The XFST interface – quick manual

1 What is XFST?

- XFST is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- XFST includes a regular expression compiler
- The interface of XFST includes a lookup operation (apply up) and a generation operation (apply down)
- The regular expression language employed by XFST is an extended version of standard regular expressions

2 Getting started

2.1 Invoking XFST

When invoking XFST, it will respond with a welcome banner and an xfst prompt, and it then waits for you to type a command after the prompt.

```
Copyright ©Xerox Corporation 1997-2000

Xerox Finite-State Tool, version 7.4.0

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

xfst[0]:
```

2.2 Help

If you enter **help** or a question mark (?) at the xfst prompt, xfst will display a long menu of available commands, organized into subclasses. For short documentation on a particular command, enter help followed by the name of that command. XFST will respond with a useful summary of the usage and semantics of the command.

If you dont 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.

2.3 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.

2.4 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.

3 Compiling Regular Expressions

3.1 XFST Regular Expressions notation

XFST includes a regular expression compiler. For the regular-expression notation of XFST look at the lectures slides.

3.2 Basic Regular-Expression Compilation and Testing

There are two xfst commands, **define** and **read regex**, that invoke the regular expression compiler.

3.2.1 define

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 (carriage return). 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 can be used in subsequent regular expressions.

3.2.2 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 howmany 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.
An important remark:
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 (carriage return). Other commands not containing regular expressions cannot extend over multiple lines and are not terminated with a semicolon.

3.2.3 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]:
```

Another important remark:
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.

3.3 The stack

XFST uses a built-in stack 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.

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, 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.

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 users 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, will have the last-pushed network, the one corresponding to

```
[ {dog} | {cat} | {elephant} ]
```

on the top, with the network for `c a t` underneath it, and the network for `a` on the very bottom. Continuing the same session, if we invoke print words it will refer by default to the top network on The Stack. The other networks below the top are temporarily inaccessible.

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

print stack prints 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 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.

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. For example:

```
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 newtop 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]:
```

4 Working with files

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 both output and input of data and commands.

4.1 Regular Expression Files

We are already acquainted with the read regex command and how it allows us to enter a regular expression manually and have it compiled into a network. 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, and to compile it using the read regex command, adding the `<` symbol to indicate that the input is to be taken from a specified 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. Your regular-expression file might appear as follows:

```
( 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 ] ;
```

Notice: the text in a regular-expression file must contain a single regular expression, and, as with regular expressions typed manually in the 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 file.

Another remark: By convention at Xerox, regular-expression filenames are given the extension .regex.

Inside a regular-expression file, it is often desirable to add comments. In xfst regular-expression files, any text from an exclamation mark to the end of the line is a comment and is ignored by the compiler. For example:

```
! This is a comment
( r e )                                          ! prefix
[ l o c k | c o r k | m a r k | p a r k ]        ! root
[ i n g | e d | s | 0 ] ;                        ! suffix
```

4.2 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.

4.3 Binary Files and Defi ned Variables

In the same way also all the defined variables can be saved into a binary file. 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.

4.4 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 the following example:

```
aardvark
apple
avatar
binary
boy
bucolic
cat
```

```
coriander
cyclops
day
```

containing a list of words, one word to a line, where each word consists of normal alphabetic characters (i.e. no multicharacter symbols). The example shows words listed in alphabetical order, but this is not required. One very 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 regular-expression file like:

```
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 ;
```

The converted file can then be compiled using the read regex command. However, xfst provides a special compiler called **read text** that takes as input a simple wordlist as shown above and compiles it into the network that encodes the language. The result, as usual, is pushed onto The Stack. For example:

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

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.

```
xfst[0]: write text > myfile.wordlist
xfst[1]:
```

If no output file is specified, then write text is effectively equivalent to print words, outputting the strings to the standard output (the terminal).

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. In such cases you should use use the **write spaced-text** and **read spaced-text** commands. For example:

```
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.

4.5 Scripts in XFST

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 commands into an xfst 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, you can execute it from within the interactive interface by using the source command.

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

By convention, script files are given the extension .xfst or .script. In general, any sequence of xfst commands that you might type manually at the command line can be put in a script file, executed with source, re-edited until the commands are correct, and then re-executed whenever you desire. The construction of some linguistic products, especially tokenizers and taggers, traditionally involves the writing of complex script files.

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
```

5 Printing Information about Networks

When dealing with non-trivial networks, size often becomes an issue. The **print size** command displays the size of a network in terms of 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”. By default, print size displays information about the top network on The Stack.

The **print stack** command displays size information about all the networks on The Stack.

The **print net** command 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. 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. Notations such as fs5 and fs6 indicate final states.