The Stack using the **push defined** command. Note in the following example how the counter increments after the push.

```
xfst[0]: read regex a b c* d (e) f+ ;
xfst[1]: define VarX
xfst[0]: push defined VarX
xfst[1]:
```

The command `push defined VarX` is completely equivalent to the command `read regex VarX ;` (but note that the latter command compiles **VarX** as a regular expression and so needs to be terminated with a semicolon).

Pushing the value of the defined variable **VarX** onto The Stack does not reset or destroy **VarX**; its value persists in the symbol table until **VarX** is reset by another **define** or undefined using the **undefine** command.

```
xfst[1]: undefine VarX
```

Undefinition causes the memory tied up by **VarX**, which may be considerable, to be recycled and made available for other networks.

3.2.5 Basic Regular Expressions Denoting Relations

crossproduct

Returning to our review of regular-expression operators, the basic regular expressions for denoting relations are shown in Table 3.10. Other rule-like notations for denoting relations will be presented later.

`U .x. L`	The crossproduct of `U` and `L`, where `U` and `L` are regular expressions denoting languages, not relations. The result is a relation that maps every string in `U`, the upper language, to every string in `L`, the lower language.
`U:L`	The crossproduct of `U` and `L`, where `U` and `L` are regular expressions denoting languages, not relations. The expression `U:L` is equivalent to `[U .x. L]`. The colon is a general-purpose crossproduct operator, but it has higher precedence than the `.x.` operator; in particular, it has higher precedence than concatenation.
`a:a`	Denotes the relation <"a", "a"> but is, by convention, stored as a simple label **a** in the network. The regular expression `a` is therefore equivalent to `a:a`.
`?:?`	Denotes the relation consisting of all pairs of ANY symbol mapped to ANY symbol.
`?`	Denotes the relation consisting of all symbols identity-mapped to themselves.

Table 3.10: Regular Expressions Denoting Relations

The crossproduct operation, denoted with the `.x.` operator, which consists of a period, an **x**, and another period, without intervening spaces, is the basic operation for defining a relation.

> In the `U .x. L` notation for the crossproduct, both `U` and `L` must denote languages, not relations. The overall expression `U .x. L` denotes the relation wherein each string of `U` is related to each string of `L`. The language denoted by `U` is the upper language and the language denoted by `L` is the lower language.

The colon notation was originally implemented only as a convenient shorthand for notating the crossproduct of two single symbols, e.g. `a:b` is equivalent to `[a .x. b]` and a bit more convenient to type. Now the colon can be used as a

general-purpose crossproduct operator, but it has a precedence higher than that of `.x.` (see page 202 for a precedence table).

> The colon (:) and the operator `.x.` are both general-purpose crossproduct operators, but the colon has higher precedence. In particular, it is important to note that the precedence of the colon operator is higher than that of concatenation, while the precedence of `.x.` is lower than that of concatenation.

Experimenting with the crossproduct Operation

This section will continue with some simple examples using the crossproduct operators. If you are comfortable with the notion of crossproducts, proceed to the next section.

Let us suppose that language `Y` consists of the two strings "`dog`" and "`cat`" and that language `Z` consists of the two strings "`chien`" and "`chat`", which happen to mean "`dog`" and "`cat`", respectively, in French. In **xfst**, they might be defined in the following way:

```
xfst[0]: clear stack
xfst[0]: define Y [ d o g | c a t ] ;
xfst[0]: define Z [ c h i e n | c h a t ] ;
```

If we then compute the crossproduct of `Y` and `Z` using **read regex**, the result will be a single transducer network on The Stack.

```
xfst[0]: read regex Y .x. Z ;
xfst[1]:
```

Invoke **xfst**, compile the relation as shown, and follow along in the tests below. Remember that in the expression `Y .x. Z`, the `Y` language is the upper-side of the relation and the `Z` language is the lower-side.

We now use **apply up** and **apply down** to test the behavior of our transducer. If we **apply up** or analyze the string "`chien`" we get back the two strings "`dog`" and "`cat`". Note that "`dog`" and "`cat`" are all the strings in the upper language `Y`.

```
xfst[1]:  apply up chien
dog
cat
```

Similarly, if we analyze the string "`chat`", we get the same answer.

Now try using **apply down** to the strings "`dog`" and "`cat`". You should see the following output.

```
xfst[1]:  apply down dog
chien
chat
xfst[1]:  apply down cat
chien
chat
```

The response should be "chien" and "chat" in both cases; these are all the strings in the lower language Z. This behavior results from the fact that the cross-product operator relates each string of one language to every string of the other.

> The crossproduct operator, .x., is a binary operator between two regular languages Y and Z, e.g. [Y .x. Z], such that Y is the upper-level language, Z is the lower-level language, each string in Y is related to each string of Z, and each string of Z is related to each string of Y. Y and Z themselves must both denote regular languages, not relations. The crossproduct [Y .x. Z] denotes a relation.

Students often wonder what possible use there is for an operator like .x. that relates each string in one language to every string of the other. In practice, however, the languages Y and Z often contain only a single string as in

```
xfst[0]:  clear stack
xfst[0]:  read regex [ [ d o g .x. c h i e n ] |
                       [ c a t .x. c h a t ] ] ;
```

Try entering this expression and using **apply up** and **apply down** to test the resulting network. The upper-side language again consists of the two strings "dog" and "cat", and the lower-side language again consists of the two strings "chien" and "chat", but the relation between the languages has changed. Now when you **apply down** "dog", you should get only "chien", and when you **apply down** "cat" you should get only "chat".

- With the relation denoted above, try **apply up** on "chien" and "chat". What are the results?
- Try **apply up** on "dog" and "cat". Try **apply down** on "chien" and "chat". What are the results? Why?
- Define a relation such that the upper-side language contains the two strings "dog" and "cat", and the lower-side language contains the four strings "chien", "chienne", "chat" and "chatte", and so that "dog" is related to the two strings "chien" and "chienne" and so that "cat" is related to both "chat" and "chatte". (A "chienne" is a female dog

(bitch) and a "chatte" is a female cat, while "chien" and "chat" are masculine or unmarked.) Compile your description using **read regex**. Test the resulting network using **apply up** and **apply down**.

You may have noted that the last two relations perform a kind of dictionary lookup or "translation" between English and French words. Although this particular kind of mapping, i.e. French-English, is not very efficiently stored as a finite-state transducer, the concept of relations as bilingual mappers or translators of words between two languages is valid and useful.

In a full-sized Spanish morphological analyzer, for example, the lower-side language is a set of millions of strings that look like conventionally spelled words of Spanish. The upper-side language, defined entirely by the linguist, consists of strings that typically contain baseforms and multicharacter TAG symbols that indicate tense, number, person, mood, aspect, etc. The relation between these two languages is defined, by the linguist, using finite-state grammars, compiled into transducers, so that when you **apply up** an orthographic (surface) word like "canto", you get back related upper-side or lexical strings like "canto+Noun+Masc+Sg" and "cantar+Verb+PresIndic+1P+Sg" that represent useful ANALYSES of the surface input. Conversely, if you **apply down** "canto+Noun+Masc+Sg" or "cantar+Verb+PresIndic+1P+Sg", the relation returns the surface word "canto".[3]

It is often necessary to designate a relation that has a certain symbol, e.g. **b**, on the upper side and the empty string (epsilon) on the lower side; this is notated `b:0` or `[b .x. 0]` or `[b .x. []]`. Similarly, the relation having **b** on the lower side and the empty string (epsilon) on the upper side can be notated `0:b` or `[0 .x. b]` or `[[] .x. b]`.

> In the regular expression `a:b`, remember that the **a** is on the upper side and the **b** is on the lower side of the resulting relation.

Try compiling the following relation:

```
xfst[0]:  clear stack
xfst[0]:  read regex [ s w i:a m | s w i m ] ;
```

Test it using **apply down** and **apply up**. The same relation could be notated as `[ s:s w:w i:a m:m | s:s w:w i:i m:m ]` because in a regular expression denoting a relation, the symbol `s` is compiled like `s:s`. This particular relation could also be defined as `[ {swim}:{swam} | {swim} ]`, as well as

[3]While the surface strings to be analyzed are usually pre-defined for us by the standard orthography, the upper-side analysis strings are designed according to the tastes and needs of the developers. This topic is discussed in detail in Chapter 5. The example analysis strings in the text represent only one possible way, among an infinity of ways, to design the upper-side language.

many other ways. There are, in fact, an infinite number of regular expressions that denote the same language or relation and compile into equivalent networks.

> The colon (:) is a general-purpose composition operator, but unlike `.x.` it has higher precedence than concatenation. The regular expression `[ s w i:a m | s w i m ]` is therefore equivalent to `[ s w [i:a] m | s w i m ]`.

N.u	Return the upper projection of network N.
N.l	Return the lower projection of network N.

Table 3.11: Operations for Extracting Projections of a Transducer

Projections

As explained in Chapter 2, a transducer encodes a relation between two regular languages, an upper language and a lower language. The upper language is also called the UPPER PROJECTION, and the lower language is also called the LOWER PROJECTION of the relation. Given a transducer N, the operations in Table 3.11 return one of the projections of N, i.e. they take a transducer and return a simple automaton encoding one of the two related languages.

The following **xfst** session defines a small relation and sets the variable `myvar`. Then the upper projection of `myvar` is computed and pushed onto the top of The Stack, and the words of that upper language are enumerated. Then the lower projection (the lower-side language) is extracted and enumerated. It should be clear that the operators in Table 3.11 disassemble a relation into its component languages. When N denotes a language, `N.u` and `N.l` are equivalent to `N`.

```
xfst[0]:  clear stack
xfst[0]:  define myvar [ [ d o g .x. c h i e n ] |
                         [ c a t .x. c h a t ] ] ;
9 states, 9 arcs, 2 paths.
xfst[0]:  read regex myvar.u ;
6 states, 6 arcs, 2 paths.
xfst[1]:  print words
cat
dog
xfst[1]:  pop stack
xfst[0]:  read regex myvar.l ;
7 states, 7 arcs, 2 paths.
```

```
xfst[1]:  print words
chat
chien
```

N.i	Return the upper-lower inverse of network N.
N.r	Return the left-to-right reverse of network N.

Table 3.12: Re-Orienting Operations

Orientations

Using the finite-state operations in Table 3.12, one can compute both vertical and left-to-right re-orientations of a network. Where N is a transducer, relating an upper language U and a lower language L, the INVERSE of N, traditionally notated N^{-1}, is an upside-down version of N, with the original upper language becoming the lower language, and vice versa. In **xfst** regular expressions, the inverse of N is denoted `N.i`. If N is a simple automaton, denoting a language, the inverse operation has no effect. Any network N is equivalent to `[[N.i].i]`.

The following **xfst** session starts with the same trivial relation/transducer as in the previous example. It then inverts the network, extracts the upper-side language, which is the original lower-side language, and prints it. It should be obvious that the inversion operation simply turns a transducer upside-down.

```
xfst[0]:  clear stack
xfst[0]:  define myvar [ [ d o g .x. c h i e n ] |
                         [ c a t .x. c h a t ] ] ;
9 states, 9 arcs, 2 paths.
xfst[0]:  read regex myvar.i.u ;
7 states, 7 arcs, 2 paths.
xfst[1]:  print words
chat
chien
```

If N is any network, the notation `N.r` denotes the left-to-right REVERSE of N.

```
xfst[0]:  clear stack
xfst[0]:  define myvar [ d o g | c a t ] ;
xfst[0]:  read regex myvar.r ;
xfst[1]:  print words
tac
god
```

3.3 Interacting with the Operating System

3.3.1 File I/0, Defined Variables and The Stack

So far we have used **xfst** as an interactive command-line interface; we type a command at the prompt, and **xfst** executes the command and returns with another prompt. In this section, we will learn how **xfst** can use files for output and for input of data and commands.

Regular Expression Files

We are already acquainted with the **read regex** command and how it allows us to enter a regular expression manually at the prompt and have it compiled into a network.

```
xfst[0]: read regex  ( r e )
[ l o c k | c o r k | m a r k | p a r k ]
[ i n g | e d | s | 0 ] ;
xfst[1]:
```

Recall that the regular expression can extend over any number of lines, and that it must be terminated with a semicolon followed by a newline.

When you progress beyond trivial regular expressions, typing them manually into the **xfst** interface will soon become tedious and impractical; a single error will prevent compilation and force you to retype the entire regular expression. **xfst** therefore allows you to type your regular expression into a REGULAR-EXPRESSION FILE, using your favorite text editor,[4] and to compile it using the **read regex** command, adding the < symbol to indicate that the input is to be taken from the file. The command schema is the following:

```
xfst[0]: read regex < filename
```

The regular-expression file should contain only a single regular expression, terminated, as always in **xfst**, with a semicolon and a newline. Optional comments can also appear in the file; such comments are introduced with either an exclamation mark or a pound sign (#) and continue to end-of-line. Your regular-expression file might appear as in Figure 3.4.

[4]When editing regular-expression files in a Unix-like system, be sure to use a text editor like **vi**, **pico** or **xemacs** that stores its output as plain text. Word processors typically try to store files in proprietary formats, including invisible markup codes that will confuse the **xfst** regular-expression compiler. In Windows, the Notepad editor creates plain-text files.

```
! this is a comment
# this is another comment
( r e )
[ l o c k | c o r k | m a r k | p a r k ]
[ i n g | e d | s | 0 ] ;
```

Figure 3.4: A Regular-Expression File Contains a Single Regular Expression with a Terminating Semicolon and Newline. It is also possible to include comments, which are introduced with an exclamation mark or a pound sign (#) and continue to end-of-line.

The text in a regular-expression file must contain a single regular expression, and, as with regular expressions typed manually in the **xfst** interface, it must be terminated with a semicolon *and a newline*. Failure to add the newline after the semicolon is a common mistake. It is an unfortunate quirk of the regular-expression compiler that an invalid regular-expression file, lacking just a final newline after the semicolon, is visually indistinguishable from a valid regular-expression file containing the final newline.

If the file is named `fragment.regex`, then the command

```
xfst[0]: read regex < fragment.regex
```

will cause the text to be read in and compiled into a network that is pushed onto The Stack, just as if the text had been typed manually at the command line. The advantage, of course, is that the regular expression needs to be typed only once, even if it is compiled many times; and you simply use your text editor to make any corrections or additions to the regular-expression file.

By convention at **Xerox**, regular-expression filenames are given the extension `.regex`. On some operating systems, such as Windows, you may be more constrained in the file extensions you can use.

Binary Files and The Stack

Once one or more networks are compiled and pushed onto The Stack, they can be saved to a BINARY FILE using the **save stack** command, specifying a filename as an argument. E.g. to save a network on The Stack to `myfile.fst`:

```
xfst[0]: clear stack
xfst[0]: read regex [ {dog} | {cat} | {elephant} ] ;
xfst[1]: save stack myfile.fst
xfst[1]:
```

As this example shows, saving The Stack to file does not clear or pop The Stack. The resulting file does not contain the regular expression `[ {dog} | {cat} | {elephant}]` but rather a binary representation of the compiled network; such a binary file is not human-readable.

> By convention at **Xerox**, binary files are often given the extension `.fst` or `.fsm`.

The **save stack** command saves all the networks on The Stack, so the resulting binary file may store any number of pre-compiled networks. The network or networks saved in a binary file can be read back onto The Stack, in the original order, using the **load stack** command.

```
xfst[0]: clear stack
xfst[0]: load stack myfile.fst
xfst[1]:
```

If the binary file contains multiple networks, then multiple networks will be pushed onto The Stack, and the appropriate number will be displayed in the prompt. Loading a binary file pushes the stored networks back onto The Stack, on top of any other networks that may already be there.

Binary Files and Defined Variables

Compiled networks in **xfst** must be stored on The Stack, in defined variables, or in a binary file. You are already acquainted with the variant of the **define** command that is followed by a regular expression, terminated with a semicolon and a newline, as in

```
xfst[0]: define Var [ m o u s e |
r a b b i t |
b i r d ] ;
```

You are also acquainted with the variant of **define** that has no overt argument and so, like so many **xfst** commands, takes as its default argument the top network on The Stack, popping it and assigning the value to the variable.

```
aardvark
apple
avatar
binary
boy
bucolic
cat
coriander
cyclops
day
```

Figure 3.5: A Typical Wordlist

```
xfst[0]: clear stack
xfst[0]: read regex [ m o u s e |
r a b b i t |
b i r d ] ;
xfst[1]: define Var
xfst[0]:
```

We saw above that a user might well want to save a set of networks on The Stack to file, and the same may apply to a set of networks stored in variables. The command to save all the current defined-variable networks to file is **save defined**, which takes a filename argument.

```
xfst[0]: save defined myfile.vars
```

The command to read such a file back in, restoring all the previous variable definitions, is **load defined**.

```
xfst[0]: load defined myfile.vars
```

The **save defined** command allows you to save potentially useful networks to file and restore them later, using **load defined**, perhaps in a future **xfst** session.

Wordlist or Text Files

In practical natural-language processing, it is often useful to take a simple WORDLIST file and compile it into a network. By simple wordlist file we mean a file like that in Figure 3.5 containing a list of words, one word to a line, where each word consists of normal alphabetic characters (i.e. there are no multicharacter symbols). The example shows the words listed in alphabetical order, but this is not required.

One tedious way to compile this list of words, which effectively enumerates a language, into a network is first to edit the file and convert it into an appropriate

```
a a r d v a r k   |
a p p l e         |
a v a t a r       |
b i n a r y       |
b o y             |
b u c o l i c     |
c a t             |
c o r i a n d e r |
c y c l o p s     |
d a y             ;
```

Figure 3.6: A Wordlist Converted into a Regular Expression

```
{aardvark}    |
{apple}       |
{avatar}      |
{binary}      |
{boy}         |
{bucolic}     |
{cat}         |
{coriander}   |
{cyclops}     |
{day}         ;
```

Figure 3.7: Another Way to Convert a Wordlist into a Regular Expression

regular-expression file like that in Figure 3.6 or the equivalent in Figure 3.7. The converted file can then be compiled using **read regex** < ***filename***.

Mercifully, **xfst** provides a special compiler called **read text** that takes as input a simple wordlist as shown in Figure 3.5 and compiles it into the network that encodes the language. **read text** automatically "explodes" each word, compiling it as a concatenation of alphabetic symbols, and all the words in the list are unioned to form the result. The resulting network, as usual, is pushed onto The Stack. For example, assume that the file `myfile.wordlist` contains a list of words, with one word/string to a line.

```
xfst[0]: read text myfile.wordlist
xfst[1]:
```

In such **read text** commands, the name of the file to be read from can optionally

be preceded with a left angle bracket.

```
xfst[0]: read text < myfile.wordlist
xfst[1]:
```

If it's small enough to be humanly convenient, the wordlist to be compiled by **read text** can be typed in interactively. If one types **read text** followed by a newline, **xfst** goes into a text-entry mode and displays the prompt `text>`. After typing one string per line, the user exits text-entry mode by typing the pseudo-entry `END;` (in all capital letters, and including the semicolon), e.g.

```
xfst[0]: read text
text> dog
text> elephant
text> gorilla
text> whale
text> salamander
text> END;

1.0 Kb. 30 states, 33 arcs, 5 paths.
xfst[1]:
```

Instead of typing `END;`, Unix users can type Control-D to escape from text-entry mode; on Windows, the equivalent is Control-Z.

The reverse operation, **write text** > ***filename***, takes the top network on The Stack, which must encode a language (not a relation), and outputs all the words recognized by the network to an indicated file. If there is no network on The Stack, then **xfst** will return an error message.

```
xfst[1]: write text > myfile.wordlist
```

The name of the output file must be preceded with a right angle bracket. If no output file is specified, then **write text** is effectively equivalent to **print words**, outputting the strings to the standard output (the terminal).

Spaced-Text Files

The **read text** and **write text** commands just presented read and write simple wordlist files that contain straightforward alphabetic words, typed one to a line. Such a file contains a set of words and therefore enumerates a finite language. The routines are reciprocal in that any wordlist file created from a network by **write text** can be read back in and compiled into an equivalent network using **read text.**

If a network contains multicharacter symbols like `+Noun`, `+Sg` and `+Pl`, or if it is a transducer (encoding a relation, which is a set of ordered string pairs), then **write text** and **read text** cannot be used. **xfst** will prompt you in such cases to use **write spaced-text** and **read spaced-text** instead.

read text reads a plain wordlist file and compiles it into a simple automaton that recognizes that set of words. **write text** does the reverse. For transducers and for any network containing multicharacter symbols, use the commands **read spaced-text** and **write spaced-text**.

```
d o g +Noun +Sg
d o g

d o g +Noun +Pl
d o g s

e l e p h a n t +Noun +Sg
e l e p h a n t

e l e p h a n t +Noun +Pl
e l e p h a n t s

l o u s e +Noun +Sg
l o u s e

l o u s e +Noun +Pl
l i c e

o x +Noun +Sg
o x

o x +Noun +Pl
o x e n

m o u s e +Noun +Sg
m o u s e

m o u s e +Noun +Pl
m i c e
```

Figure 3.8: Typical File Output from **write spaced-text**

Figure 3.8 shows part of a typical output file from **write spaced-text**, with pairs of strings separated with a blank line, and with individual symbols spaced out. The first string of each pair is taken to represent the upper member of the ordered pair,

and all symbols are spaced, allowing +Noun, +Sg and +Pl to be recognized and handled appropriately as multicharacter symbols. The routines are reciprocal in that any wordlist file created from a network by **write spaced-text** can be read back in and compiled into an equivalent network using **read spaced-text**.

The syntax is similar to that of **read text** and **write text**, e.g. the following commands would result in two equivalent networks being placed on The Stack.

```
xfst[0]: clear stack
xfst[0]: read regex [ [ d o g %+Noun %+Sg .x. d o g ]
| [ d o g %+Noun %+Pl .x. d o g s ] ] ;
xfst[1]: write spaced-text > myfile.spaced
xfst[1]: read spaced-text myfile.spaced
xfst[2]:
```

If **write spaced-text** is not given a filename argument, it will output to the terminal.

lexc Files

Source files written in the **lexc** language, which will be presented in Chapter 4, can be compiled from within **xfst**.

```
xfst[0]:  read lexc < lexc_source_file
xfst[1]:
```

This command, parallel to **read regex** from a file, requires the left-angle-bracket, and it handles only a single input file at a time. The resulting network is pushed onto The Stack.

Prolog Files

For completeness we mention yet another kind of input-output file, the PROLOG FILE. The command **write prolog** outputs networks to file in a Prolog format that can be read back in by **read prolog**. See the **help** messages for further information.

3.3.2 Referring to Binary Files within Regular Expressions

A file storing a single binary network, a regular expression, a wordlist, a spaced wordlist or a prolog representation can be referenced and used directly in regular expressions. Assume, for example, that a network was compiled and stored in the binary file `myfile.fst`. Then `@"myfile.fst"` is a regular expression that denotes the value of that network.[5] Such expressions can appear inside other regular expressions within **read regex** and **define** commands, and in regular-expression files. The following example causes two networks to be read from two previously stored binary files, `myfile.fst` and `yourfile.fst`, unions the networks together, and pushes the result on The Stack.

[5]This example shows the name of the file surrounded with double quotes, which is necessary when the filename contains non-alphabetic characters like the period.

```
xfst[0]: read regex  @"myfile.fst" | @"yourfile.fst" ;
xfst[1]:
```

Table 3.13 summarizes the various regular-expression operators that are available for referencing a file. Binary files are by far the most used in practice. Double quotes are shown around the *filename* in all examples, but they are optional unless, as in the example just above, the *filename* contains non-alphabetic characters like the period.

`@"filename"`	Denotes the network value stored in the binary file `filename`. The file format, which is compressed, is read and decompressed.
`@bin"filename"`	Equivalent to `@"filename"`.
`@re"filename"`	Denotes the network value represented in the regular-expression file `filename`. The regular-expression compiler is called.
`@txt"filename"`	Denotes the network value represented by the wordlist file `filename`. The wordlist (text) compiler is called.
`@stxt"filename"`	Denotes the network value represented in the spaced-text file `filename`. The spaced-text compiler is called.
`@pl"filename"`	Denotes the network value stored in the Prolog file `filename`. The Prolog notation is read and rebuilt into the original network.

Table 3.13: Regular Expressions Referencing a File

3.3.3 Scripts in xfst

Script Files

When working interactively with **xfst**, you type a command at the command line, and **xfst** executes it and returns a new prompt. An **xfst** session consists of a sequence of commands and responses. For complex operations, and especially those that will be performed multiple times, you can type a sequence of **xfst** commands into a SCRIPT file using any convenient text editor and then command **xfst** to execute the entire script. The effect is the same as if all the commands in the script were retyped manually. The advantages, of course, are that the script can be edited until it is correct and that the entire script can then be rerun as many times as desired without retyping.

If `myfile.xfst` is an **xfst** script file, pre-edited to contain a list of **xfst** commands, you can execute it from within the interactive interface by using the **source** command.

```
xfst[0]: source myfile.xfst
```

The filename may in fact be a complex pathname.

> By convention, script files are often given the extension `.xfst` or `.script`. In some operating systems, your choice of extensions may be more constrained.

When you first invoke **xfst** from the UNIX command line, you can also specify a script name after the -f flag.

```
unix> xfst -f myfile.xfst
unix>
```

When invoked in this way, **xfst** will execute the script and then exit automatically back to the operating system. To execute a start-up script and then continue to enter commands manually in the **xfst** interactive loop, invoke **xfst** with the -l flag.

```
unix> xfst -l myfile.xfst
xfst[0]:
```

In general, any sequence of **xfst** commands that you might type manually at the command line can be edited in a script file, executed with **source** (or from the UNIX command line using the -l and -f flags), re-edited until the commands are correct, and then re-executed whenever you desire. An **xfst** script can also contain comments, which are introduced with an exclamation mark or a pound sign (#) and continue to end-of-line. The construction of some linguistic products, especially tokenizers and taggers, traditionally involves the writing of complex **xfst** script files.

> **xfst** beginners often confuse regular-expression files and script files. A regular-expression file contains only a single regular expression, terminated by a semicolon and a newline, and is read using the command `read regex < filename`. A script file, in contrast, contains a sequence of **xfst** commands; scripts are executed using the `source` command. Inside a script file, there may of course be **define** and **read regex** commands that include regular expressions.

echo

The **echo** command displays its string argument, which is terminated by a newline, to the terminal. **echo** is primarily useful in scripts to give the developer some runtime feedback on the progress of time-intensive computations. The general template for **echo** commands is the following:

```
echo  text_terminated_by_a_newline
```

The following script for building a lexical transducer includes potentially useful **echo** commands. The asterisks and angle brackets are not necessary but help to make the echoed text stand out on your terminal.

```
echo *** Script to build lt.fst ***
clear stack
echo << Creating the lexicon FST >>
read regex < lex.regex
define LEX
echo << Creating the rule FST >>
read regex < rul.regex
define RUL
echo << Creating the lexical transducer >>
read regex LEX .o. RUL ;
save stack lt.fst
echo << Output written to binary file lt.fst >>
echo << End of Script >>
```

3.3.4 Tokenized Input Files for apply up and apply down

Tokenized files of words, i.e. files with one word per line, can also be used as input to the **apply up** and **apply down** commands. First recall that the top network on the stack can be applied to individual words by typing the word manually after **apply up** or **apply down** as appropriate. These examples were produced with an English morphological-analyzer transducer on The Stack.

```
xfst[1]:  apply up  sink
sink+Noun+Sg
sink+Verb+Pres+Non3sg
xfst[1]:  apply down  sink+Noun+Sg
sink
```

The abbreviations **up** and **down** can be used instead of the full names.

If you need to test a number of words, then typing "`apply up`" or "`apply down`" (or even just the command abbreviations "up" or "down") before each one of them soon becomes tedious. If you enter simply **apply up** or **up** without an argument, **xfst** will switch into apply-up mode and display the prompt `apply up>`. Input words can then be typed without any prefix.

```
xfst[1]: apply up
apply up> sink
sink+Noun+Sg
sink+Verb+Pres+Non3sg
apply up>  sank
sink+Verb+PastTense+123SP
apply up> sunk
sink+Verb+PastPerf+123SP
sunk+Adj
apply up> END;
xfst[1]:
```

To escape from apply-up mode back to normal **xfst** interactive mode, enter "`END;`" as shown, including the semicolon. Apply-down mode works in just the same way.

Finally, **apply up** and **apply down** can take batch input from pre-edited tokenized files that have one string (i.e. token) per line.[6] If the tokenized file is named `english-input` and contains the two strings

```
left
leaves
```

and the same English transducer is on the top of The Stack, then the output is the following:

```
xfst[1]:  apply up < english-input
Opening file english-input...

left
leave+Verb+PastBoth+123SP
left+Adv
left+Adj
left+Noun+Sg

leaves
leave+Verb+Pres+3sg
leave+Noun+Pl
leave+Verb+Pres+3sg
leaf+Verb+Pres+3sg
leaf+Noun+Pl
Closing file english-input...
xfst[1]:
```

The output for each input word consists of a blank line, the input word itself, and a set of solutions. The **apply down** utility can also take its input from a tokenized file in exactly the same way.

```
xfst[1]:  apply down < english-solution-string-list
```

[6]Each whole line of the input file, including internal spaces, is treated as a string of input. The tokenized file therefore cannot contain comments of any kind.

3.3.5 System Calls

The **system** command passes its string argument, which is terminated by a newline, to the host operating system. In a Unix system, for example, the `ls` command causes the contents of the current directory to be displayed on the terminal. The following **system** command calls the Unix `ls` command without having to exit from **xfst**.

```
xfst[0]:  system ls
```

In modern multiwindow operating systems, where **xfst** can be run in one window and the bare operating system in another, **system** calls are seldom used.

Another little used command in **xfst** is **print directory**, which prints out the contents of the current directory. It is sometimes useful when inputting and outputting files.

```
xfst[0]: print directory
```

In a Unix system, this is the equivalent of invoking

```
xfst[0]: system ls -F
```

See **help print directory** and **help system** for more information.

3.4 Incremental Computation of Networks

3.4.1 Modularization

Any finite-state language or relation can theoretically be described in a single, monolithic regular expression. However, as you model more and more complicated languages and relations, your regular expressions will tend to get correspondingly bigger and much harder to read and edit. In extreme cases, a single large regular expression may become difficult for your computer to compile. Whether for human or computational considerations, it is often useful to break up the task into smaller modules.

xfst offers two approaches to modularization, which can be characterized as computing with defined variables and computing on The Stack. The two approaches are not incompatible at all, and may be mixed as the developer finds necessary and convenient.

3.4.2 Computing with Defined Variables

The **define** utility gives us a way to compile a regular expression, set a variable to the network value, and then use the variable in subsequent regular expressions. In this approach, The Stack may be left completely untouched.

As an exercise, define the following three variables.

```
xfst[0]:  define FEE [ a | b | c | d ] ;
xfst[0]:  define FIE [ d | e | f | g ] ;
xfst[0]:  define FOO [ f | g | h | i | j ] ;
```

Recall that once such a variable is defined, you can make the system display all the words in the language by invoking the **print words** command, e.g.

```
xfst[0]: print words FEE
d
c
b
a
xfst[0]:
```

To see more detailed information about the network, use the **print net** command:

```
xfst[0]: print net FEE
Sigma: a b c d
Size: 4
Net: EE3E0
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   a -> fs1, b -> fs1, c -> fs1, d -> fs1.
fs1:  (no arcs)
xfst[0]:
```

Once networks are defined and named, the names can be used in subsequent regular expressions. For example, the intersection of FEE and FIE can be computed and the resulting language printed to the screen with the following commands.

```
xfst[0]:  define mylang1 FEE & FIE ;
xfst[0]:  print words mylang1
d
xfst[0]:
```

Try the following operations, using **define** and variables.

- Compute the intersection of FIE and FOO. What word(s) does the resulting network recognize?
- Compute the intersection of FEE and FOO. What word(s) does the resulting network recognize?
- Compute the union of FEE and FIE. What word(s) does the resulting network recognize?

- Compute FEE minus FIE. What word(s) does it recognize?
- Define three variables:

  1. Prefix, including the words "un" and "re"
  2. Root, including the words "lock" and "cork"
  3. Suffix, including the words "ing", "ed", "s" and the empty string

  and then concatenate them in the order Prefix followed by Root followed by Suffix using another regular expression. What words does the resulting network recognize?

Computing with defined variables is intuitive to most users and leads to more readable and maintainable scripts (see Section 3.3.3).

3.4.3 Computing on The Stack

Stack-Based Finite-State Algorithms

As an alternative to building networks via regular-expressions, where syntactic operators denote finite-state operations such as union (|), intersection (&), subtraction (-) and composition (.o.), **xfst** also provides explicit spelled-out commands like **union net**, **intersect net**, **minus net**, **concatenate net** and **compose net** that invoke the algorithms directly, popping their arguments off of The Stack and pushing the resulting network back onto The Stack.

The way to compute networks on The Stack is to push suitable arguments and then invoke a command like **union net**; this **union net** pops its arguments off The Stack, computes the result, and pushes the resulting network, encoding the union of the arguments, back onto The Stack. Different commands need different numbers of arguments. Some unary commands, such as **negate net**, **invert net**, **reverse net** and **one-plus net**, refer to and operate on only one net, the top network on The Stack. Some binary commands, such as **crossproduct net** and **minus net**, operate on the top two networks on The Stack, and other "n-ary" commands, such as **union net**, **intersect net**, **concatenate net** and **compose net**, operate on all the networks on The Stack.

Try entering the following sequence of commands. Trace what is happening at each step, sketching diagrams as necessary. In particular, trace the status of The Stack itself after each command.

```
xfst[0]: clear stack
xfst[0]: read regex d o g | c a t | e l e p h a n t ;
xfst[1]: read regex b i r d | s n a k e | h o r s e ;
xfst[2]: print stack
xfst[2]: union net
xfst[1]: print stack
xfst[1]: print net
xfst[1]: print words
```

The commands above compute the union of two networks, encoding two small languages, that have been pushed onto The Stack. The **union net** command pops the networks off of The Stack (in this case there are only two networks), computes the union and pushes the resulting network, leaving a single network on The Stack. Similarly, the commands below compute the intersection of three small networks. Again, trace what is happening at each step, sketching diagrams as necessary. Recall that the **read regex** commands can extend over multiple lines; as always, the regular expression part of the command must be terminated with a semicolon followed by a newline.

```
xfst[0]: clear stack
xfst[0]: read regex
d o g |
c a t |
e l e p h a n t |
b i r d |
w o r m ;
xfst[1]: read regex
b i r d |
s n a k e |
h o r s e |
w o r m ;
xfst[2]: read regex
s n a k e |
d o n k e y |
c r a b |
f i s h |
w o r m |
f r o g ;
xfst[3]: print stack
xfst[3]: intersect net
xfst[1]: print stack
xfst[1]: print net
xfst[1]: print words
```

If you want more practice, redo the two previous exercises using defined variables and perform the union or intersection using regular-expression operators.

intersect net and **union net** are two COMMUTATIVE finite-state operations where the order of the arguments is irrelevant. The arguments in the two examples above could have been pushed onto The Stack in other orders and the result would have been completely equivalent. With other non-commutative operations, however, the order is significant and a clear understanding of The Stack and its LIFO (Last-In, First-Out) behavior becomes crucial.

The order-dependent (non-commutative) operations we will exemplify here are **concatenate net** and **minus net**. Another one, **compose net**, is also order-dependent, but composition is a more difficult operation to grasp and will be handled later.

Concatenation (from *CATENA*, the Latin word for "chain") refers to the linear chaining together of languages, one after another. If we have two languages A and B, and we wish to concatenate B after A, we would notate this in the regular expression metalanguage as `[ A B ]`. It should be obvious that this denotes quite a different language from `[ B A ]`, which is the concatenation of B before A. To compute `[ A B ]` on The Stack, we must be aware that the first argument is A and the second argument is B; but because The Stack is LIFO, i.e. Last-In First-Out, we must first push B on The Stack and then A before we invoke the **concatenate net** command. See Figure 3.9.

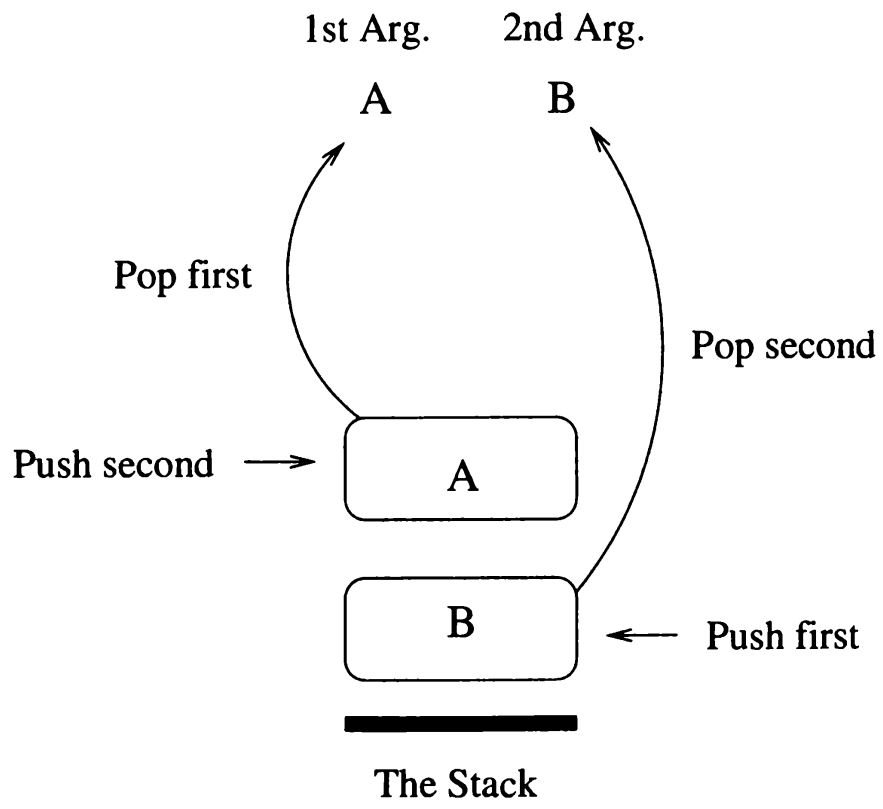


Figure 3.9: Pushing and Popping Networks. To compute the concatenation `[ A B ]` on The Stack using the command **concatenate net**, it is important to remember that 1) The Stack in **xfst** is last-in, first-out, and 2) the first-popped argument to **concatenate net** is the first argument popped off The Stack. Thus to compute the concatenation `[ A B ]`, the B network must be pushed first, and then the A network. Then when **concatenate net** is invoked, it will pop A (its first argument) first, and then pop B (its second argument) second. The same is true for other ordered operations like **minus net** and **compose net**.

concatenate net will then pop the top network, A, off The Stack as its first argument, after which it will pop the next network, B, off The Stack as its second argument, computing the concatenation of B after A or `[ A B ]`; the resulting

network will then be pushed onto The Stack. For ordered operations like **concatenate net**, the order in which you push arguments on The Stack is the opposite of how they are popped off as arguments; this seems backwards to most users, and it is a cause of many errors and much confusion.

The following example generates a network whose language contains a few regular English verbs. Note especially that the language of verb endings is pushed on The Stack first.

```
xfst[0]: clear stack
xfst[0]: read regex e d | i n g | s | [] ;
xfst[1]: print words
xfst[1]: read regex t a l k | w a l k | k i c k ;
xfst[2]: print stack
xfst[2]: print net
xfst[2]: print words
xfst[2]: concatenate net
xfst[1]: print words
```

When the **concatenate net** command is called, it pops its first argument (the network encoding the verb stems) off The Stack and then it pops its second argument (encoding the verbal endings) off The Stack. The concatenation is then performed and the result is pushed back onto The Stack.

The **minus net** operation is another one where the order obviously makes a difference: `[ A - B ]` is usually very different from `[ B - A ]`, just as the arithmetic expression `(5 - 2)` has a value distinct from `(2 - 5)`. As an exercise, define a language (think of it as the Big Language) containing the words

```
apple
peach
pear
orange
plum
watermelon
cherry
medlar
```

In a regular expression, remember to put white space between symbols that are to be concatenated or to enclose strings with curly braces to "explode" them. Then define another language (think of it as the Small Language) containing just the words

```
peach
plum
cherry
banana
```

Using the **minus net** command and The Stack, subtract the Small Language from the Big Language and print the words in the resulting language. Trace the effect

of the commands, sketching a representation of The Stack and showing how the various nets are pushed and popped during the processing.

> The Stack is a Last-In, First-Out data structure. If you have an ordered operation like concatenation whose arguments are A, B, C, and D, in that order, then to compute the operation equivalent to `[A B C D]` on The Stack, you must push D first, C second, B third and finally A. Then when you invoke **concatenate net** the arguments will be popped off The Stack one at a time in the appropriate order.

Unary Commands
invert net
lower-side net
negate net
one-plus net
reverse net
upper-side net
zero-plus net
Binary Commands
crossproduct net
minus net
N-ary Commands
compose net
concatenate net
intersect net
union net

Table 3.14: The Main Stack-Based Finite-State Commands. Unary commands operate on the top network on The Stack. Binary commands work on the top two networks on The Stack; and n-ary commands operate on all the networks on The Stack. Most users will find it wise to avoid these commands in favor of computing with defined variables. See Section 3.4.2.

The main commands that operate on stack-based networks are shown in Table 3.14. Computing on The Stack typically appeals to experts who are experimenting interactively and perhaps even debugging new finite-state algorithms; **xfst** started out as an algorithm debugging tool and only gradually added the capability

of defining networks via regular expressions, inputting from and outputting to files, etc. For most users, computing with defined variables and regular expressions (see Section 3.4.2) is usually safer and more perspicuous.

Reordering Networks on The Stack

When working interactively with **xfst**, users often push networks onto The Stack in the wrong order for subsequent stack-based operations. **xfst** offers the commands **rotate stack** and **turn stack** that can often resolve the order.

Let's assume that we want to define a simple language consisting of prefixes, root and suffixes, and that we want to concatenate them on The Stack. Intuitively, but incorrectly, most users begin by defining, and pushing onto The Stack, a network for prefixes first, then a network for the roots, and then a network for the suffixes, as in this **xfst** session:

```
xfst[0]: echo ! This is Wrong !
xfst[0]: clear stack
xfst[0]: read regex {re} | {un} | 0 ;
xfst[1]: read regex {cork} | {lock} ;
xfst[2]: read regex {ing} | {ed} | s | 0 ;
xfst[3]: concatenate net
```

The intent would then be to invoke **concatenate net** to pop and concatenate the three networks and push the resulting network back onto The Stack. Unfortunately, this intuitive way of going about the definition leaves the networks on The Stack in the wrong order; for **concatenate net** to work, The Stack must have the leftmost network to be concatenated on the top of the stack and the rightmost network on the bottom. See Figure 3.10.

In this example, we can recover from our error by invoking **turn stack**, which flips the order of all the networks on The Stack.

```
xfst[3]: turn stack
xfst[3]: concatenate net
xfst[1]: print words
recork
recorking
recorks
recorked
relock
relocking
relocks
relocked
uncork
uncorking
uncorks
uncorked
unlock
```

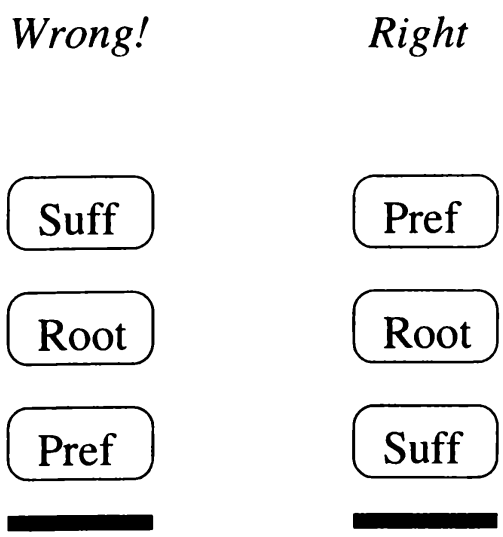


Figure 3.10: Wrong and Right Ordering of Arguments on The Stack. To compute the concatenation `[ Pref Root Suff ]` on The Stack using **concatenate net**, the arguments should be ordered as shown on the right, with `Suff` pushed first, then `Root`, then `Pref`. Backward stack ordering, as shown on the left, is a common user error. The command **turn stack** will flip the order of all the networks on The Stack, changing the wrong ordering to the right ordering in this case.

```
unlocking
unlocks
unlocked
cork
corking
corks
corked
lock
locking
locks
locked
```

Try this example, with and without invoking **turn stack** before invoking **concatenate net**.

xfst also provides the command **rotate stack** (see Figure 3.11), which pops the top network and reinserts it on the bottom of The Stack. When there are only two networks on The Stack, **rotate stack** is equivalent to **turn stack**.

For most users, it is best to avoid using stack-based commands like **concatenate net**, **minus net** and **compose net**; the ordering of arguments on The Stack causes much confusion. Instead, define variables and use the variables in subsequent regular expression as shown in Section 3.4.2, page 121.

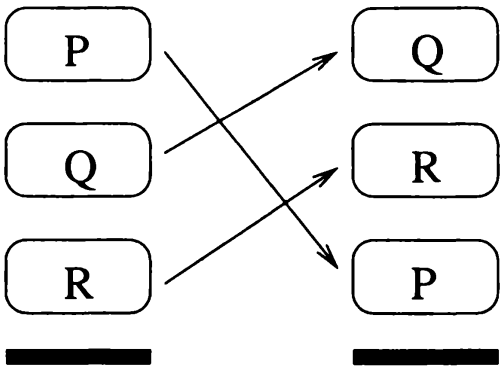


Figure 3.11: Rotation of Arguments on The Stack. The **rotate stack** command pops the top network of off The Stack and inserts it on the bottom of The Stack.

3.5 Rule-Like Notations

xfst offers a large selection of rule-like notations. They are all part of the extended **Xerox** regular-expression metalanguage, and they are compiled into networks in the usual ways, using the **define** and **read regex** commands.

3.5.1 Context Restriction

The restriction operator, =>, consisting of an equal sign followed immediately by a right angle bracket, forms regular expressions according to the following template

```
A => L _ R
```

where A, L and R are regular expressions denoting languages (not relations), and L and R are optional. The notation is rule-like in that the L functions as a left context and the R as a right context. The overall restriction expression denotes the

language of all strings with the restriction that if a string contains a substring from A, that substring must be immediately preceded by a substring from L and immediately followed by a substring from R. In other words, `A => L _ R` denotes the universal language minus all strings that do not satisfy the contextual restriction.

As a simple concrete example, consider

```
x => a _ e
```

The language denoted by this regular expression includes all strings that do not contain the symbol **x**; in addition, it contains all strings containing **x** where each **x** is immediately preceded by **a** and immediately followed by **e**. Thus the language includes the empty string, "a", "fish", "zzzzz", "axe", and "zzzzaxemmmm" but not "oxe" or "x" or "axi" that do not satisfy the contextual restrictions on the presence of **x**.

The restriction written as `A => L _ R` is in fact compiled as if it were written `A => ?* L _ R ?*`, extending the left context on the left, and the right context on the right, with the universal language `?*` . This reflects the requirement that an A must be immediately preceded by an L and immediately followed by an R; but it allows the L to be preceded by other symbols and the R to be followed by other symbols. A restriction such as `x => _` , which is compiled as `x => ?* _ ?*`, effectively restricts nothing and is just another notation for the universal language.

To block either contextual extension, the special `.#.` notation can be used to indicate the absolute beginning or end of a string. Thus `A => .#. L _ R` requires that any A be immediately preceded by an L that is at the beginning of the word; `A => .#. _ R` requires that the A itself be at the beginning of the word; `A => L _ R .#.` requires that A be immediately followed by an R which is at the end of a word; and `A => L _ .#.` requires that the A itself be at the end of a word. Both the beginning and the end of the word can be specified, as in `A =>.#. L _ R .#.`.

> The notation `.#.` in rule-like regular expressions designates the beginning or the end of a word, but it is not a symbol and does not appear as a label in a compiled network. In a network, the notion beginning-of-string is inherent in the start state, and the notion end-of-string is inherent in the final state(s).

Restriction expressions are just convenient shorthands for longer and less readable regular expressions using more fundamental operators (see Section 2.4.1). They are compiled into networks in the usual ways, using the **define** and **read regex** commands. Try the following and make sure that you understand the outputs.

```
xfst[0]: read regex x => a _ a ;
```

```
xfst[0]: apply up dog
dog
xfst[0]: apply up xylophone
xfst[0]: apply up laxative
laxative
xfst[0]: apply up axe
xfst[0]: apply up lax
```

Another restriction example, [A < B], read as "A before B", denotes the language of all strings such that no substring from B can precede, even at a distance, any substring from A. Similarly, [A > B], read "A after B", denotes the language of all strings such that no substring from B can follow, even at a distance, any substring from A. The restriction operators are summarized in Table 3.15.

A => L _ R	Any substring from A must be immediately preceded by a substring from L and immediately followed by a substring from R.
A < B	A before B. Denotes the language of all strings wherein no substring from B can precede any substring from A.
A > B	A after B. Denotes the language of all strings wherein no substring from B can follow any substring from A.

Table 3.15: Restriction Operators

3.5.2 Simple Replace Rules

As part of its extended regular-expression notation, **xfst** includes REPLACE RULES. These rules are very much like the rewrite rules used in traditional phonology, including classic works like *The Sound Pattern of English* (Chomsky and Halle, 1968) and the *Problem Book in Phonology* (Halle and Clements, 1983). Replace rules are an extension of regular expressions—they are in fact just shorthand notations for complicated and rather opaque regular expressions built with more basic operators. Like all regular expressions, replace rules are compiled using the **read regex** and **define** utilities available in the **xfst** interface. Grammars of replace rules can be written in regular-expression files, to be compiled with **read regex** < *filename* commands; and complex sequences of **xfst** commands for compiling and manipulating rule grammars can be written in **xfst** script files, which are run using the **source** command (see Section 3.3.3).

Replace rules do not increase the descriptive power of regular expressions, but they allow us to define complicated finite-state relations in perspicuous rule-like notations that are already familiar to formal linguists. The overall replace rule notation is very rich, and **Xerox** researchers continue to augment it. However,

in this section we will concentrate on the simplest and most often used forms of replace rules, and do some exercises, before moving on to the more esoteric types.

The most straightforward kind of replace rule, the right-arrow unconditional replace rule, is built on the following template:

```
A -> B
```

where A and B are regular expressions denoting languages (not relations), and the right-arrow operator consists of a hyphen and a right angle bracket, with no space between them. The overall right-arrow rule denotes a relation, and (with one exception to be discussed below) the upper language of the relation denoted by a right-arrow rule is the universal language. The relation is the identity relation, except that wherever an upper string contains a substring from A, the string is related to a surface string or strings containing a substring from B in place of the substring from A.

In a more down-to-earth example, the rule `[a -> b]` denotes the relation wherein every symbol **a** in strings of the upper-side language is related to a symbol **b** in strings of the lower-side language. The upper-side language of such a relation is the universal language; and any upper-side string that does not contain **a** simply maps to itself on the lower side. For every upper-side string that does contain **a**, it is mapped to a lower-side string that contains **b** in the place of **a**.

The fact that the upper-side language is the universal language means that the resulting network can be applied successfully in a downward direction to any input string. If the network for the rule is applied downward to the string "dog", the output is "dog". If it is applied downward to "aardvark", the output is "bbrdvbrk". When this right-arrow rule is used for generation, we often think of it procedurally as "not matching" the input "dog" and just passing the word through; but it "matches" the string "aardvark", which contains **a**s, and outputs a modification of the string, "bbrdvbrk", with the **a**s "replaced" by **b**s. Of course, no such procedural replacement is going on; the rule denotes a relation, and compiles into a transducer. The application of the rule for inputs "dog" and "aardvark" simply returns the output strings related to each input string.

> The expression `[a -> b]` defines a regular relation between two languages. It is tempting, and occasionally irresistible, to think of this as a little algorithmic program that "changes" **a**s into **b**s during generation. However, no such program or process is involved, and such rules are properly thought of as representing an infinite set of pairs of strings that are related to each other in a certain way.

Replace rules are regular expressions, and they are are compiled into networks and pushed on The Stack in the usual way. Try reproducing the following example step by step.

```
xfst[0]: read regex a -> b ;
```

As with any other regular expression, it is also possible to define a variable to hold the network value.

```
xfst[0]:  define Rule1  a -> b ;
```

When such a relation is compiled into a network and pushed on the top of The Stack, you can test it as usual with the **apply down** and **apply up** commands. The **apply down** operation applies the network on the top of The Stack in a downward direction to the input string, what we normally think of as generation. For example, if the network compiled from `[a -> b]` is applied downward to the input string "`abcda`", the result is the string "`bbcdb`". That is, the relation states that for every **a** in an upper-side string, there must be a **b** in related strings on the lower-side. Try the following:

```
xfst[0]: clear stack
xfst[0]: read regex a -> b ;
xfst[1]: apply down dog
dog
xfst[1]: apply down aardvark
bbrdvbrk
xfst[1]: apply down abcda
bbcdb
xfst[1]: clear stack
xfst[0]: read regex c -> r ;
xfst[1]: apply down cat
rat
xfst[1]: apply down dog
dog
```

Note that if the rule does not "match" an input string, as in the case of "`dog`", the string is identity mapped, i.e. it is passed through unchanged.

In the general pattern for simple replace rules:

```
A -> B
```

`A` and `B` are not limited to single symbols but can be arbitrarily complicated regular expressions that denote languages, but not relations; the overall rule denotes a relation. `A` and `B` do not have to be the same length or conform to any limitations other than denoting regular languages.

Replace rules can also specify left and right contexts, as in the template

```
A -> B || L _ R
```

which indicates that any lexical or upper-side string containing a string from `A` is related to a string or strings on the lower side containing a string from `B` but

only when the left context ends with L and the right context begins with R. From a purely syntactic point of view, the arrow consists of a hyphen and a right angle bracket, with no space in between; and the context separator || is a single operator consisting of two vertical-bar characters, with no space in between. Otherwise, white space can be used freely as in other regular expressions. The underscore character indicates the site of the replacement between the two contexts.

The double-vertical-bar operator between the replacement specification and the context indicates that both the context expressions must match on the upper-side of the relation. Other variations of this notation are explained in Chapter 2, below in Section 3.5.5, and in the online documentation. The right-arrow || rules are the most commonly used and correspond most closely to traditional rewrite rules.

As with the restriction notation in Section 3.5.1, the left and right contexts are automatically extended with the universal language, such that the rule written

```
A -> B || L _ R
```

is compiled as

```
A -> B || ?* L _ R ?*
```

A, B, L and R can denote arbitrarily complex languages (not relations), and L and R are optional. If a context L or R is empty, the resulting left or right context is then the universal language (any possible context is allowed). The overall rule denotes a relation and is compiled, using the usual **define** or **read regex** utilities, into a transducer.

```
xfst[0]: read regex A -> B || L _ R ;
```

From the naive algorithmic point of view, we often think of this rule as specifying that a lexical (upper-side) substring from A is to be replaced obligatorily by the substrings from B in the specified context; however, the rule in fact states a restriction on the relation between two regular languages and compiles into a finite-state transducer.

As explained in Section 2.4.2, replace rules can have multiple replacements that are not contextually conditioned, as in

```
[ A -> B, B -> A ]
```

or multiple replacements that share the same context; the multiple replacements are again separated by commas. The use of square brackets around such rules often improves readability, even if they aren't formally required.

```
[ A -> B, C -> D, E -> F || L _ R ]
```

In addition, a rule can have multiple contexts, similarly separated by commas, e.g.

```
A -> B || L1 _ R1, L2 _ R2
```

A single rule can have both multiple replacements and multiple contexts.

```
A -> B, C -> D, E -> F || L1 _ R1, L2 _ R2, L3 _ R3
```

When regular expressions containing rules are compiled by **define** or **read regex** commands, the regular expressions must, as always, be terminated with semicolons. In practice, to avoid confusing the compiler and yourself, it is often wise to surround replace rules with square brackets.

```
xfst[0]: read regex [ a -> b || l _ r ] ;
xfst[1]:
```

In replace-rule contexts, use the special symbol .#. to refer to either the absolute beginning or the absolute end of a string. The .#. is a special notation spelled with a period, a pound sign, and another period, with no intervening spaces. The end-of-word notation is not a symbol and does not appear in the sigma alphabet of the resulting network.

```
e -> i || .#. p _ r
```

This rule "replaces" lexical (upper-side) **e** with surface (lower-side) **i** when it appears before **r** and after a **p** at the beginning of a word. When a .#. appears in the left context, it refers to the beginning of the word; when one appears in the right context, it refers to the end of the word. It is possible to define rules whose contexts refer to both the beginning and the end of a word.

```
g o -> w e n t || .#. _ .#.
```

Occasionally one wants to specify that an alternation occurs either in a specific symbol context or at a word boundary. For example, if A maps to B either before a right context R or at the very end of the word, this could be notated as either

```
A -> B || _ R , _ .#.
```

or equivalently as

```
A -> B || _ [ R | .#. ]
```

Optional right-arrow mapping is indicated with the (->) operator, which is just the right arrow with parentheses around it. For example, the rule

```
a (->) b
```

maps each upper-side **a** both to **b** and to the original **a**.

```
xfst[0]: clear stack
xfst[0]: read regex a (->) b ;
xfst[1]: apply down abba
bbbb
bbba
abbb
abba
```

The upper-side language of a right-arrow rule is usually the universal language. The one exceptional case is when a language is mapped to the null language. Recall that `?*` denotes the universal language, which contains all possible strings. The notation ~`[?*]` therefore denotes the complement of the universal language, which is the null or empty language containing no strings at all. There are in fact an infinite number of ways to denote the empty language, including `[a - a]`, which effectively subtracts a language from itself, leaving no strings. The exceptional replace rules then look like the following:

```
A -> ~[?*]
```

or

```
A -> [a - a]
```

In such cases, all strings containing a string from `A` are removed from the upper language, leaving an identity relation of all strings that do not contain `A`. Thus `[A -> ~[?*]]` is equivalent to ~`$[A]`, the language of all strings that do not contain a substring from `A`.

Before looking at any other replace-rule types, we will consolidate our understanding of the right-arrow rules with some exercises.

3.5.3 Rule Ordering and Composition

The composition operation was introduced formally in Section 2.3.1. We reintroduce it here in examples and exercises that involve replace rules and rule ordering. You are urged to reproduce the examples as you read.

The *kaNpat* Exercise

The Linguistic Phenomena to Capture As a first exercise in writing replace rules, and learning about rule ordering and composition, consider a fictional language, where *kaNpat* is an abstract lexical string consisting of the morpheme *kaN* (containing an underspecified nasal morphophoneme *N*) concatenated with the suffix *pat*. Here, as in many natural languages, strange things happen at morpheme boundaries. It so happens that in this language, an underspecified nasal **N** that occurs just before *p* gets REALIZED as (or "replaced by") an **m**. This is formalized in a replace rule as follows

```
N -> m || _ p
```

Read this rule informally as "N is obligatorily replaced by m when it appears in the context before p". This is a garden-variety kind of nasal assimilation that occurs in many languages.

Notice that the left context in the preceding rule is empty, which means that the left context doesn't matter; any left context will match. The application of this rule to the input string "kaNpat" yields the new intermediate string "kampat". A second rule in this language states that a **p** that occurs just after an **m** gets realized as **m**, and this rule is formalized as

```
p -> m || m _
```

Note that in this formalism, the linguist must keep track of the rule order. In this language, the first rule can "feed" the second; i.e. the **m** in the left context of the second rule can be either a lexical (underlying) **m** or an **m** that replaced a lexical **N**. A complete DERIVATION from the underlying string "kaNpat" through "kampat" to the final surface string "kammat" is shown in Figure 3.12.

```
Lexical:        kaNpat

          N -> m || _ p

Intermediate:   kampat

          p -> m || m _

Surface:        kammat
```

Figure 3.12: Rules mapping *kaNpat* to *kampat* to *kammat*

A Replace-Rule Grammar for *kaNpat* Each replace rule is compiled into a finite-state transducer, and the mathematics of finite-state transducers tells us that if a first transducer maps from initial "kaNpat" to intermediate "kampat", and that if a second transducer maps from intermediate "kampat" to final "kammat", then there exists, mathematically, a single transducer that maps, in a single step, directly from "kaNpat" to "kammat", with no intermediate level. Algorithmically, that single transducer is the COMPOSITION of the two original networks, and at **Xerox** we have an efficient composition algorithm that can compute it in most practical cases.

Composition, which is an ordered (non-commutative) operation, is indicated in the **Xerox** regular-expression formalism by the `.o.` operator as shown in Figure 3.13. The composition operator is spelled with a period, the small letter **o** and

another period, with no intervening spaces. Composition is an n-ary operation, meaning that any number of transducers can be composed together in a cascade.

```
[ N -> m || _  p ]
.o.
[ p -> m || m  _ ] ;
```

Figure 3.13: A Regular Expression Denoting a Cascade of Rules. Rules in a cascade are combined via the composition operation. As the composition operator `.o.` is of very low precedence, the square brackets in this example are not formally necessary.

```
xfst[0]:  clear stack
xfst[0]:  define Rule1 [ N -> m || _ p ] ;
xfst[0]:  define Rule2 [ p -> m || m _ ] ;
xfst[0]:  read regex Rule1 .o. Rule2 ;
```

Figure 3.14: A Cascade of Rules Implemented via Defined Variables and Composition

```
xfst[0]:  clear stack
xfst[0]:  read regex [ p -> m || m _ ] ;
xfst[1]:  read regex [ N -> m || _ p ] ;
xfst[2]:  compose net
xfst[1]:
```

Figure 3.15: A Cascade of Rules Computed on The Stack. Most users find it best to avoid performing composition on The Stack.

Alternatively, one can compile the two rules separately, defining appropriate variables, and then compose them together in a subsequent regular expression as in Figure 3.14. One could also compile the rules separately, using **read regex**, pushing the results onto The Stack and then invoking **compose net** as in Figure 3.15. In such a case, it is vitally important to compile and push the second (lower) rule `[ p -> m || m _ ]` first, so that the network for the upper rule `[ N -> m || _ p ]` will be pushed on top of it before **compose net** is called.

As when using other ordered operations such as **minus net** and **concatenate net**, the order of the arguments pushed on The Stack is critical. It is up to you to ensure that the "top" rule network in a cascade is on the top of The Stack, and that the overall final order of arguments on The Stack mirrors the graphic ordering of the cascade, before you invoke the **compose net** command. Review Section 3.2.4 if you are still uncomfortable with the ordering of arguments on The Stack. Better yet, avoid performing such operations on The Stack in favor of computing with defined variables as shown in Figure 3.14.

Testing Your *kaNpat* Grammar

- Type the grammar using a text editor as a regular-expression file named something like `kaNpat.regex`. Remember that a regular-expression file contains just a single regular expression and must be terminated with a semicolon followed by a newline. Your regular-expression file should look like Figure 3.13, making sure that the semicolon is followed by a newline.

- Then launch the **xfst** interface and compile your grammar using the command **read regex** < ***filename***.

  ```
  xfst[0]: read regex < kaNpat.regex
  ```

  If there are no mistakes in the file, the grammar should compile cleanly, and the resulting transducer network will be pushed onto The Stack.

- Once you have the rule network on The Stack, you can use **apply down** to test the effect of the grammar on various input strings. Using **apply down**, the input string is matched against the upper side of the transducer, and the output string or strings are read off the lower side of the transducer. We often think of such downward application as GENERATION or LOOKDOWN.

- If you apply a network in the upward direction to an input string, using the **apply up** command, the input string is matched against the lower side of the transducer, and the solution or solutions are read off of the upper side of the transducer. We often think of upward application of the network as LOOKUP or ANALYSIS of the input string.

- In this example, we are primarily interested in generating from the abstract or underlying string "`kaNpat`", hoping that our rules will produce the correct surface form as output. With the rule grammar compiled and on the top of The Stack, try

  ```
  xfst[1]: apply down kaNpat
  ```

  If you typed and compiled the grammar correctly, the system should return "`kammat`". You have just written your first finite-state replace rules.

Experiments

1. Perform **apply down** on the string *kampat*. You should be able to explain the output by tracing its derivation through the rules.

2. Perform **apply down** on the string *kammat*. Again, trace the course of the derivation.

3. Perform **apply up** on the string *kammat*. Why are there multiple analyses? Hint: Remember that transducers are bidirectional mappers between strings in the lexical language and strings in the surface language.

4. Start over and compile the same grammar but with the order of the two rules reversed. Does it still work? Why or why not?

```
$_ = "kaNpat" ;
s/N(?=p)/m/g ;
s/(?<=m)p/m/g ;
print ;
```

Figure 3.16: The *kaNpat* Example in **Perl** 5.6 Works for Generation Only. This example takes the input string "`kaNpat`" and outputs "`kammat`".

Putting Replace Rules in Context The regular-expression notation we call replace rules corresponds very closely to the traditional rewrite rules used by phonologists, and much serious phonological work has been done with such rules. One grammar of Mongolian (Street, 1963) has a cascade of about 60 such rules mapping from abstract morphophonemic strings to surface strings written in the Cyrillic orthography.

Traditional rewrite-rule grammars were always somewhat problematic because

- The grammars usually existed only on paper and were checked tediously, and very incompletely, by hand.

- Where researchers did try to computerize their rewrite-rule grammars, they did not understand that the rules could be implemented as finite-state transducers. The solution was often to implement the cascade of rules as a cascade of ad hoc programs that matched the contexts and performed the appropriate replacements, passing the output to the next ad hoc program in the cascade. A grammar of 60 rules would thus require 60 different processing steps at runtime.

- Where grammars were successfully computerized as cascades of ad hoc programs, they tended to run only in a downward or generating direction. Running these grammars *backwards* to do analysis usually proved impossible or at least extremely inefficient. For example, generating the *kaNpat* example in a **Perl** (version 5.6) script would be especially easy, as shown in Figure 3.16, but it is impossible to run the script backwards to perform analysis because the Perl substitution commands are really procedural programs.

- Even when some researchers did understand, after Johnson's work (Johnson, 1972), that rewrite rules could theoretically be implemented as finite-state transducers, there were no rule compilers or composition algorithms available.

- The **Xerox** Finite-State Calculus now includes the algorithms necessary to compile and compose replace rules, resulting in single transducers that not only run efficiently but also run backwards (for analysis) as well as forwards (for generation). Transducers are inherently bidirectional. Replace rules can be used directly for building morphological analyzer/generators, or they can be used as an auxiliary formalism for filtering and customizing transducers built using the **Xerox** finite-state **lexc** compiler (Chapter 4).

- Replace rules are notational abbreviations of regular expressions built with more basic operators, and they compile into finite-state transducers.[7]

- Writing a grammar with replace rules usually requires writing a cascade of rules, with rules potentially feeding each other, and great care must be taken to order the rules correctly in the cascade.

- Current research and development in finite-state computing involves improving the algorithms and providing yet more perspicuous high-level notations for linguists to use. **XRCE** in particular continues work on replace rules, adding more operators and features; and we will introduce some of them below.

Parallel Rules

Sometimes rules cannot conveniently be ordered in a cascade but must apply in parallel. The classic example involves the exchange of two symbols, e.g. to replace

[7] Another rather different rule formalism, called "two-level" rules, was introduced by Kimmo Koskenniemi in his 1983 dissertation entitled *Two-level morphology: a general computational model for word-form recognition and production* (Koskenniemi, 1983); but two-level rules also compile into finite-state transducers. **Xerox** provides the **twolc** compiler for compiling two-level rules. Because a finite-state transducer made one way is mathematically as good as a finite-state transducer made any other way, the choice of using replace rules vs. **twolc** rules is ultimately a matter of human taste and convenience rather than theory or mathematics. The Xerox **twolc** compiler is included in the software licensed with this book, and the documentation for it can be obtained from the website `http://www.fsmbook.com/`.

all as by bs and, simultaneously, all bs by as. Thus for an input string "abba", the desired output from generation would be "baab". The individual rules are just `[ a -> b ]` and `[ b -> a ]`, but any attempt to order them yields the wrong results.

Consider the following attempt, where `[ a -> b ]` is ordered first.

```
xfst[0]: clear stack
xfst[0]: read regex a -> b .o. b -> a ;
xfst[1]: apply down abba
aaaa
```

Here the output is "aaaa", resulting conceptually from the top rule mapping the input string "abba" to the intermediate string "bbbb", which in turn is mapped to "aaaa" by the second rule in the cascade.

In the second attempt, we order `[ b -> a ]` first:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a .o. a -> b ;
xfst[1]: apply down abba
bbbb
```

which also fails to produce the desired result. It is obvious that neither ordering is right.

Luckily there is a way not to order the rules but to have them apply in parallel. For simple rules having no context specification, the parallel rules are simply separated by commas. The multiple rules are compiled into a single transducer which performs the desired mapping.

```
xfst[0]: clear stack
xfst[0]: read regex [ b -> a , a -> b ] ;
xfst[1]: apply down abba
baab
```

For rules having contexts, the rule separator for parallel compilation is not the single comma but the double comma; this complication is required because even a single rule can have multiple left-hand-sides or contexts, separated by single commas (see Section 3.5.2). The following rather artificial example illustrates parallel rules with contexts, separated by a double comma.

```
xfst[0]: clear stack
xfst[0]: read regex
[ b -> a || .#. s ?* _  ,, a -> b || _ ?* e .#. ] ;
xfst[1]: apply down sabbae
sbaabe
```

In most practical cases, ordered cascades of rules are sufficient and are more efficiently compiled; you should therefore avoid writing parallel rules if you can.

Parallel rules must share the same template, i.e. all must be of the general form `[ A -> B ]`, or all of the form `[ A -> B || L _ R ]`, etc. Unconditional replace rules (rules without contexts) to be compiled in parallel are separated by a single comma. Conditional replace rules (having contexts) to be compiled in parallel are separated by a double comma.

Ordering Rule Networks on The Stack

Consider the case where the user intends to compose a rule network below a lexicon network, but mistakenly pushes the lexicon network on The Stack first:

```
xfst[0]: clear stack
xfst[0]: define Cons b|c|d|f|g|h|j|k|l|m|n|p|q|r|
s|t|v|w|y|z ;
xfst[0]: read regex
[ {kick} | {try} | {bore} ]
[ %+Prog:{ing} | %+Pres3PSg:s | %+Past:{ed} | %+Bare:0 ] ;
xfst[1]: read regex
[ y -> i || Cons _ e d .#.  ,,  y -> i e || Cons _ s .#. ]
.o.
e -> 0 || Cons _ [ {ing} | {ed} ] ;
xfst[2]: print stack
0: 17 states, 222 arcs, Circular.
1: 14 states, 18 arcs, 12 paths.
```

As shown by **print stack**, these commands leave two networks on The Stack, the rule network above the lexicon network, which is the wrong order if one then intends to call **compose net** to compose the rules on the bottom of the lexicon. The solution here is to call **turn stack**, which reverses the order of all the networks stored on The Stack. The new output of **print stack** shows that the order of the networks has indeed been reversed, and now **compose net** will yield the desired result.

```
xfst[2]: turn stack
xfst[2]: print stack
0: 14 states, 18 arcs, 12 paths.
1: 17 states, 222 arcs, Circular.
xfst[2]: compose net
xfst[1]: print lower-words
trying
try
tries
tried
```

```
kicks
kick
kicking
kicked
bored
boring
bore
bores
```

If you have more than two networks on the stack and want to turn the whole stack upside down, use **turn stack**.

3.5.4 More Exercises

Southern Brazilian Portuguese Pronunciation Exercise

Capturing Phonemics vs. Orthography When linguists have written finite-state morphology systems at **Xerox**, the practical goal has usually been to map between an abstract lexical level, wherein the strings consist of a canonical baseform plus specially defined multicharacter symbol tags, and a surface level, where the strings are words written in a standard orthography. For example, one of the analyses of the Portuguese surface word "`canto`", using a large Portuguese lexical transducer built at **XRCE**, is "`cantar+Verb+PresInd+1P+Sg`", having the baseform "`cantar`" (used by convention as the citation form in traditional printed dictionaries) and the tags `+Verb`, `+PresInd` (present indicative), `+1P` (first person), and `+Sg` (singular). The multicharacter-symbol tags were chosen and defined by the linguists who wrote the system; there's nothing mysterious or sacred about them. By **Xerox** convention, the lexical side is always represented graphically on the upper side of the relation with the surface side on the lower side.

```
Lexical:  cantar+Verb+PresInd+1P+Sg
Surface:  canto
```

The emphasis on generating and analyzing orthographical words reflects the fact that **Xerox** researchers deal with natural language mainly in the form of online written texts. However, traditional rewrite rules were most often used by linguists attempting to capture the phonological, rather than orthographical, facts of a language. Replace rules, which closely resemble the traditional rewrite rules, are also well suited for formalizing phonological alternations which may be only partially or awkwardly reflected in the standard orthography.

For this exercise, the task is to create a cascade of rules that maps from orthographical strings in Portuguese (this will be the lexical side) down to strings that represent their pronunciation (this will be the surface side). There will not be a lexicon. A sample mapping of written "`caso`" to spoken "`kazu`" looks like this, with, by convention, the lexical string on top and the surface string on the bottom.

```
Lexical:   caso
Surface:   kazu
```

After studying the facts listed below, you have a choice of approaches:

1. You can write your grammar as a single, monolithic regular expression, in a regular-expression file named something like `port-pronun.regex`. You would then compile it into a network and push it on The Stack by entering **read regex < port-pronun.regex**.

2. You can write your grammar as an **xfst** script, defining variables and then using the variables in subsequent regular expressions (see Section 3.4.2). The script, named something like `port-pronun.xfst` or perhaps `port-pronun.script`, would then be run by entering **source port-pronun.xfst**. The script should leave the final network on The Stack to allow immediate testing.

When writing grammars involving cascades of rules, it is usually more convenient to write the grammar in the form of a script, first defining a variable for each rule, and then compiling a regular expression that refers to the rule variables and composes them together in the proper order. This facilitates any necessary re-ordering of rules in the cascade during debugging. Either way, the resulting network will be tested by using the **apply down** utility as in the *kaNpat* exercise. You will once again need to write a cascade of replace rules, and the relative ordering of the rules will be very important.

Standard Portuguese orthography is not always a complete guide to the pronunciation of a word (especially in the case of the letter **x** and the vowels written **o** and **e**). As usual, we will restrict and simplify the data slightly to make the solution manageable as an exercise.

The Facts to be Modeled

- The following description is based on the rather conservative pronunciation of Portuguese in Porto Alegre, Rio Grande do Sul, Brazil. Because the orthography is even more conservative, the rules will roughly characterize the phonological changes that have occurred in this one dialect since the orthography fossilized.

- The final transducer produced by your regular expression or script should generate (using **apply down**) from lexical strings like the following, written in standard Brazilian Portuguese orthography. We will limit the input to lowercase words in this exercise. The full list of test words is shown in Table 3.17, page 152.

```
casa
```

```
cimento
me
disse
peruca
simpático
braço
árvore
```

- The surface level produced by your grammar will be written in a kind of crude phonemic alphabet, with special use of the symbols in Table 3.16. Because we have limited our input words to lowercase letters, the six special characters will appear only in surface strings, never at the lexical level. The dollar sign $ character is special in regular expressions, so you will need to literalize it with a preceding percent sign (%) or with surrounding double quotes.

J	palatalized **d**, similar to the phoneme spelled **j** in English "judge"
C	palatalized **t**, similar to the phoneme spelled **ch** in "church"
$	alveopalatal sibilant, like the phoneme spelled **sh** in English "ship"
L	phoneme spelled **lh** in Portuguese *filho* (or **gli** in Italian *figlio*)
N	phoneme spelled **nh** in Portuguese *ninho* (like the French **gn** in *digne*)
R	phoneme spelled **rr** inside words, single **r** at the beginning of words

Table 3.16: Special Symbols Used on the Lower-Side to Represent Portuguese Phonemes and Allophones

- The mapping from orthography (lexical side) to pronunciation (surface side) includes the following alternations:

  - The orthographical (lexical-side) **ç** is always pronounced **/s/**; in other words, a **ç** on the upper side always corresponds to an **s** on the lower side. In these explanations, we follow the IPA convention of indicating phonemes, the lower-side symbols, between slashes. These slashes should not appear in the output strings.

```
Lexical:        braço
Surface:        brasu
```

> Hint: Your rule cascade should include a rule that looks like `[ ç -> s ]`. Warning: It will have to be ordered carefully with respect to the other rules in the cascade so that the **s** is not changed by subsequent rules in the cascade.

- The orthographical **ss** is always pronounced **/s/**. In this and following illustrations, the lexical and surface strings are lined up character pair by character pair, with the **0** (zero, also called epsilon) filling out the places where a lexical symbol maps to the empty string. These zeros are for illustration only and should not actually appear in the surface language of your transducer.

```
Lexical:        interesse
Surface:        interes0i
```

> Hint: Your cascade should include a simple rule that looks like `[ s s -> s ]`, and it will have to be ordered carefully relative to the other rules. Remember that replace rules are regular expressions and that the two symbols **s** and **s** on the left side of the arrow must be separated by white space to indicate concatenation; if you write them together as **ss**, the compiler will treat them as a single multicharacter symbol, which is not what you want.

- The orthographical **c** before **e** or **i**, or before accented versions of these vowel letters (**í**, **é** and **ê**), is always pronounced **/s/**.

```
Lexical:        cimento
Surface:        simentu
```

- The orthographical digraph **ch** is pronounced **/$/**.

```
Lexical:        chato
Surface:        $0atu
```

> Your grammar should include a rule `[ c h -> %$ ]`. The literalizing percent sign `%` is required because the dollar sign `$` is a special character in **xfst** regular expressions. Alternatively, one can literalize the dollar sign by putting it inside double quotes, i.e. `[ c h -> "$" ]`.

- Elsewhere (i.e. not **ch**, and not **ci**, **ce**, **cí**, **cé** or **cê**), orthographical **c** is always pronounced **/k/**.

```
Lexical:        casa
Surface:        kaza
```

No **c** should appear in surface strings.

- The orthographical digraph **lh** is realized as **/L/**.

```
Lexical:        filho
Surface:        fiL0u
```

- The orthographical digraph **nh** is realized as **/N/**.

```
Lexical:        ninho
Surface:        niN0u
```

Remember that the zeros shown in these examples are for illustration only and should not appear in your real output strings.

- Elsewhere, **h** is silent and is simply realized as **0** (zero, the empty string).

```
Lexical:        homem
Surface:        0omem
```

- The orthographical digraph **rr** is always realized as **/R/**. Also, the single **r** at the beginning of a word is always realized as **/R/**. Elsewhere, **r:r**, i.e. lexical **r** is realized as **/r/**.

```
Lexical:        carro   rápido  caro    cantar
Surface:        kaR0u   Rápidu  karu    kantar
```

- The unaccented **e** is pronounced **/i/** at the end of a word, and when it appears in the context between **p** and **r** at the beginning of a word; e.g.

```
Lexical:        peruca          case
Surface:        piruka          kazi
```

An unaccented **e** is also pronounced **/i/** before an **s** at the end of a word. Elsewhere **e:e**.

```
Lexical:    cases
Surface:    kazis
```

– An unaccented **o** is pronounced **/u/** at the end of a word.

```
Lexical:    braço       caso
Surface:    brasu       kazu
```

An unaccented **o** is also pronounced **/u/** before an **s** at the end of a word. Elsewhere **o:o**.

```
Lexical:    braços
Surface:    brasus
```

– A single **s** is pronounced **/z/** when it appears between two vowels.

```
Lexical:    camisa      case
Surface:    kamiza      kazi
```

Elsewhere **s:s** (but see above where `s s -> s`).

– A word-final **z** is pronounced as **/s/**.

```
Lexical:    vez
Surface:    ves
```

Elsewhere, **z:z**.

– A **d** is pronounced **/J/** when it appears before a surface phoneme **/i/**. (N.B. This change occurs in the environment of any surface **/i/**, no matter what that surface **/i/** may have been at the lexical level.) Elsewhere **d:d**.

```
Lexical:    disse    verdade    paredes
Surface:    JisOi    verdaJi    pareJis
```

– A **t** is pronounced **/C/** when it appears before a surface phoneme **/i/**. (N.B. This change occurs in the environment of any surface **/i/**, no matter what that surface **/i/** may have been at the lexical level.) Elsewhere **t:t**.

```
Lexical:    tio        partes
Surface:    Ciu        parCis
```

– The vowels are **a, e, i, o, u, á, é, í, ó, ú, ã, õ, â, ê, ô, ü** and **à**. All lexical symbols map to themselves on the surface level by default.

Testing Portuguese Pronunciation Write a set of replace rules that performs the mappings indicated. As in the *kaNpat* example, the rules should be organized in a cascade, with the composition operator (`.o.`) between the rules. Be very careful about ordering your rules correctly; the rules cannot be expressed in exactly the same order as the facts listed just above.

If you write your grammar as a monolithic regular expression, then you can compile it by entering

```
xfst[0]: read regex < regex_filename
```

If you write the grammar as an **xfst** script (see Section 3.3.3), then you will run the script by entering

```
xfst[0]: source script_filename
```

It is probably preferable to write the grammar as a script.

Using **apply down**, you should be able to handle all the examples in Table 3.17, entering the lexical or upper-side string in each case and getting back the surface or lower-side string. The zeros representing the empty strings in examples above are not shown here and should not appear in your output. To facilitate the testing, you should type all the input (upper-side) words into a file, called something like `mydata`, with one word to a line, and tell **apply down** to read the various input strings from that file (review Section 3.3.4).

```
xfst[1]:  apply down < mydata
```

If you have written your grammar as a script, you can even include this line at the end of the script. Be sure to test *all* the examples to ensure that your rules are really working as they should. Modify your rules, or the rule ordering, and re-test the input words until the grammar is working perfectly.

disse Jisi	peru piru	pedaço pedasu
livro livru	parte parCi	parede pareJi
sabe sabi	cada kada	simpático simpáCiku
verdade verdaJi	casa kaza	braço brasu
chato $atu	vermelho vermeLu	gatinho gaCiNu
filhos fiLus	luz lus	case kazi
braços brasus	partes parCis	paredes pareJis
me mi	antes anCis	ninhos niNus
rápido Rápidu	carro kaRu	caro karu
cantar kantar	bicho bi$u	diferentes JiferenCis

Table 3.17: Test Data for the Portuguese Pronunciation Exercise. Type the upper-side strings into a separate file, called something like `mydata`, with one word to a line, to facilitate repeated testing with **apply down**.

The Bambona Language Exercise

For students who need a challenge, our next exercise concerns vowel changes in the mythical Bambona language. This is a more difficult exercise, involving the definition of a lexicon relation and a rule-based relation that are then composed together.

Some Preliminaries Recall that in regular expressions, strings of characters written together such as `dog` are treated as single symbols called multicharacter symbols. By **Xerox** convention, we employ multicharacter symbols such as `+Verb`, `+Noun`, `+Sg` (singular), and `+Pl` (plural) (or `[Noun]`, `[Verb]`, etc.) in our networks to convey part-of-speech and other featural information. The punctuation marks included by **Xerox** convention in the spelling of the multicharacter symbols make the output more readable, but because the plus sign and square brackets (and, indeed, almost all the punctuation characters) are special characters in regular expressions, we need to literalize them with a preceding percent sign, e.g. `%+Noun` or `%[Noun%]`, or by placing the entire multicharacter symbol in double quotes, e.g. `"+Noun"` or `"[Noun]"`.

Recall also that the crossproduct operator in expressions such as `[Y .x. Z]` designates the relation wherein `Y` is the upper-side language, `Z` is the lower-side language, and each string in each language is related to all the strings in the other language. Similarly, the colon operator as in `a:b` denotes crossproduct, but it has higher precedence than `.x.` and higher precedence than concatenation. The expression `[%+Noun:0]`, which is equivalent to `[%+Noun .x. 0]` therefore denotes the relation that has the single symbol +Noun on the upper side and the empty string on the lower side. The expression `[%+Pl:{il}]`, equivalent to `[%+Pl .x. [ i l ]]`, denotes the relation that has the single symbol `+Pl` on the upper side and the concatenation `[i l]` on the lower side. You will need to use such notations in this exercise. Be aware that multicharacter symbols cannot appear inside curly braces.

> The curly brace notation, as in `{ing}` and `%+Pres-Part:{ing}`, can contain only simple alphabetic symbols, not multicharacter symbols. If `^U` is a multicharacter symbol, do not try to write `%+Tag:{^Uabc}`; rather write `%+Tag:[%^U a b c]` or `[ %+Tag .x. %^U a b c ]`.

The Facts Here are the linguistic facts of Bambona:

1. There are seven vowels in Bambona.

i		u
e		o
é		ó
	a	

where e is a mid-high front vowel, é is a mid-low front vowel, o is a mid-high back vowel and ó is a mid-low back vowel. There are no length distinctions.

2. We will limit this example to Bambona nouns. Nouns always start with a root, which is usually based on the pattern CVC or CVCC. E.g.

mad	"book"
nat	"house"
posk	"girl"
rip	"arm"
kuzm	"cooking pot"
karj	"cellular telephone"
zib	"enemy"
lér	"wild pig"
kóp	"nuclear reactor"
sob	"dentist"
mélk	"stone axe"
rut	"integrated circuit"

> Hint: The noun-root sublanguage in your regular expression should simply be the union of all the noun roots listed above. Do not try to handle all possible Bambona roots in this exercise.

3. The English glosses listed for the roots and for the suffixes below are provided for informational purposes only. Do not try to include the glosses in your network.

4. After the root, nouns can continue with an optional suffix from the following list:

ak	"pejorative"
et	"diminutive"
ig	"augmentative"

A maximum of one of these three suffixes can appear in a word. In regular expressions, an expression is made optional by surrounding it with parentheses or by unioning it with the empty string: thus the expression `(a)` is equivalent to `[a | []]`. This new sublanguage should eventually be concatenated onto the end of the noun-root sublanguage. Use the multicharacter tags `+Pej`, `+Dim` and `+Aug` as appropriate on the lexical side.

> Hint: One portion of your regular expression should look like this: `[ %+Pej:{ak} | %+Dim:{et} | %+Aug:{ig} ]`.

5. Next can come, optionally, a single suffix indicating the speaker's confidence from the following list:

```
izm             "obvious"
ubap            "probable"
ópot            "alleged"
```

Use the tags `+Obv`, `+Prob` and `+All` on the lexical side.

6. Next can come, optionally, a single suffix that indicates number.

```
il              "plural"
ejak            "paucal (a few)"
```

Overt number marking is optional in Bambona, even when the referent of the word is obviously plural or paucal. In fact, in the syntax, when the noun is preceded by another word that states a number explicitly, the number is never marked in the noun itself. There is no suffix to mark the singular (really unmarked number) reading explicitly, but construct the grammar so that each noun is marked either `+Pl` or `+Pauc`, or `+NoNum` (which is realized as zero/epsilon on the lower side); this effectively makes a number marking obligatory.

7. After the number marking must come one of the following case suffixes (or nothing for nominative/unmarked).

```
0       nominative/unmarked
am      accusative      (marks direct objects)
ad      dative          (like English "to")
it      ablative        (like English "from")
ek      benefactive     (like English "for")
```

```
ozk      genitive          (like English "of")

ém       inessive          ("in"/"at")
ót       elative           ("from/out of")

ep       comitative        ("with")
```

The following details are a bit tricky, so read carefully: When overt prepositions are used in the sentence, the noun may be unmarked for case. The accusative, dative, ablative and benefactive (and nominative) cases, if present, always end the word. The genitive *ozk* can end a word, or it can optionally be followed by the *on* suffix which denotes inalienable possession (like your own arms and legs, as opposed to a Sony Walkman or Saab 9000 or the severed limbs of an enemy). Remember that optionality in regular expression is indicated by surrounding an expression with parentheses. The inessive *ém* and the elative *ót* can optionally be followed with a suffix *el*, a general intensifier in the language, resulting in the meanings "into" and "out of", respectively. The comitative *ep* can be followed optionally with the suffix *eg*, denoting the negative, yielding the reading "without". The suffix *eg* is also used in the verb system, not treated here. Use the following tags on the lexical side:

```
+Nom +Acc +Dat +Abl +Ben +Gen +Ine +Ela +Com
+Inalien +Int +Neg
```

> Read the preceding description carefully and picture the possible suffix sequences before writing your regular expressions. Graphing sometimes helps. Published natural-language grammars are seldom this explicit in specifying the morphotactic possibilities, requiring you to read between the lines, study the examples, and, ideally, do some original field-work with native informants. Finite-state tools give you a way to encode the linguistic facts about your language, but they cannot do the linguistics for you.

8. There are two ways of going about writing a finite-state description of Bambona morphotactics: One way is to write it as a single regular expression (a "regular-expression file") to be compiled with **read regex** < ***filename***; the other is to write it as an **xfst** script (a "script file") to be executed using the **source** command. Review the distinction between regular-expression files and scripts (Section 3.3.3) if this is not clear.

Lexical	nat+Pej+NoNum+Nom
Surface	natak
Lexical	nat+Dim+Obv+NoNum+Com+Neg
Surface	natetizmepeg
Lexical	sob+Dim+All+Pauc+Gen
Surface	sobetópotejakozk
Lexical	lér+Aug+All+Pauc+Ine+Int
Surface	lérigópotejakémel
Lexical	kóp+Aug+Pl+Ela
Surface	kópigilót
Lexical	rip+Dim+Pl+Gen+Inalien
Surface	ripetilozkon

Table 3.18: Lexical/Surface Pairs from the Lexicon Alone. Note that these surface forms are still just intermediate forms, awaiting correction via the application of alternation rules.

It is probably best to write the grammar as a script. Either way, the lexical (upper) side should contain alphabetic symbols and multicharacter tags, while the surface (lower) side contains only alphabetic symbols.

If you write the grammar as a monolithic regular expression in the file `bambona-lex.regex` you would compile it with the command

```
xfst[0]: read regex < bambona-lex.regex
xfst[1]: save stack bambona-lex.fst
```

Save the resulting network, using **save stack**, as the binary file `bambona-lex.fst`. If you write your grammar as a script file named `bambona-lex.xfst`, you would run the script with

```
xfst[0]: source bambona-lex.xfst
```

and the script itself should contain the command to save The Stack out to file `bambona-lex.fst`.

The lexicon network by itself is a transducer that should encode a relation that includes the ordered pairs of words shown in Table 3.18; test to make sure (using **apply up** and **apply down** as appropriate). Be warned that some of these lower-side forms from the lexicon are not quite right yet—this is still just the first step of the solution.

natak	"awful house(s)"
natetizmepeg	"without (the) obvious little house(s)"
sobetópotejakozk	"of (the) few alleged dentists"
lérigópotejakémel	"into (the) few alleged wild-pigs"
kópigilót	"from (the) big nuclear reactors"
ripetilozkon	"of (the) little arms"

Table 3.19: Glosses for Some Intermediate Bambona Words. Again, these examples are intermediate words formed by simple concatenation of morphemes; they must subsequently be corrected via the application of suitable alternation rules.

The surface forms in Table 3.18 are best thought of as intermediate steps on the way to a final solution. The final mapping of these intermediate forms to the real surface forms will be done later by the application of alternation rules. The words in Table 3.18 might be glossed as in Table 3.19, but do not try to make your transducer produce anything like a translation.

9. In actual surface words of Bambona, the consonants **p**, **t**, and **k** are never followed by the front vowels **i**, **e**, or **é** but always by their corresponding back vowels **u**, **o** and **ó** respectively; and the symbol pairs **i:u**, **e:o** and **é:ó** occur only after **p**, **t** or **k**. The vowel **a** is neutral in this phenomenon. Therefore the example "`rip+Dim+Pl+Gen+Inalien`" conceptually has three levels:

```
Lexical:          rip+Dim+Pl+Gen+Inalien
Intermediate:     ripetilozkon
Surface:          ripotulozkon
```

where the intermediate **e** after **p** is realized in the surface string as **o**, and the **i** after **t** is realized as **u**. Note that the mapping from the lexical level to the intermediate "`ripetilozkon`" is already performed by the lexicon transducer itself. Thus, for the lexicon grammar, the analysis string "`rip+Dim+Pl+Gen+Inalien`" is the lexical or upper side, and the string "`ripetilozkon`" is the surface or lower side.

```
Lexical (lexicon FST):  rip+Dim+Pl+Gen+Inalien
Surface (lexicon FST):  ripetilozkon
```

For the alternation rules, the string "`ripetilozkon`" will be the upper side, and the final form "`ripotulozkon`" will be the lower side. The

rule transducer performs the mapping between intermediate, not yet correct, strings like "ripetilozkon" and final correct surface strings like "ripotulozkon".

```
Lexical (rule FST):   ripetilozkon
Surface (rule FST):   ripotulozkon
```

When the rule transducer is composed on the bottom of the lexicon transducer, the resulting transducer will map directly from the strings on the upper side of the lexicon transducer to the strings on the lower side of the rule transducer. The intermediate level simply disappears in the process of the composition. Don't proceed until you understand this. See Sections 1.5.3 and 2.3.1 for a review of composition.

```
Lexical (final FST):  rip+Dim+Pl+Gen+Inalien
Surface (final FST):  ripotulozkon
```

In a separate file `bambona-rul.regex` write a regular expression consisting of replace rules that perform the vowel alternations described above. Compile the rules in **xfst** using the command `read regex < bambona-rul.regex` and save the compiled network to file as `bambona-rul.fst`.

```
xfst[0]: clear stack
xfst[0]: read regex < bambona-rul.regex
xfst[1]: save stack bambona-rul.fst
```

Test the rules separately by using **apply down**, entering strings from the lower side of the lexicon network (e.g. "ripetilozkon" and other lower-side examples from Table 3.18) to see if the rules produce the correct surface form. Use the additional examples in Table 3.20 to test your rules (perform **apply down** on the intermediate string, and you should get the surface string).

Intermediate	ripetilozkon
Surface	ripotulozkon
Intermediate	kópizmepeg
Surface	kópuzmepog
Intermediate	poskigizmilek
Surface	poskugizmilek
Intermediate	natetópotilótel
Surface	natotópotulótol

Table 3.20: Intermediate to Surface Mappings, Showing Alternations

Compose the Rules under the Lexicon Having written, compiled and tested the lexicon and the rules separately, the next step is to compose the rules under the lexicon and test the result. There are several ways to do this.

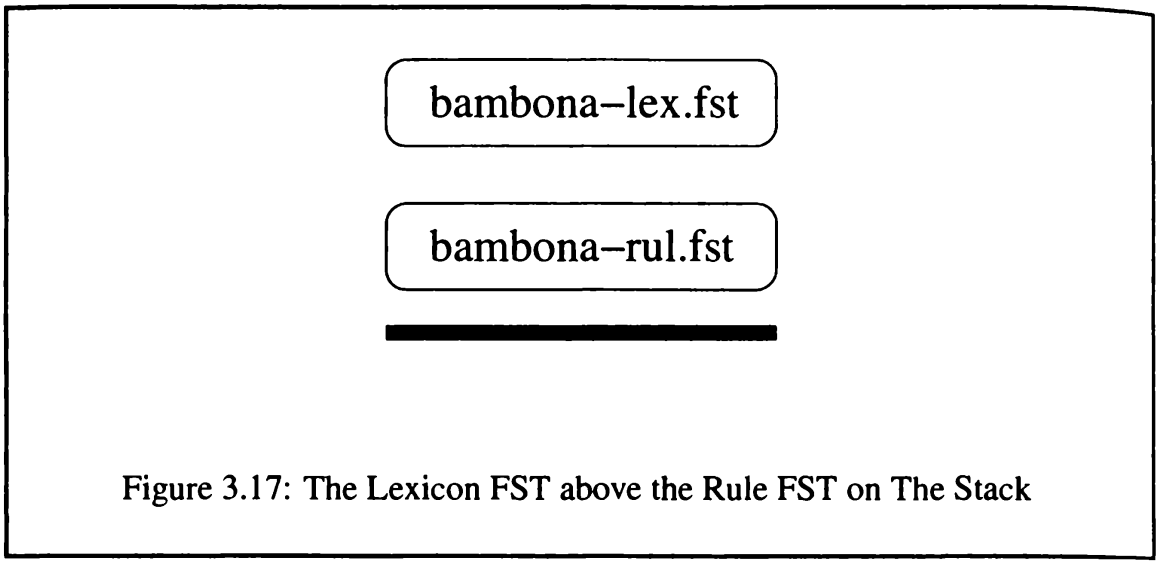


Figure 3.17: The Lexicon FST above the Rule FST on The Stack

Composition on The Stack Both `bambona-lex.fst` and `bambona-rul.fst` are saved to file, so they can be read back onto The Stack, in the correct order, and composed on The Stack. In order to compose the rule transducer under the lexicon transducer, it must appear under the lexicon transducer on The Stack; therefore it must be pushed onto The Stack first. The following commands will leave The Stack as shown in Figure 3.17.

```
xfst[0]: clear stack
xfst[0]: load stack bambona-rul.fst
xfst[1]: load stack bambona-lex.fst
```

There should be two networks on The Stack. The final network is then created by calling **compose net**.

```
xfst[2]: compose net
xfst[1]: save stack bambona.fst
```

This should leave one network on The Stack, ready for testing. Also save a copy to file `bambona.fst` as shown.

Composition in a Regular Expression The binary networks saved in the files `bambona-lex.fst` and `bambona-rul.fst` can also be composed using a regular expression. The regular-expression notation `@"filename"` retrieves the value of the binary network stored in `filename`. One can therefore compose the two networks with the following commands:

Lexical	nat+Pej+NoNum+Nom
Surface	natak
Lexical	nat+Dim+Obv+NoNum+Com+Neg
Surface	natotuzmepog
Lexical	sob+Dim+All+Pauc+Gen
Surface	sobetópotojakozk
Lexical	lér+Aug+All+Pauc+Ine+Int
Surface	lérigópotojakómel
Lexical	kóp+Aug+Pl+Ela
Surface	kópugilót
Lexical	rip+Dim+Pl+Gen+Inalien
Surface	ripotulozkon

Table 3.21: Lexical/Surface Pairs in the Final Bambona Lexical Transducer

```
xfst[0]: clear stack
xfst[0]: read regex @"bambona-lex.fst"
.o.
@"bambona-rul.fst" ;
xfst[1]: save stack bambona.fst
```

Once again, this should result in one network being left on The Stack, being the composition of the rules onto the bottom of the lexicon.

Test this lexical transducer with all the examples in Table 3.21, and make up some of your own. Use both **apply up** and **apply down** as appropriate. Note that the mapping between the upper-side and lower-side strings is now done in one step; the intermediate string levels have literally disappeared in the composition.

Bambona Using a Single Script File

The Bambona exercise above involved writing two separate grammar files, compiling or sourcing them separately into networks, and composing them together. Using an **xfst** script, the whole grammar, plus final testing, can be written together in one file.[8]

In real applications, it is often necessary to edit your lexicon and rules many times before you get everything working correctly, and every time you change the regular-expression files you will need to re-execute all the **xfst** commands required to rebuild the final network and retest it. Retyping these commands manually can be tedious and error-prone. It is usually preferable to put all the necessary **xfst**

[8]The use of makefiles, exemplified on `http://www.fsmbook.com/`, is an even more general and powerful technique for compiling and testing complex systems.

commands in a script file (see Section 3.3.3) and then use the **source** command to re-execute the script after each change of the grammar.

Lexicon and rule grammars themselves can also be made much more readable by writing them in script files. For example, Bambona noun roots can be defined as a sublanguage using the **define** command, written in a script file like this:

```
define NROOT [ {mad} | {nat} | {posk} | {rip} | {kuzm} |
               {karj} | {zib} | {lér} | {kóp} |
               {sob} | {mélk} | {rut}
             ] ;
```

Recall that the regular expression {mad} is equivalent to [m a d]. Once defined in this way, the name **NROOT** can be used in subsequent regular expressions. Similarly, the class of Bambona suffixes *ak/et/ig* can be defined in a script file with the following statement:

```
define SUFF1 [ %+Pej:{ak} | %+Dim:{et} | %+Aug:{ig} ] ;
```

After each of the morpheme classes has been defined and named in this manner, a final definition can concatenate the named sublexicons into the preliminary full lexicon:

```
define Lexicon NROOT (SUFF1) (SUFF2) SUFF3 SUFF4 ;
```

In the expression above, **SUFF1** and **SUFF2** are parenthesized because they are optional suffixes in Bambona. Replace rules can also be defined, named, and composed with the lexicon via commands in the script file.

Now redo the Bambona exercise, writing everything in a single **xfst** script file that defines all the sublexicons and rules and then performs all the operations necessary to build the final Bambona network, test it, and save it to file as `bambona2.fst`. This `bambona2.fst` should be equivalent to the `bambona.fst` you built and saved previously using multiple source files.

To test the equivalence of two networks, load them onto an **xfst** stack and invoke the **test equivalent** command.

```
xfst[0]: clear stack
xfst[0]: load stack bambona.fst
xfst[1]: load stack bambona2.fst
xfst[2]: test equivalent
```

xfst will then indicate if the two networks are equivalent or not.

The Monish Language Exercise

The following exercise is based on the fictional Monish language.

Front	Back
i	u
e	o
é	ó
ä	a

Table 3.22: The Eight Vowels of Monish

ruuzod	"walk"
tsarlók	"drink"
ntonól	"gamble"
bunoots	"fish"
vésiimb	"invent facts for sociologists"
yääqin	"drink (alcoholic)"
feséény	"steal/borrow"

Table 3.23: A Few Monish Roots

1. Monish has eight vowels, divided into Front and Back groups, as shown in Table 3.22.

2. Monish exhibits a simple kind of vowel harmony: all of the vowels in a word must be either front vowels or back vowels. So one never finds front vowels and back vowels mixed in the same word.

3. Morphotactically, Monish words always begin with a root and always have suffixes; roots are BOUND morphemes, meaning that they are not valid words by themselves. We will concern ourselves with a subset of the Monish verb system.

4. Table 3.23 shows a few Monish verb roots. The English glosses given here and below are for information only and should not be included in any way in your morphological analysis. Write your grammar so that it accepts only words that are based on the listed roots.

5. Monish suffixes in the lexicon contain underspecified vowels that are realized on the surface as front vowels or as back vowels depending on the overall harmony established by the vowels in the root. The following representation scheme is recommended, but you can use another if you design

and document it well.

- Monish suffixes can also contain the following four underspecified vowel ARCHIPHONEMES or MORPHOPHONEMES:

  ```
  ^U ^O ^Ó ^A
  ```

- **^U** is the underspecified high vowel; it must be realized on the surface as either **u**, in back-vowel words, or as **i** in front-vowel words.
- Similarly, **^O** is the underspecified mid-high vowel; it must be realized on the surface as either **o**, in back-vowel words, or as **e** in front-vowel words.
- The underspecified mid-low vowel **^Ó** is realized as either **é** (front) or **ó** (back).
- The underspecified **^A** is realized as either **ä** (front) or **a** (back).

6. All Monish verbs begin with a root. After the root comes an optional intensifier that at the lexical level is spelled

   ```
   ^U^Uk
   ```

   consisting of two underspecified high vowels and a **k**. Use the multicharacter-symbol tag `+Int` at the lexical level for this optional intensifier morpheme.

> Note that you cannot use the curly braces, as in `%+Pl:{ed}`, to map `+Int` to a string of symbols like "`^U^Uk`"; the problem is that multicharacter symbols cannot appear inside the curly-brace notation. Instead, use `[ %+Int .x. [ %^U %^U k ] ]`, `[ %+Int : [ %^U %^U k ] ]` or `[ %+Int : [ "^U" "^U" k ] ]`. See Section 3.2.5, page 103.

7. Next comes a single required aspect marker, one of those shown in Table 3.24. Use the tags `+Perf`, `+Imperf`, and `+Opt` on the lexical side.

8. After the aspect marker can come an optional suffix, shown in Table 3.25, conveying the confidence of the speaker.

   Use the tags `+True`, `+Belief`, `+Doubt`, and `+False` at the lexical level.

Lexical Tag	Morphophonemic Form	Semantics
+Perf	^On	"perfective"
+Imperf	^Ómb	"imperfective"
+Opt	^Udd	"optative"

Table 3.24: Monish Aspect Suffixes

Lexical Tag	Morphophonemic Form	Semantics
+True	^Ank	"true" (the speaker invites you to believe that he/she witnessed the event personally)
+Belief	^A^Av^Ot	"belief" (second-hand, from a witness judged reliable)
+Doubt	^U^Uz	"credulous" (second-hand, from rumor or an unreliable witness)
+False	^Óq	"false" (negative)

Table 3.25: Monish Confidence Suffixes

Lexical Tags	Morphophonemic Form	Semantics
+1P +Sg	^A^Ab^A	"I"
+2P +Sg	^Óm^A	"you (sing)"
+3P +Sg	^Uvv^U	"he/she"
+1P +Pl +Excl	^A^Ab^Or^A	"we (exclusive)"
+1P +Pl +Incl	^A^Ab^Ug^A	"we (inclusive)"
+2P +Pl	^Óm^Or^A	"you (plural)"
+3P + Pl	^Uvv^Or^U	"they"

Table 3.26: Monish Person-Number Suffixes

9. Next the seven suffixes shown in Table 3.26 convey person and number. Exactly one is required in each verb. Use the tags `+1P`, `+2P`, `+3P`, `+Sg`, `+Pl`, `+Incl`, and `+Excl` on the lexical level.

10. Visualize the morphotactics of the words before writing a regular-expression grammar. Diagramming often helps.

11. Use comments in your source files to document the meaning of the tags. They may seem obvious now, but they won't be in a week. Comments in regular-expression files and in **xfst** scripts start with an exclamation mark or pound sign (#) and extend to the end of the line.

```
! This is a comment
# This is another comment
```

12. You now have a couple of choices: either

    - Create a regular-expression lexicon file called `monish-lex.regex`, compile it using `read regex < monish-lex.regex` and save the result as `monish-lex.fst` using **save stack**; or
    - Define the lexicon using a script file and run it using the **source** command. This script file should save the final result in the file `monish-lex.fst`.

    Either way, the lexicon file will describe the morphotactics of Monish words, and the resulting network should have baseforms and tags on the upper side, and baseforms followed by abstract suffixes on the lower side. Assuming that you choose to write your grammar as an **xfst** script, you will have definitions like the following:

```
define Root  [
              r u u z o d       |
              t s a r l ó k     |
              n t o n ó l       |
              b u n o o t s     |
              v é s i i m b     |
              y ä ä q i n       |
              f e s é é n g
             ] ;

define Suff1 %+Int .x. [ %^U %^U k ] ;

define Suff2 [ %+Perf .x. [ %^O n ]  ]  |
             [ %+Imperf .x. [ %^Ó m b ] ]  |
             [ %+Opt .x. [ %^U d d ] ] ;
```

13. After writing your lexicon grammar, create a separate rule file with a name like `monish-rul.regex` that describes the alternations required for Monish. Compile it, and save the result as `monish-rul.fst`. Alternatively, if

Lexical	yääqin+Perf+2P+Pl
Surface	yääqinenémerä
Lexical	feséén̈g+Opt+False+1P+Pl+Incl
Surface	feséén̈giddéqääbigä
Lexical	bunoots+Int+Perf+2P+Sg
Surface	bunootsuukonóma
Lexical	tsarlók+Opt+False+1P+Sg
Surface	tsarlókuddóqaaba
Lexical	ntonól+Imperf+1P+Pl+Excl
Surface	ntonólómbaabora

Table 3.27: Some Pairs of Words in the Final Monish Transducer

you wrote the lexicon in a script, simply expand the script to define the rules as well. The trick is to write rules that realize each underspecified vowel as front or back depending on the frontness or backness of the vowels of the root.

14. Compose the rule network under the lexicon network, and save the result as `monish.fst`, i.e. perform the following by hand or add the commands to your script file.[9]

```
xfst[0]:  clear stack
xfst[0]:  read regex @"monish-lex.fst"
.o.
@"monish-rul.fst" ;
xfst[1]:  save stack monish.fst
```

15. Test the resulting system. It should analyze and generate examples like those shown in Table 3.27.

16. Test for bad data as well. Your system should not be able to analyze input strings like "`yääqinenémorä`", because it contains **o** in a front-harmony word; and it should similarly fail to analyze "`tsarlókuddóqaabe`" because it contains **e** in a back-harmony word.

The Monish Guesser

The Monish lexicon above contains only seven roots and can analyze only words that are based on these known roots. In a real development project, you will typi-

[9] You can, of course, also push the two compiled networks on The Stack and compose them using **compose net**; in this case, remember that `monish-rul.fst` must be pushed onto The Stack first so that it is underneath `monish-lex.fst`.

cally add tens of thousands of roots to the lexicon before you have a commercially viable system.

In some applications you may want to build an auxiliary analyzer or GUESSER that tries to analyze words with roots that are not in the dictionary. To do this, you need to define a regular expression that matches all phonologically possible roots in the language, and then use that in place of the seven unioned roots in the original example. This does not mean that the system should accept just anything as a root; we know, or at least can intelligently guess, some facts about Monish roots that should be respected in our guesser:

1. A Monish root must contain at least one vowel.

2. A Monish root cannot contain both a front vowel and a back vowel.

3. In addition, we can assume that the underspecified morphophonemes **^U**, **^O**, **^Ó** and **^A** cannot appear in a root; they appear only in suffixes.

Make a copy of your script-based grammar of Monish, modify the definition of roots to cover all possible roots (which should of course cover all the known ones), rebuild the system, and save the resulting network as `monish-guesser.fst`. The trick is to define, using a regular expression, your roots as the set of all strings that either

1. Contain a back vowel plus any number of other symbols that are not front vowels and not morphophonemic vowels, or

2. Contain a front vowel plus any number of other symbols that are not back vowels and not morphophonemic vowels.

The containment operator `$` should come in handy for this exercise. Test the result by reanalyzing the good and bad surface examples given above on page 167; it should return analyses for the good words and not for the words that have illegal combinations of front and back vowels. In addition, analyze the following strings which cannot be analyzed by `monish.fst`.

```
viinémbivveri
monobuddóqaabora
minebiddéqääberä
```

In each case, your guesser should propose an analysis that identifies a potential root. The following examples should be analyzed to show two possible roots; confirm this and explain the results.

```
viiniikenänkémä
bunootsuukonóma
```

3.5.5 More Replace Rules

Replace rules are a very rich, and still growing,[10] set of formalisms for specifying transducer networks. We will first look at the family of right-arrow rules, progressing later to left-arrow and double-arrow rules.

Right-Arrow Rules

Double-Vertical-Bar Rules The most commonly used form, the right-arrow, double-vertical-bar rules, were introduced above in Section 3.5.2. You will recall that these rules are based on the following template

```
A -> B || L _ R
```

where A, B, L and R are regular expressions denoting languages (not relations), and L and R are optional. The overall rule denotes a relation. The left context L is extended by default on the left, and the right context R on the right, with `?*`, the universal language. The `.#.` notation may be used to override this default and indicate the beginning and/or end of a word. These notational conventions will hold for the other subtypes of replace rules to be presented.

But before we move on to the more esoteric rules, it is important to explain the || operator, which indicates, in a right-arrow rule, that the contexts L and R must both match on the upper-side of the relation. Relations and the transducers that encode them are of course bi-directional, but the upper side of a right-arrow rule is most naturally viewed as the "input" side; i.e. right-arrow rules are written with a notational bias toward generation. This semantics, where the sequence L A R must appear in the upper side or input word for the rule to match and apply, corresponds to traditional rewrite rules. Some significant finite-state systems require no other subtypes of rules.

Double-Slash Rules Right-arrow, double-slash rules are built on the following template:

```
A -> B // L _ R
```

where A, B, L and R have the same restrictions and possibilities as for right-arrow, double-vertical-bar rules. The overall rule again denotes a relation, but here the // operator indicates that the left context L must match on the lower or output side, while the right context R must match on the upper or input side.

For some examples, // rules appear to "generate their own left context" and have been found useful for modeling vowel harmony, which conceptually moves left-to-right inside a word. As we saw in the Monish exercise (Section 3.5.4), the simplest kind of vowel harmony requires that all the vowels in a word be either

[10] For the latest information, see `http://www.fsmbook.com/`.

```
xfst[0]: define FrontVowel i | e | é | ä ;
xfst[0]: define Rules %^U -> i,
%^O -> e, %^Ó -> é, %^A -> ä || $[FrontVowel] _
.o.
%^U -> u
.o.
%^O -> o
.o.
%^Ó -> ó
.o.
%^A -> a ;
```

Figure 3.18: Monish Alternation Mapping Front Vowels First

front or back. The root, which itself contains only front or back vowels, sets the harmony for the entire word.

In real life, languages with vowel harmony are often more complicated. They may allow compounding of roots, which means that front-vowel and back-vowel roots may occur together in the same word. In such cases, the harmony may change several times from left-to-right as roots are added, with the finally compounded root dictating the harmony for any following suffixes. And whereas Monish suffixes contained only underspecified morphophonemic vowels, some languages contain suffixes with surfacy fully-specified vowels that can attach to any word, regardless of the previous harmony setting, and that potentially reset the harmony for the remainder of the word (or until some other suffix re-sets the harmony again).

Monish suffixes contain the underspecified vowels **^U**, **^O**, **^Ó** and **^A**, and the simple rules in Figure 3.18 suffice to realize them correctly. That is, in such a simple vowel-harmony language, where a whole word contains only front or back vowels, the underspecified vowels are realized as front vowels if and only if the left context contains one or more front vowels. Otherwise they are realized as back vowels. The rules could just as well be written the other way, of course, first realizing the underspecified vowels as back vowels after a root containing back vowels, and then realizing any leftovers as front vowels, as shown in Figure 3.19.

In a more complicated language where the vowel harmony may change as you move from left to right, the left context may contain both front and back vowels. Any underspecified vowels must be realized according to the most recently set harmony feature, which is exemplified by the nearest vowel to the left *on the surface side*. The following example, using a double-slash rule, achieves the desired mapping:

```
xfst[0]: define FrontVowel i | e | é | ä ;
```

```
xfst[0]: define BackVowel u | o | ó | a ;
xfst[0]: define Rules %^U -> u,
%^O -> o, %^Ó -> ó, %^A -> a || $[BackVowel] _
.o.
%^U -> i
.o.
%^O -> e
.o.
%^Ó -> é
.o.
%^A -> ä ;
```

Figure 3.19: Monish Alternation Mapping Back Vowels First

```
xfst[0]: define Cons b|c|d|f|g|h|j|k|l|m|n|p|q|
r|s|t|v|w|y|z ;
xfst[0]: read regex
%^U -> i, %^O -> e, %^Ó -> é, %^A -> ä // FrontVowel Cons* _
.o.
%^U -> u
.o.
%^O -> o
.o.
%^Ó -> ó
.o.
%^A -> a ;
```

Let us suppose, for example, that a language much like Monish, but with compounding and fully specified vowels in certain suffixes, constructs lexical strings like the following:

```
Abstract word: totutike^U^Um^Aqr^O^Onotun^Uq

Morphemes:
totu     root
tike     root
^U^Um    suffix
^Aqr     suffix
^O^On    suffix
otun     suffix (N.B. the fully-specified vowels)
^Uq      suffix
```

In this example, the root *totu* starts the word in back harmony, but the compounded stem *tike* then changes it to front harmony. The following *^U^Um*, *^Aqr* and *^O^On* suffixes contain only underspecified vowels and so must agree with the current

(front) harmony. Then the *otun* suffix comes along, resetting the harmony to back with its fully-specified back vowels, and then the suffix *^Uq* must be realized according to the currently active back harmony. The mapping should be as follows (spaces line up the symbols here for comparison of the two levels but do not really appear in the surface string)

```
Lexical:  totutike^U^Um^Aqr^O^Onotun^Uq
Surface:  totutike i im äqr e enotun uq
```

Note that each underspecified vowel must be realized in accordance with the harmony of the preceding vowel on the left as it appears on the surface side. The // rule, which matches the left context on the surface side, is therefore perfectly suited for capturing this kind of vowel harmony.

Type in the rules above, generate using the lexical string

```
totutike^U^Um^Aqr^O^Onotun^Uq
```

as input, and satisfy yourself that the proper surface form is generated. Then generate from "`tikewalowo^U^Um^Aqr^O^Onetiq^Uq^Amm^A`" and satisfy yourself that the result is correct.

To further compare || and // rules, try the following:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a || a _ ;
xfst[1]: down abbbbb
aabbbb
```

Note in this example that the left-context **a** must match on the upper (input) side of the relation, and that only the first **b** has an **a** as the upper left context, so the output is "`aabbbb`". Now try the following variant with a double-slash operator.

```
xfst[0]: clear stack
xfst[0]: read regex b -> a // a _ ;
xfst[1]: down abbbbb
aaaaaa
```

The output here is "`aaaaaa`" because the first **a** in the input maps (by default) to itself on the surface, the first **b** is mapped to an **a** on the surface because it has a surface left-context of **a**, and then the remaining **b**s also map to **a**, with the rule appearing to generate its own left context. In reality, of course, the rule denotes a relation rather than some kind of algorithm, and the relation simply maps a lexical "`abbbbb`" to a surface "`aaaaaa`".

Double-Backslash Rules The right-arrow, double-backslash rules are built on the following template:

```
A -> B \\  L _ R
```

where A, B, L and R have the same restrictions and possibilities as in the previous rules. Such a rule indicates that the left context L is to be matched on the upper (input) side of the relation and the right context R is to be matched on the lower or output side of the relation. The double-backslash rule is therefore the inverse of the double-slash rule as far as context matching is concerned.

As you might expect, the double-backslash rule can appear to generate its own surface context from right to left, and it can be useful for modeling the phonological process of umlaut, which is the opposite of vowel harmony.

To compare || and \\ rules, try the following:

```
xfst[0]: clear stack
xfst[0]: read regex b -> a || _ a ;
xfst[1]: down bbbbba
bbbbaa
```

Note in this example that the right context **a** must match on the upper (input) side of the relation, and that only the last **b** has an **a** as the upper right context, so the output is "bbbbaa". Now try the following variant with a double-backslash operator.

```
xfst[0]: clear stack
xfst[0]: read regex b -> a \\ _ a ;
xfst[1]: down bbbbba
aaaaaa
```

The output here is "aaaaaa" because the last **a** in the input maps (by default) to itself on the surface, the last **b** is mapped to an **a** on the surface because it has a surface right context of **a**; and then the remaining **b**s also map to **a**, with the rule appearing to generate its own right context. In reality, of course, the rule denotes a relation rather than some kind of algorithm, and the relation simply maps a lexical "bbbbba" to a surface "aaaaaa".

Longest Match Right-arrow, left-to-right, longest-match rules are built on the following template

```
A @-> B || L _ R
```

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is @->, consisting of an at-sign, a minus sign and a right angle-bracket, written without any intervening spaces. Where the expression A can match the input in multiple ways, the @-> rule conceptually matches left-to-right and replaces only the longest match at each step.

Right-arrow, right-to-left, longest-match rules are built on the following template

```
A ->@ B || L _ R
```

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is ->@, consisting of a minus sign, a right angle-bracket and an at-sign, written without any intervening spaces. Where the expression A can match the input in multiple ways, the ->@ rule conceptually matches right-to-left and replaces only the longest match at each step.

Longest-match rules are often useful in noun-phrase identification and other kinds of syntactic "chunking".

Shortest Match Right-arrow, left-to-right, shortest-match rules are built on the following template

```
A @> B || L _ R
```

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is @>, consisting of an at-sign and a right angle-bracket, written without any intervening spaces. Where the expression A can match the input in multiple ways, the @> rule conceptually matches left-to-right and replaces only the shortest match at each step.

Right-arrow, right-to-left, shortest-match rules are built on the following template

```
A >@ B || L _ R
```

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The new arrow operator is >@, consisting of a right angle-bracket and an at-sign, written without any intervening spaces. Where the expression A can match the input in multiple ways, the >@ rule conceptually matches right-to-left and replaces only the shortest match at each step.

Shortest-match rules are not widely used.

Backslash-Slash Rules The right-arrow backslash-slash rules are built on the following template:

```
A -> B \/ L _ R
```

where A, B, L and R have the same restrictions and possibilities as in other rule subtypes. The \/ operator indicates that both contexts L and R must match on the lower (output) side of the relation.

Backslash-slash rules have not been widely used.

Epenthesis Rules Epenthesis rules insert new symbols *ex nihilo* into strings, mapping the empty string into non-empty strings. Such rules are built on the template:

```
[..] -> A || L _ R
```

where A, L and R have the same restrictions and possibilities as in other rule subtypes. A simple epenthesis rule that inserts the symbol **p** between a left context **m** and a right context **k** is shown in Figure 3.20. If the network corresponding to this rule is applied in a downward direction to the string "pumkin", the output is "pumpkin".

```
xfst[0]: clear stack
xfst[0]: read regex [..] -> p || m _ k ;
xfst[1]: apply down pumkin
pumpkin
```

```
[..] -> p || m _ k
```

Figure 3.20: A Simple Epenthesis Rule that Maps "pumkin" to "pumpkin", Inserting a Single **p** between **m** and **k**

```
[] -> p || m _ k
```

Figure 3.21: A Bad Epenthesis Rule that will Try to Insert an Infinite Number of **p**s between **m** and **k**

The rule in Figure 3.20 illustrates the DOTTED BRACKETS **[.** and **.]**, which are required in such epenthesis rules. The need for the dotted brackets is best explained by illustrating what will happen if one tries to write the rule using ordinary square brackets as in Figure 3.21, or with the equivalent **0** notation as in `0 -> p || m _ k`. The notation `[]` or **0** denotes the empty string, and the rule would appear at first glance to be correct, mapping the empty string into **p** between **m** and **k**. However, it must be understood that there are an infinite number of empty strings (which take up no space) between the **m** and the **k** in a word like "pumkin", and therefore the rule in Figure 3.21 will dutifully try to insert an infinite number of **p**s. In practice, this results in an error or strange output.

```
xfst[0]: clear stack
xfst[0]: read regex [] -> p || m _ k ;
xfst[1]: apply down pumkin
Network has an epsilon loop on the input side.
pumkin
pumpkin
```

Here **xfst** warns that the network has an epsilon loop, corresponding to the infinite number of epsilons (empty strings) between **m** and **k**, and the output is not what we expected. In some cases, an attempt to apply such rules will even result in a segmentation fault and cause **xfst** to crash.

```
[.a*.] @-> p || m _ k
```

Figure 3.22: Dotted Brackets Around the Expression `a*`, which can Match the Empty String

The useful effect of the new `[.` and `.]` dotted brackets is to cause the rule to be compiled without an epsilon loop, so that it will treat the input as having only one empty string before and after each input symbol. In addition to the notation `[..]`, indicating a single empty string, dotted brackets are also necessary in examples like the rule in Figure 3.22, where the expression being replaced is `[.a*.]`. It should by now be understood that the regular expression `a*` matches all strings containing zero or more **a**s, and that this includes the empty string. The bad epenthesis rule `a* @-> p || m _ k` will therefore try, in cases where there are no **a**s in the context, to insert an infinite number of **p**s between **m** and **k**. So in any replace rule, where the input expression can match the empty string, the input expression should be enclosed in dotted brackets.

```
[a|e|i|o|u] -> %[ ... %]
```

Figure 3.23: A Bracketing Rule to Surround Vowels in the Output with Literal Square Bracket Symbols

Bracketing or Markup Rules It is often useful to have rules that match certain patterns in input strings and surround them in the output with some kind of bracketing. This bracketing might consist of punctuation symbols like [and], XML tags like `<latin>` and `</latin>`, or whatever is meaningful and useful in the current application. For example, the rule in Figure 3.23 surrounds vowels with

literal square brackets. The three dots . . ., written without intervening spaces, are in fact a special finite-state operator that refers to the matched symbols. The rule indicates that the matched symbols are to be mapped without change to the output and surrounded with square brackets.

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u] -> %[ ... %] ;
xfst[1]: apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]ly
```

Bracketing rules perform a kind of double epenthesis around the matched substring. The vowel-bracketing rule could be rewritten, in a less readable and harder to maintain form, using a pair of parallel epenthesis rules.

```
xfst[0]: clear stack
xfst[0]: read regex [ [..] -> %[ || _ [a|e|i|o|u] ,,
[..] -> %] || [a|e|i|o|u] _ ];
xfst[1]: apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]ly
```

The bracketing expressions can include strings of symbols, for example XML tags.

```
xfst[0]: clear stack
xfst[0]: read regex "ad hoc" -> %<latin%> ... %<%/latin%> ;
xfst[1]: apply down avoid writing ad hoc code
avoid writing <latin>ad hoc</latin> code
```

Bracketing rules can also be conditioned with contexts in the usual ways. The following rule surrounds "ad hoc" with the tags <latin> and </latin> unless it is already surrounded with such tags.

```
xfst[0]: clear stack
xfst[0]: read regex "ad hoc" -> %<latin%> ... %<%/latin%> ||
.#. ~[ ?* %<latin%> " "* ] _  ~[ " "* %<%/latin%> ?* ] .#. ;
xfst[1]: apply down avoid writing ad hoc code
avoid writing <latin>ad hoc</latin> code
xfst[1]: apply down avoid writing <latin>ad hoc</latin> code
avoid writing <latin>ad hoc</latin> code
```

Bracketing rules using the longest-match operator are often useful for bracketing maximally long sequences where the matching expression could match in multiple ways. As a simple example, assume that we wanted to bracket not just single vowels but sequences of vowels. The following rule can match the input in multiple ways and so generates multiple outputs.

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u]+ -> %[ ... %] ;
xfst[1]: apply down feeling
```

```
f[e][e]l[i]ng
f[ee]l[i]ng
xfst[1]: apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[oo]lcl[ea]n[i]ng
```

To favor the longest matches, the @-> operator can be used:

```
xfst[0]: clear stack
xfst[0]: read regex [a|e|i|o|u]+ @-> %[ ... %] ;
xfst[1]: apply down feeling
f[ee]l[i]ng
xfst[1]: apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```

This longest-match behavior is often useful in syntactic "chunking" applications, such as bracketing noun phrases in text. Let's assume that the words in our input sentences have already been morphologically analyzed and reduced to part-of-speech tags such as **d** for determiner, **a** for adjective, **n** for nouns, **p** for prepositions and **v** for verbs. The sentence originally reading "`The quick brown fox jumped over the lazy dogs.`" would then be reduced to "daanvpdan". Similarly, the sentence

```
Boy scouts do good deeds for the senior citizens
living on the poor side of town.
```

would reduce to the string of symbols "nnvanpdanvpdanpn". A very loose characterization of English noun phrases, appropriate for this simple example, is that they start with an optional determiner (d), followed by any number of adjectives a*, and end with one or more nouns n+. The following rule brackets all possible noun phrases:

```
xfst[0]: clear stack
xfst[0]: read regex (d) a* n+ -> %[ ... %] ;
xfst[0]: apply down daanvpdan
[daan]vp[dan]
[daan]vpd[an]
[daan]vpda[n]
d[aan]vp[dan]
d[aan]vpd[an]
d[aan]vpda[n]
da[an]vp[dan]
da[an]vpd[an]
da[an]vpda[n]
daa[n]vp[dan]
daa[n]vpd[an]
```

```
daa[n]vpda[n]
xfst[0]: apply down nnvanpdanvpdanpn
[n] [n]v[an]p[dan]vp[dan]p[n]
[n] [n]v[an]p[dan]vpd[an]p[n]
[n] [n]v[an]p[dan]vpda[n]p[n]
[n] [n]v[an]pd[an]vp[dan]p[n]
[n] [n]v[an]pd[an]vpd[an]p[n]
[n] [n]v[an]pd[an]vpda[n]p[n]
[n] [n]v[an]pda[n]vp[dan]p[n]
[n] [n]v[an]pda[n]vpd[an]p[n]
[n] [n]v[an]pda[n]vpda[n]p[n]
[n] [n]va[n]p[dan]vp[dan]p[n]
[n] [n]va[n]p[dan]vpd[an]p[n]
[n] [n]va[n]p[dan]vpda[n]p[n]
[n] [n]va[n]pd[an]vp[dan]p[n]
[n] [n]va[n]pd[an]vpd[an]p[n]
[n] [n]va[n]pd[an]vpda[n]p[n]
[n] [n]va[n]pda[n]vp[dan]p[n]
[n] [n]va[n]pda[n]vpd[an]p[n]
[n] [n]va[n]pda[n]vpda[n]p[n]
[nn]v[an]p[dan]vp[dan]p[n]
[nn]v[an]p[dan]vpd[an]p[n]
[nn]v[an]p[dan]vpda[n]p[n]
[nn]v[an]pd[an]vp[dan]p[n]
[nn]v[an]pd[an]vpd[an]p[n]
[nn]v[an]pd[an]vpda[n]p[n]
[nn]v[an]pda[n]vp[dan]p[n]
[nn]v[an]pda[n]vpd[an]p[n]
[nn]v[an]pda[n]vpda[n]p[n]
[nn]va[n]p[dan]vp[dan]p[n]
[nn]va[n]p[dan]vpd[an]p[n]
[nn]va[n]p[dan]vpda[n]p[n]
[nn]va[n]pd[an]vp[dan]p[n]
[nn]va[n]pd[an]vpd[an]p[n]
[nn]va[n]pd[an]vpda[n]p[n]
[nn]va[n]pda[n]vp[dan]p[n]
[nn]va[n]pda[n]vpd[an]p[n]
[nn]va[n]pda[n]vpda[n]p[n]
```

The reason for all the outputs is that a string like "`the lazy dogs`" is a noun phrase, but so are "`lazy dogs`" and just "`dogs`". What we want, in practice, is to bracket whole noun phrases, preferring the longest possible match in each case where multiple matches are possible. Again, the `@->` operator saves the day, returning a single output for each input string.

```
xfst[0]: clear stack
xfst[0]: read regex (d) a* n+ @-> %[ ... %] ;
xfst[0]: apply down daanvpdan
[daan]vp[dan]
```

```
xfst[0]: apply down nnvanpdanvpdanpn
[nn]v[an]p[dan]vp[dan]p[n]
```

Left-Arrow Rules

While all replace rules compile into transducers, and so are bi-directional, right-arrow rules are written with a certain notational bias towards generation. The input side of a right-arrow rule is most naturally visualized as the upper side. Right-arrow rules are very frequently used in phonology and morphology to map from abstract lexical strings, on the upper side, to surface or more surfacy strings on the lower side; networks compiled from right-arrow rules are typically composed on the bottom of a lexicon network.

The family of left-arrow rules, in contrast, has a notational bias towards upward or analysis-like application. Left-arrow rules are often composed on the upper side of a lexicon network, mapping from lexical strings upwards to modified lexical strings.

The simplest left-arrow rules are built on the following template

```
B <- A
```

where `A` and `B` are regular expressions denoting languages (not relations). The overall `B <- A` rule compiles into a transducer that is the inversion of `A -> B`.

In morphological analyzers at **Xerox**, left-arrow replace rules are often used to modify morphological tags on the upper side of a lexical transducer. Consider the case where an English-speaking linguist has written a morphological analyzer of Portuguese that contains pairs of strings like the following, where `+Noun`, `+Masc` and `+Pl` are multicharacter symbols.

```
Lexical: livro+Noun+Masc+Pl
Surface: livros
```

A Portuguese-speaking user might prefer to see a tag like `+Subst`, for *substantivo*, in place of the English-oriented `+Noun` tag. Changing such a tag on the upper side is a trivial matter using left-arrow rules. Assuming that the original lexical transducer is stored in a binary file named `lexicon.fst`, the command

```
xfst[0]: clear stack
xfst[0]: read regex [%+Subst <- %+Noun] .o. @"lexicon.fst" ;
xfst[1]: save stack lexicon-port.fst
```

will produce a new version of the lexical transducer with pairs of strings like the following

```
Lexical: livro+Subst+Masc+Pl
Surface: livros
```

As far as anyone can tell from looking at the result `lexicon-port.fst`, the tag `+Subst` was there from the beginning.

Of course, it can easily be shown that left-arrow rules are not formally necessary. One could achieve the same result by

1. Inverting `lexicon.fst`,
2. Composing the right-arrow rule `[ %+Noun -> %+Subst ]` on the bottom of the inverted lexicon, and
3. Inverting the result.

as in the following commands.

```
xfst[0]: clear stack
xfst[0]: read regex
[[@"lexicon.fst"].i .o. [ %+Noun -> %+Subst ] ].i ;
xfst[1]: save stack lexicon-port.fst
```

However, such multiple inversions, and the writing of upside-down rules, are not at all natural to most developers.

Left-arrow rules can also have contexts

```
B <- A || L _ R
```

and the || operator here indicates that both contexts must match on the default input side, which in a left-arrow rule is the *lower* side.

The // and \\ operators are also available in left-arrow rules, but their interpretation requires some comment. In left-arrow double-slash rules such as

```
B <- A // L _ R
```

the // operator indicates that the left context must be matched on the *upper* side, i.e. the output side, and that the right context must be matched on the *lower* or input side. Conversely, in a rule such as

```
B <- A \\ L _ R
```

the left context is matched on the lower or input side, and the right context is matched on the upper or output side.

The left-arrow backslash-slash rules are built on the following template.

```
A <- B \/ L _ R
```

In a left-arrow rule, the \/ operator indicates that both contexts `L` and `R` must match on the upper (output) side of the relation.

Optional left-arrow mapping is also possible using the `(<-)` operator, which is just the left arrow with parentheses around it. A rule like `[ b (<-) a ]` maps each lower-side **a** both to **b** and to **a**, resulting in behavior like the following:

```
xfst[0]: clear stack
xfst[0]: read regex b (<-) a ;
xfst[1]: apply up abba
bbbb
bbba
abbb
abba
```

The left-arrow operators `<-@`, `<@`, `@<-` and `@<` are also available.

Double-Arrow Rules

Finally, **xfst** provides double-arrow rules built on the following templates:

```
A <-> B
A <-> B || L _ R
```

where `A`, `B`, `L` and `R` must denote languages, not relations. Such a rule is compiled like the simultaneous pair of rules

```
[ A -> B || L _ R ,, A <- B || L _ R ]
```

where in the first rule, `A -> B || L _ R`, the contexts must match on the upper side of the relation, and in the second rule, `A <- B || L _ R`, they must match on the lower side. Double-arrow rules are also possible with the `//` and `\\` operators, but in these cases the understanding of where the contexts must match transcends normal human comprehension. Double-arrow rules have not been widely used in practice.

3.6 Examining Networks

Once a network has been compiled and pushed onto The Stack, one can always test it using the **apply up** and **apply down** commands. However, there exist other helpful commands that reveal the nature and contents of your networks.

3.6.1 Boolean Testing of Networks

xfst does not include control structures such as **if-then**, **while** or **for** that respond to boolean tests, but certain tests are available during manual interaction.

N-ary Boolean Tests

- The **test equivalent** command returns 1 (TRUE) if and only if all the networks on the stack are equivalent.
- The **test overlap** command returns 1 (TRUE) if and only if all the networks on the stack have a non-empty intersection.

- The **test sublanguage** returns 1 (TRUE) if and only if the language of the i-th network is a sublanguage of the next (i+1-th) network on The Stack. The comparison starts at the topmost network (i=0) and works down.

Unary Boolean Testing

The unary boolean tests are always applied to the top network on The Stack.

- The **test null** command returns 1 (TRUE) if and only if the top network on The Stack encodes the null language.
- The **test non-null** command returns 1 (TRUE) if and only if the top network on The Stack encodes a non-null language or relation.
- The **test upper-bounded** command returns 1 (TRUE) if and only if the upper side of the network on the top of The Stack has no epsilon cycles.
- The **test upper-universal** command returns 1 (TRUE) if and only if the upper side of the network on the top of The Stack contains the universal language.
- The **test lower-bounded** command returns 1 (TRUE) if and only if the lower side of the network on the top of The Stack has no epsilon cycles.
- The **test lower-universal** command returns 1 (TRUE) if and only if the lower side of the network on the top of The Stack contains the universal language.

3.6.2 Printing Words and Paths

Print Commands We have already introduced the **print words** utility, which might better be termed *print paths*. When the network on the top of The Stack encodes a language, as in Figure 3.24, the output is just a list of the words in the language.

When the network on the top of The Stack encodes a relation, as in Figure 3.25, **print words** outputs each path, using the notation <**u:l**> when an arc label has **u** on the upper side and **l** on the lower side.

For any transducer, i.e. any network encoding a relation, use **print upper-words** to see an enumeration of the upper language; similarly, use **print lower-words** to see an enumeration of the lower language.

As networks grow larger, precluding practical enumeration to the terminal of the entire language or relation, you can resort to using **print random-words**, which, as its name suggests, prints a random selection of the paths. **xfst** also provides **print random-upper** and **print random-lower** to display random strings from the upper or lower language, respectively.

Finally, it is sometimes useful to identify the longest string in a network, or its length, and the commands **print longest-string** and **print longest-string-size** are provided.

```
xfst[n]: clear stack
xfst[0]: read regex [ {dog} | {cat} | {horse} ] [s|0] ;
xfst[1]: print words
dog
dogs
cat
cats
horse
horses
```

Figure 3.24: **print words** Enumerates a Language

```
xfst[n]: clear stack
xfst[0]: read regex
[ {dog} | {cat} | {horse} ] %+Noun:0 [ %+Pl:s | %+Sg:0 ]
| {ox} %+Noun:0 [ %+Sg:0 | %+Pl:{en} ]
| [ {sheep} | {deer} ] %+Noun:0 [ %+Sg:0 | %+Pl:0 ]
;
xfst[1]: print words
cat<+Noun:0><+Sg:0>
cat<+Noun:0><+Pl:s>
horse<+Noun:0><+Sg:0>
horse<+Noun:0><+Pl:s>
ox<+Noun:0><+Sg:0>
ox<+Noun:0><+Pl:e><0:n>
sheep<+Noun:0><+Pl:0>
sheep<+Noun:0><+Sg:0>
dog<+Noun:0><+Sg:0>
dog<+Noun:0><+Pl:s>
deer<+Noun:0><+Pl:0>
deer<+Noun:0><+Sg:0>
```

Figure 3.25: **print words** Enumerates a Relation

Variables Affecting Print Commands By default, the **print** commands return valid "words", i.e. strings to which the network could be successfully applied. These words do not show which sequences of letters are treated as multicharacter symbols.

xfst provides the interface variable **print-space**, set to **OFF** by default, that can be set **ON** to force the various **print** commands to print spaces between symbols. The effect can be seen in the following example:

print words print upper-words print lower-words
print random-words print random-upper print random-lower
print longest-string print longest-string-size
print sigma print labels print sigma-tally print label-tally

Figure 3.26: Some Useful Print Commands

```
xfst[0]: read regex [ {dog} | {cat} ]
%+Noun:0
[ %+Pl:s | %+Sg:0 ] ;
364 bytes. 8 states, 9 arcs, 4 paths.
xfst[1]: up dog
dog+Noun+Sg
xfst[1]: up cats
cat+Noun+Pl
xfst[1]: set print-space ON
variable print-space = ON
xfst[1]: up dog
d o g +Noun +Sg
xfst[1]: up cat
c a t +Noun +Sg
xfst[1]: up cats
c a t +Noun +Pl
```

Note that the output, when **print-space=ON**, shows clearly that +Noun, +Sg and +Pl are being treated as multicharacter symbols in the network.

The effect of a similar variable, **show-flags**, will be demonstrated in Chapter 7.

3.6.3 Alphabets of a Network

Printing Alphabets

Each network has two alphabets that are usually distinct: the LABEL ALPHABET and the SIGMA ALPHABET. The label alphabet is the collection of labels that actually appear on arcs in the network. Labels of the form **u:l** overtly signal that the

network is a transducer, and single-character labels such as **a** are effectively treated like **a:a** in the **Xerox** encoding of networks.

The sigma alphabet is the set of individual symbols known to the network; the sigma alphabet never contains labels of the form **u:l** but only the individual symbols **u** and **l**. The example in Figure 3.27 illustrates the use of the **print labels** command to display the label alphabet and the **print sigma** command to display the sigma alphabet. As usual, these commands refer by default to the network on the top of The Stack. If `Myvar` is a defined variable set to a network value, then **print labels Myvar** and **print sigma Myvar** will print the alphabets for that network. The rarely used **print label-tally** and **print sigma-tally** quantify the frequencies of labels and symbols, respectively.

```
xfst[0]: clear stack
xfst[0]: read regex
[ {dog} | {cat} | {horse} ] %+Noun:0 [ %+Pl:s | %+Sg:0 ]
| {ox} %+Noun:0 [ %+Sg:0 | %+Pl:{en} ]
| [ {sheep} | {deer} ] %+Noun:0 [ %+Sg:0 | %+Pl:0 ]
;
xfst[1]: print labels
a c d e g h o p r s t x <0:n> <+Noun:0> <+Pl:e>
<+Pl:s> <+Pl:0> <+Sg:0>
Size: 18
xfst[1]: print sigma
a c d e g h n o p r s t x +Noun +Pl +Sg
Size: 16
```

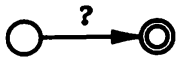
Figure 3.27: Displaying the Label and Sigma Alphabet of a Network

In some cases (see Section 2.3.4), symbols in the sigma alphabet may not actually appear on any arc and so will not appear in the label alphabet. The simplest and most obvious examples involve networks compiled from a regular expression that contains the symbol-complement operator \ as in Figure 3.28. The network for `[\a]` is shown in Figure 3.29. Note that the symbol **a** occurs in the sigma alphabet but not in the label alphabet. The question mark in the sigma alphabet represents UNKNOWN (or "OTHER") symbols, and the presence of **a** in the sigma alphabet indicates that **a** is known (and therefore not included in the UNKNOWN symbols).

The question mark (denoting UNKNOWN symbols) must always be interpreted relative to the symbols that are known, i.e. to the concrete symbols that are in the sigma alphabet. For this reason, networks are always associated with a sigma alphabet. In finite-state operations like union and composition that join two networks, the resolution of the respective sigma alphabets is a non-trivial problem that is taken care of by the underlying algorithms.

```
xfst[0]: clear stack
xfst[0]: read regex \a ;
xfst[1]: print labels
?
xfst[1]: print sigma
? a
```

Figure 3.28: Label Alphabet vs. Sigma Alphabet



Sigma: {?, a}

Figure 3.29: The Network Encoding the Language [\a]

Symbol Tokenization of Input Strings

When a network is applied to an input string using **apply up** or **apply down**, the input string is first tokenized into individual symbols (see Section 2.3.6). This tokenization includes the identification of multicharacter symbols, and it must be performed relative to the sigma alphabet of the network being applied. In the following example, the network's sigma includes the multicharacter symbols +Noun, +Pl and +Sg; when the network is applied in a downward direction to the string "dog+Noun+Pl", the sigma alphabet is consulted and the string is symbol-tokenized into

```
d o g +Noun +Pl
```

before the symbols are matched against upper-side labels in the network.

```
xfst[0]: read regex [ {dog} | {cat} ]
%+Noun:0
[ %+Pl:s | %+Sg:0 ] ;
364 bytes. 8 states, 9 arcs, 4 paths.
xfst[1]: sigma
a c d g o s t +Noun +Pl +Sg
Size: 10
xfst[1]: apply down dog+Noun+Pl
dogs
```

Where an input string might be tokenized in multiple ways, the apply routines resolve the ambiguity by processing the input string left-to-right, always selecting at each point the longest possible match. If the sigma includes **+**, **P**, **l**, and **+Pl**,

the multicharacter symbol will always be chosen over the single symbols. If the sigma includes multiple multicharacter symbols whose spellings start the same way, e.g. **+Pl**, **+Plur** and **+Plural**, the longest multicharacter symbol will always have precedence.

> The deterministic tokenization of input strings is one of the several reasons to avoid creating multicharacter symbols like **ing**, which are visually indistinguishable from concatenations of single alphabetic symbols. If it looks like a network should match an input string, but doesn't, then you should suspect a problem with multicharacter symbols and tokenization. Use **print sigma** to see what the network's symbols really are.

3.6.4 Printing Information about Networks

When dealing with non-trivial networks, size often becomes an issue. The command **print size** displays the size of a network in terms of bytes, states, arcs and paths, the same information that is displayed when a network is compiled using **read regex**. If the network contains a loop, and so contains an infinite number of paths, **print size** indicates that the network is "Circular".

```
xfst[0]: clear stack
xfst[0]: read regex {dog} | {cat} ;
6 states, 6 arcs, 2 paths.
xfst[1]: print size
6 states, 6 arcs, 2 paths.
xfst[1]: clear stack
xfst[0]: read regex a b* (c) d+ [e|f] ;
5 states, 8 arcs, Circular.
xfst[1]: print size
5 states, 8 arcs, Circular.
```

The output of **print size** can also be directed to a file using **print size** > ***filename***.

By default, **print size** displays information about the top network on The Stack. If **MyNet** is a variable defined to have a network value, then the command `print size MyNet` will display the size of that network.

```
xfst[0]: define MyNet [ {dog} | {cat} ] [ s | 0 ] ;
7 states, 7 arcs, 4 paths.
xfst[0]: print size MyNet
7 states, 7 arcs, 4 paths.
```

The **print stack** command, previously introduced in Section 3.2.4, displays size information about all the networks on The Stack. The output of **print stack**

can also be directed to a file using **print stack** > *filename*. Similarly, use **print defined** and **print defined** > *filename* to display information about the set of networks stored in defined variables.

The **print net** command, with the variants **print net** *defined-variable* and **print net** > *filename* displays detailed information about a network, including the sigma alphabet, the size (in paths), and other features including the ARITY, where an arity of 1 indicates a simple automaton encoding a language and an arity of 2 indicates a transducer. Finally, **print net** lists each state in the network, followed by notations describing the arcs leading from that state.

```
xfst[0]: clear stack
xfst[0]: read regex [ {dog} | {cat} ] [ s | o ] ;
7 states, 7 arcs, 4 paths.
xfst[0]: print net
Sigma: a c d g o s t
Size: 7
Net: EE2D8
Flags: deterministic, pruned, minimized, epsilon_free,
loop_free
Arity: 1
s0:   c -> s1, d -> s2.
s1:   a -> s3.
s2:   o -> s4.
s3:   t -> fs5.
s4:   g -> fs5.
fs5:  s -> fs6.
fs6:  (no arcs)
```

In the listing of states and arcs, notations such as **s0**, **s1** and **s2** indicate non-final states. The state numbered 0 is the start state, and the other numbering is more or less arbitrary though used consistently in the output. The line

```
s0:  c -> s1, d -> s2.
```

indicates that state 0 has two arcs leading from it: one is labeled **c** and leads to state 1; and the other is labeled **d** and leads to state 2. The notations **fs5** and **fs6** indicate final states.

3.6.5 Inspection of Networks

The **print net** command, which identifies all the states and arcs of a network, is useful for visualizing fairly small nets. However, a full-sized net for natural language processing frequently contains hundreds of thousands of states and arcs, and the output from **print net** would be overwhelming. To explore a larger network, traversing it state by state, following the arcs, use the **inspect net** command. As **inspect net** is primarily intended for expert users, and particularly for **Xerox** developers debugging new finite-state algorithms, we will not go into the details here.

If you want to experiment with network inspection, see **help inspect net** and the command summaries displayed when you invoke **inspect net**.

3.7 Miscellaneous Operations on Networks

3.7.1 Substitution

Substitution Commands

The various **substitute** commands are a powerful method for modifying networks. To better understand the following commands, recall that each arc in a network has a LABEL, which may be a single SYMBOL like **V**, visible on both sides, or a compound label like **U:L**, where **U** is a symbol visible on the upper side and **L** is a symbol visible on the lower side. Thus **U:L** is a label; and **V**, if it appears alone on an arc, is also a label. **V**, **U** and **L** are symbols; but **U:L** is not a symbol.

- The **substitute label** command is built on the following template

  `substitute label` *`list-of-labels`* `for` *`label`*

  It replaces the top network on The Stack with a network derived by replacing every arc with the given label by an arc or arcs, each labeled with one of the substitute labels. If the list consists of the keyword NOTHING, all paths containing the given label are removed. The label and the substitutes may be single symbols or symbol pairs.

  The following example would replace all instances of the label **V** with the labels **a**, **e**, **i**, **o** and **u**.

  ```
  xfst[1]: substitute label a e i o u for V
  ```

  Note that this command, as written, will not affect any labels such as **V:x** or **y:V**, where **V** is not the whole label but only a symbol on one side of an upper:lower label.

- The **substitute symbol** command is built on the following template

  `substitute symbol` *`list-of-symbols`* `for` *`symbol`*

  It replaces the top network on The Stack with a network derived by replacing every arc whose label contains the given symbol by an arc or arcs whose label instead contains one of the substitute symbols. If the list consists of the keyword NOTHING, all paths containing the given symbol are removed. The symbol and the substitutes must be single symbols, not symbol pairs.

  The following example would replace all instances of the symbol **V** with the symbols **a**, **e**, **i**, **o** and **u**.

```
xfst[1]: substitute symbol a e i o u for V
```

Note that this command, in contrast to the **substitute label** command, will affect labels such as **V:x** and **y:V** where **V** is a symbol in a label. It will also affect the label **V**, which, in the **Xerox** implementation of networks, encodes the relation **V:V**.

The **substitute symbol** command can also be indicated in regular expressions as

```
'[ [A], S, L ]
```

where `A` is the network to be affected, `S` is the symbol to be replaced, and `L` is the list of replacement symbols, e.g.

```
'[ [ @"MyNetwork.fst" ], V, a e i o u ]
```

The regular-expression syntax is not much used.

- The **substitute defined** command is built on the following template

```
substitute defined defined-variable for label
```

It replaces the top network on the stack with a network derived by splicing in the network associated with the given defined variable for each arc that has the indicated label. If the network is a transducer, the label must not be a symbol that occurs in a (non-identity) symbol pair.

The **substitute** commands provide powerful, and probably under-appreciated, ways to modify networks. The **substitute defined** command, in particular, allows networks to be built with single placeholder labels that can later be substituted with whole networks (see Section 9.5.4). Different versions of a common system could be produced by building a master network with placeholder labels, and then substituting in different networks for the placeholder labels.

Substitution Applications

Shuffling The **shuffle net** function, which is invoked by the binary operator <> in regular expressions, was developed some years ago to shuffle the individual letters of the strings of one language with the individual letters of the strings of a second language. For example,

```
xfst[0]: read regex [ a b <> c d ] ;
9 states, 12 arcs, 6 paths.
xfst[1]: print words
acdb
acbd
abcd
cadb
cabd
cdab
```

However, shuffling found little practical use until we looked at some American Indian languages and saw suffix classes that could appear in free relative order. When **shuffle** is used with individual symbols, the result is a language of strings that contains all permutations of those symbols, i.e. they appear in free relative order.

```
xfst[0]: read regex [ Suff1 <> Suff2 ] <> Suff3 ;
8 states, 12 arcs, 6 paths.
xfst[1]: words
Suff1Suff3Suff2
Suff1Suff2Suff3
Suff2Suff3Suff1
Suff2Suff1Suff3
Suff3Suff1Suff2
Suff3Suff2Suff1
```

Once such permutations are encoded in a network with individual multicharacter labels like **Suff1**, **Suff2** and **Suff3**, and if the network is left on the top of The Stack, one can then use **substitute defined** to replace them with networks of arbitrary complexity representing the corresponding suffix classes.

```
xfst[1]: define S1 [ k o n | d i i | f o t ] ;
xfst[1]: substitute defined S1 for Suff1
xfst[1]: define S2 [ l l a t | b o o | t e e ] ;
xfst[1]: substitute defined S2 for Suff2
xfst[1]: define S3 [ x o n | l m a n | k o z ] ;
xfst[1]: substitute defined S3 for Suff3
```

In this example, the resulting network contains 162 paths, representing the possible permutations of all the individual suffixes in the three classes.

Adding Roots to a Network The following script defines a network that accepts a language of words built on two roots, *raam* and *cazard*.

```
! lexicon script

clear stack
define Roots {raam} | {cazard} ;
```

```
define Suff1 {ze} | {azuu} ;
define Suff2 {lazam} | {mule} | {kiba} ;
read regex Roots Suff1 Suff2 ;
save stack Lex.fst
```

To add the two new roots *zalam* and *gexar*, you usually have to edit the script and change the `define Roots` line, e.g.

```
define Roots {raam} | {cazard} | {zalam} | {gexar} ;
```

and then recompile everything from scratch.

In some simple cases, like this one, it's possible to add new roots to a compiled lexicon without recompilation. The trick is to introduce a placeholder multicharacter symbol such as **%*ROOTPLACEHOLDER%***, as in the following script.

```
! lexicon script

clear stack
define Roots {raam} | {cazard} | %*ROOTPLACEHOLDER%* ;
define Suff1 {ze} | {azuu} ;
define Suff2 {lazam} | {mule} | {kiba} ;
read regex Roots Suff1 Suff2 ;
save stack Lex.fst
```

This will add meaningless strings like "`*ROOTPLACEHOLDER*zelazam`" to the language, but if the placeholder symbol is given a bizarre name like this one, preferably with some literal punctuation symbols thrown in, the bizarre strings won't match any normal input. Now we can add new roots to the lexicon via substitution, e.g.

```
xfst[0]: define New {zalam} | {gexar} |
%*ROOTPLACEHOLDER%* ;
xfst[0]: load stack Lex.fst
xfst[1]: substitute defined New for %*ROOTPLACEHOLDER%*
```

The resulting network now contains four roots plus a new copy of the placeholder, in case some more new roots need to be added later. Test these examples, using **print words** to see the language before and after the **substitute** command.

It should be noted that this substitution trick, avoiding a recompilation, works only if the roots are unaffected by subsequently composed alternation rules or other similar processing; the general ability to add new roots to a complex lexical transducer is a non-trivial problem. Nevertheless, the substitution trick shown here has proved useful in certain applications.

3.7.2 Network Properties

It is often useful to attach documentation to a network, including a version number, the date of creation, the names of the authors, etc. This is especially important for commercial products and any other networks that are delivered for use in other projects. **xfst** therefore allows an arbitrary number of feature:value pairs called PROPERTIES to be associated with a network, where the feature and value are strings chosen by the developer. Properties do not affect the functioning of a network in any way.

The **read properties** < ***filename*** command reads a set of feature:value pairs from a text file and stores them in the top network on The Stack. If you don't specify a filename, **xfst** will expect you to enter feature:value pairs at the terminal. The input should be of the format

```
FEATURE1: VALUE1
FEATURE2: VALUE2
FEATURE3: VALUE3
etc.
```

where FEATUREn and VALUEn are strings, and they must be enclosed by double quotes if they contain spaces. Each feature:value pair must reside on a single line. Such files are typically created with a text editor like **xemacs** or **vi**.

The following is a slightly edited version of a real property file used to mark a network that performed morphological analysis for Italian.

```
LANGUAGE: ITALIAN
CHARENCODING: ISO8859/1
VERSIONSTRING: "1.2"
COPYRIGHT1: "Copyright (c) 1994 1995 Xerox Corporation."
COPYRIGHT2: "All rights reserved."
TIMESTAMP: "29 November 1995; 16:30 GMT"
AUTHOR1: "Antonio Romano"
AUTHOR2: "Cristina Barbero"
AUTHOR3: "Dario Sestero"
AUTHOR4: "Kenneth Beesley"
```

The **read properties** command overwrites any previously set properties.

The **add properties** command is like **read properties**, reading feature:value pairs from a file or the terminal, but the new pairs are added to any existing pairs rather than overwriting them.

The **edit properties** command initiates a dialog that allows the developer to edit the property list manually. The dialogs are a bit awkward, so most developers prefer to edit feature:value pairs in a text file and call **read properties** or **add properties**.

Finally, **write properties** > ***filename*** writes the properties of the top network on The Stack out to the designated file in the format required by **read properties**. If you do not specify a filename, the output is displayed on the terminal. Use

write properties ***defined-variable*** to write the properties of a network stored in a defined variable.

3.7.3 Housekeeping Operations

The housekeeping operations listed in Table 3.28 are rarely used by developers. For example, after a call to an algorithm like **union net**, the result is automatically pruned, epsilon-removed, determinized and minimized before being pushed onto The Stack. See the **help** messages for more information.

cleanup net
complete net
determinize net
epsilon-remove net
minimize net
prune net
sort net
compact sigma

Table 3.28: Housekeeping Operations

3.7.4 Generation of Derived Networks

The operations listed in Table 3.29 produce various kinds of derived networks from the top network on The Stack. See the **help** messages for more information.

label net
name net
sigma net
substring net

Table 3.29: Commands to Build Derived Networks

3.8 Advanced xfst

3.8.1 Modifying the Behavior of xfst

Quite separate from the defined variables that hold network values, there is a fixed predefined set of internal interface variables that control the behavior of **xfst**. You can see a list of the internal variables, and the commands that query and reset them, by entering **apropos variable**.

```
xfst[0]: apropos variable
set       : sets a new value to the variable
show      : show the value of the variable
assert            : OFF
          : quit if a test fails and quit-on-fail is ON
directory: (prints the name of the current directory)
flag-is-epsilon   : OFF
          : treat flag diacritics as epsilons in composition
minimal           : ON
          : minimize the result of calculus operations
name-nets         : OFF
          : use regular expressions as network names
obey-flags        : ON
          : enforce flag diacritics
print-pairs       : OFF
          : show both sides of labels in apply
print-sigma       : ON
          : show the sigma when a network is printed
print-space       : OFF
          : insert a space between symbols in printing words
quit-on-fail      : ON
          : quit scripts abruptly on any error
quote-special     : OFF
          : print special characters in double quotes
random-seed
          : seed of the random number generator.
recursive-define : OFF
          : allow self-reference in definitions
retokenize        : ON
          : retokenize regular expressions in 'compile-replace'
show-flags        : OFF
          : show flag diacritics when printing
sort-arcs         : ON
          : sort the arcs before printing a network
verbose           : ON
          : print messages
```

We have already seen **print-space** in Section 3.6.2. We will eventually learn about some of the others, including **obey-flags**, **show-flags**, **retokenize** and **flag-is-epsilon**, in coming chapters.

To display short documentation about a particular internal variable, use **help** as usual, e.g.

```
xfst[0]: help obey-flags
variable obey-flags == ON|OFF

when ON, the constraints expressed as flag diacritics
are taken into account by 'print [upper|lower]-words'
and 'print random-[upper|lower]' commands. When OFF,
flag diacritics are treated as ordinary symbols in
listing the contents of a network. Default is ON.
Current value is ON.
```

To show the current value of a variable, use **show** ***variable-name***, e.g.

```
xfst[0]: show obey-flags
variable obey-flags = ON
```

To reset the value of an internal variable, use **set** ***variable-name new-value***, e.g.

```
xfst[0]: set obey-flags OFF
variable obey-flags = OFF
```

As shown, **xfst** will reset and reflect the new value.

3.8.2 Command Abbreviations

You will have noted that **xfst** commands consist of two or even three separate words. However, almost all the commands can be entered as a single word following the examples shown in Table 3.30. Notice that the abbreviated command consists of the content word of the full command, dropping the qualifying words such as **apply**, **print**, and **net**.

3.8.3 Command Aliases

xfst includes a simple kind of macro capability called ALIASES. An alias definition consists of the keyword **alias** followed by a user-chosen alias name, a newline, a series of **xfst** commands, and then `END;`, including the semicolon, to terminate. After entering the alias name and newline, a special prompt is displayed to remind you that you are defining an alias. The alias name should consist of plain alphabetic letters.

```
xfst[0]: alias myaliasname
alias> load lexicon.fst
alias> print sigma
alias> END;
xfst[0];
```

The alias name can then be entered like any other **xfst** command, e.g.

```
xfst[0]: myaliasname
```

Full Command	Abbreviation
apply up	up
apply down	down
clear stack	clear
load stack	load
save stack	save
rotate stack	rotate
print words	words
print upper-words	upper-words
print lower-words	lower-words
print random-words	random-words
print random-upper	random-upper
print random-lower	random-lower
print labels	labels
print sigma	sigma
test equivalent	equivalent
compose net	compose
concatenate net	concatenate
intersect net	intersect
invert net	invert
negate net	negate
reverse net	reverse
union net	union

Table 3.30: Some Common Command Abbreviations

An alias is thus like a script file that can be invoked without the **source** command.

An alternative definition format puts the commands on the same line as the **alias** command, and the alias is then terminated by the newline. In this single-line format, multiple commands must be separated by semicolons.

```
xfst[0]: alias myaliasname load lexicon.fst; print sigma
xfst[0]:
```

To see which aliases have already been defined, type **print aliases**. The defined aliases can also be output to file using **print aliases** > *filename*.

3.8.4 xfst Command-Line Options

If you invoke `xfst -h` from the operating-system command line, you will see the following cryptic usage message about command-line options.

```
unix> xfst -h
usage: xfst [-e "command"] [-f scriptfile] [-flush] [-h]
[-help] [-l startscript] [-pipe] [-s binaryfile]  [-stop]
[-q] [-v]
```

To print out the version number, use `-v`.

```
unix> xfst -v
```

To invoke **xfst** and run a startup script, use the `-l` flag. **xfst** will perform the script and then go into normal interactive mode.

```
unix> xfst -l startup_script
xfst[0]:
```

Multiple startup scripts can be indicated using multiple `-l` flags.

```
unix> xfst -l startup_script1 -l startup_script2
xfst[0]:
```

To run an **xfst** script file and then exit immediately back to the operating system, use `-f`.

```
unix> xfst -f script
```

This is equivalent to

```
unix> xfst
xfst[0]: source script
xfst[0]: exit
```

To indicate an **xfst** command on the operating-system command-line itself, use the `-e` flag. The commands themselves must be enclosed in double quotes if they contain spaces or other symbols that could confuse the operating system. Multiple commands can be specified with multiple `-e` flags. Use `-stop` if you want **xfst** to exit back to the operating system immediately after performing the indicated commands.

```
unix> xfst -e "regex d o g | c a t | m o u s e ;" \
-e "print words" -e "apply up mouse" -stop
10 states, 11 arcs, 3 paths.
dog
cat
mouse
mouse
unix>
```

The backslash in this and following examples indicates to the Unix-like operating system that the command continues on the next line. The command could also be typed on a single line, in which case no backslash should be used.

Use `-q` to indicate "quiet mode", suppressing interactive messages. This is particularly appropriate when running **xfst** as a background process. The `-stop` flag tells **xfst** to terminate the background process, e.g.

```
unix> xfst -q -e "load myfile.fst" -e "apply up dog" \
-stop > outputfile
```

Here's another example that invokes **xfst**, loads a network, analyzes a whole list of words from a tokenized file called `myInput`, and then stops.

```
unix> xfst -q -e "load myfile.fst" \
-e "apply up < myInput" -stop > outputfile
```

More exotic flags include `-pipe` which causes **xfst** to take commands as usual, but without printing command prompts. The flag `-flush` flushes the output buffer after each print command without waiting for an end-of-line character.

Invoking `xfst -help` prints the following summary of command-line options:

```
XFST COMMAND-LINE OPTIONS:
--------------------------
-e "command"    execute the command before starting the
                application.
                If more than one '-e' option is specified,
                the commands are executed in the specified
                order. For example:
                'xfst -e "load foo" -e "reverse net" -e "save fie"'
                After the commands have been executed,
                xfst starts in the interactive mode unless the
```

```
                option '-stop' follows.
-f scriptfile   execute the commands in 'scriptfile' and exit.
-flush          show output immediately without waiting for a
                carriage return.
-h              print a cryptic 'usage' message and exit.
-help           print this help message and exit.
-l startscript  execute the commands in 'startscript' and start
                xfst in the interactive mode.
-pipe           execute without prompting for commands.
-q              operate quietly. Don't print unnecessary
                messages.
-s binaryfile   load 'binaryfile' and start xfst in the
                interactive mode.
                The file must have been created with the 'save'
                command.
-stop           stop xfst without entering into the interactive
                mode.
                Any option following '-stop' is ignored.
-v              print xfst version number and exit.
```

3.8.5 Error Recovery

When **xfst** is executing a script, it may occasionally run into an error. Maybe a command is incorrectly spelled or cannot be executed because of a missing file or some other reason. It may be the case that the error is not important and the script can produce the intended result in spite of the error. However, most scripts are "fragile"; a failure at any point in the script makes the subsequent commands useless or even harmful. When an error occurs in a script, **xfst** will either stop the script or continue executing it depending on the setting of the interface variable **quit-on-fail**. If **quit-on-fail** is **ON** (the default value), the execution stops on the faulty line. For example, if the script contains the line `load lexicon.fst` and the file does not exist, **xfst** aborts the script with an error message such as:

```
Cannot open 'lexicon.fst'
*** Error on line 39 in file 'lexicon.script'.
Aborting the script because quit-on-fail is ON.
```

If the execution of the faulty script was started in non-interactive mode, for example, with the command `xfst -f lexicon.script`, **xfst** itself will also quit on error. If the script was launched in an interactive mode from **xfst** with the **source** command, **xfst** returns control to the user and waits for further commands. If the script was started from Unix with the command `xfst -l lexicon.script`, **xfst** will start interactive processing in the usual way even if an error was encountered and the script ended prematurely.

If the **quit-on-fail** is set to **OFF**, **xfst** runs all the commands in a script even if some of them fail. The default setting is **ON**.

:	High-precedence crossproduct operator
~\ $ $? $.	Complement, symbol complement, containment
+ * ^ .1 .2 .u .l .i .r	Kleene plus and star, iteration, upper-lower, invert and reverse
/	Ignore
	Concatenation (no overt operator)
> <	Precede and follow
| & −	Union, intersect, minus
=> -> (->) @-> etc.	Rule operators
.x. .o.	Crossproduct and compose

Table 3.31: Precedence of Operators from High to Low

3.9 Operator Precedence

In regular expressions as in arithmetic expressions, operators have to be assigned a precedence. For example, in the unbracketed arithmetic expression $3 + 4 * 2$, multiplication conventionally has precedence over addition, resulting in a value of 11. It is obvious that the precedence is important because if the addition were perversely performed first, the answer would be 14. If in fact the intent of the author is to have the addition performed first, then that interpretation would need to be forced by parenthesizing the expression as $(3 + 4) * 2$.

The **xfst** regular-expression operators have the precedence shown in Table 3.31, which descends from high precedence at the top to low precedence at the bottom. In regular expressions, to force particular subexpressions to be performed before others, they can be grouped using square brackets `[]`. Remember that parentheses in **xfst** regular expressions denote optionality. Note also that there are two crossproduct operators in **xfst** regular expressions, the colon (:) which has high precedence, and the `.x.`, which has low precedence.

3.10 Conclusion

xfst is a large and powerful tool for finite-state development. In addition to stack-based operations, it includes a compiler for regular expressions in the **Xerox** format, which includes powerful abbreviated notations such as replace rules. You should now be comfortable with regular expressions and the basic commands in **xfst**; and you should know how to use **help** and **apropos** to explore the rich and somewhat overwhelming set of options. Mastery of **xfst** will come only after considerable practice.

In practical development, **xfst** is usually used together with **lexc** (Chapter 4) to build complete systems.