Pattern Matching with FST — A Tutorial

Lauri Karttunen
Palo Alto Research Center

November 29, 2010

Contents

1 What is FST? 2
1.1 FST as an authoring tool . . . 2
1.1.1 Insert symbols . . . 7
1.1.2 List and list membership symbols . . . 10
1.1.3 Functions . . . 13
1.1.4 Optimizations . . . 17
1.2 FST as a runtime tool . . . 20
1.2.1 apply . . . 21
1.2.2 pmatch . . . 23

2 Pattern matching with FST 26
2.1 The `pmatch` algorithm . . . 26
2.2 Context constraints . . . 30
2.3 Embedded pattern components . . . 33

3 Conclusion 34

References 34

Acknowledgements 34

Appendix 35
Plural script . . . 35
Numbers script . . . 36
Pattern script . . . 37

1 What is FST?

FST stands for *Finite-State Toolkit*. It is an application that can be launched from an `xterm` (Linux), `Terminal` (MacOS X), or `cmd.exe` (Windows) command line. Starting the application brings up the FST command line:

```
% fst
Copyright © Palo Alto Research Center 2001-2010
PARC Finite-State Tool, version 2.14.10 (libcfsm-2.22.7) (svn 33321)

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

fst[0]:
```

FST has a stack-based interface. The `[0]` part of the FST command prompt indicates the number of networks on FST's stack. Initially the stack is empty.

FST serves two purposes. It is an AUTHORING TOOL allowing the user to create networks from text files and from regular expressions. It is also a RUNTIME TOOL allowing the user to apply networks to input strings or files. Most of the FST commands are described in the chapter on the XFST application in the 2003 book on *Finite State Morphology* by Kenneth R. Beesley and Lauri Karttunen [1]. The difference between XFST and FST is that FST includes commercially valuable compressions and other optimizations for network structures and a facility for pattern matching that is the focus of this document.

In this tutorial we limit the discussion to FST commands that are relevant for creating and applying transducers. We start with a brief introduction to the use of FST as an authoring and as a runtime tool before moving on to the main topic, pattern matching with FST.

1.1 FST as an authoring tool

In this section we first show with a few simple examples how FST can be used to compile finite-state networks with the commands described in the chapter on XFST in the Beesley and Karttunen book. The new features of the current FST are explained in the sections on insert symbols, list symbols, functions and optimizations.

The command `read text` compiles a network from a list of words. The list may be in a file or it can be entered on the `fst` command line as in the example below.

```
fst[0]: read text
 One word per line. Use ^D to terminate input.
apple
orange
pear
```

```
peach
^D
1.5 Kb. 14 states, 16 arcs, 4 paths.
fst[1]:
```

When entering words for `read text` on the command line, `^D` (holding down the Ctrl key and typing D) terminates the list.[1] The new command prompt `fst[1]` indicates that we now have one network on the FST stack. The command `write dot`, or just `dot`, shows a picture of the network.[2]

```
fst[1]: write dot
```

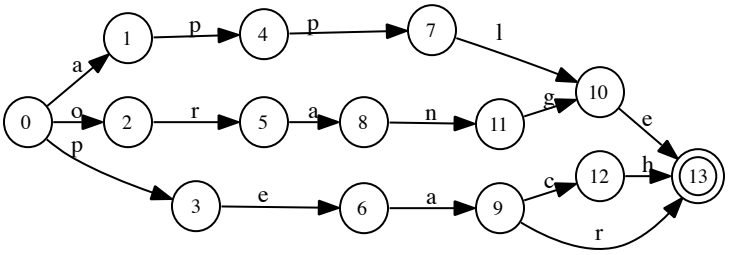

Figure 1: Fruit

It is generally useful not to keep networks on the FST stack but to give each network a name with the `define` command.

```
fst[1]: define Fruit
fst[0]:
```

The define command above pops the top network off of the stack and binds the symbol `Fruit` to that network, which encodes a language consisting of the words *apple*, *orange*, *pear* and *peach*. We can now use `Fruit` in a subsequent regular expression to represent these four words. The prompt changes to `fst[0]:` indicating that the defined `Fruit` network is no longer on the FST stack.

Another way to create a network in FST is to compile a regular expression. For example, to create a network representing dollar amounts from $0.00 to $999.99 we can compile the expression shown below and immediately give the network a name.

[1] To compile a network from a word list stored in a file, use `read text < filename`.

[2] The `write dot` command is useful only for small networks. For larger networks, use the command `print net` to display the network structure as text. `write dot` requires that you install the `graphviz` application from `http://www.graphviz.org/`. The MacOSX version is superior to the Linux and Windows versions of `graphviz`.

```
fst[0]: define digit %0|1|2|3|4|5|6|7|8|9 ;
fst[0]: define Amount "$" [digit^{1,3} - ["0" digit ?*]] ("." digit^2);
Defined 'Amount': 1.9 Kb. 8 states, 54 arcs, 101000 paths.
```

In the regular expression `"$" [digit^{1,3} - ["0" digit ?*]] ("." digit^2)` the dollar sign, zero and the period must be quoted because they are special symbols. The parentheses around `("." digit^2)` mark it as optional. The effect of `- ["0" digit ?*]` in this expression is to eliminate amounts with leading zeros such as *$001.00*.

Instead of using the `read text` command to compile a network of names, we can also define `Fruit` directly with a `define` statement:

```
fst[0]: define Fruit {apple} | {orange} | {pear} | {peach};
Defined 'Fruit': 1.5 Kb. 14 states, 16 arcs, 4 paths.
```

The curly brackets in the above definition indicate the their content should be compiled into a sequence of single-character symbols as shown in Figure 1 rather than a single multi-character name. The `read text` command always interprets the input as if each line were surrounded with curly braces.

Instead of being driven interactively from the command line, FST is usually called to execute commands from a file. We often call them SCRIPT FILEs. A script file can contain any number of FST commands and definitions. The `plural.script` given in the appendix shows how to define a function that maps singular English nouns to the corresponding plural form. The command `source` executes commands from a designated script file. The `plural.script` starts with some auxiliary definitions and ends with the definition of `Plural(X)`, a function that we can use in a regular expression. We discuss this and other function definitions in section 1.1.3 below.

```
fst[0]: source plural.script
Opening input file './plural.script'
October 27, 2010 21:54:20 GMT
Defined 'Irregular': 6.7 Kb. 95 states, 120 arcs, 27 paths.
Defined 'AddE': 4.2 Kb. 41 states, 92 arcs, 57 paths.
Defined 'Vowel': 776 bytes. 2 states, 5 arcs, 5 paths.
Defined 'FtoV': 2.1 Kb. 20 states, 35 arcs, 17 paths.
Defined 'Regular': 80.9 Kb. 113 states, 3126 arcs, Circular.
defined Plural(X).
Closing file ./plural.script...
fst[0]:
```

Having defined `Fruit` as a set of four singular nouns, we can now deploy the function `Plural(X)` in two ways: (1) to create a transducer that maps each word to the corresponding plural form and (2) to create a simple network that contains both the singular and the plural form of each fruit name. The first alternative is shown below.

```
fst[0]: regex Plural(Fruit);
1.6 Kb. 16 states, 18 arcs, 4 paths.
```

The resulting network in Figure 2 is a transducer, that, is, some transitions in the network are labeled with symbol pairs such as `0:s` where `0` represents an epsilon, an empty string. The "upper" side of the network contains the singular forms, the "lower" side the plural

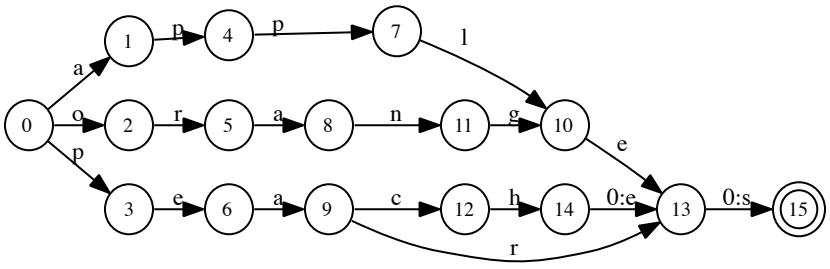

Figure 2: Fruit:Fruits

forms. The path `[p e a c h 0:e 0:s]` encodes the mapping *peach:peaches*.

The second way to use the `Plural(X)` function expands the network in Figure 1 into one that contains both singular and plural forms of each word.

```
fst[0]: regex Fruit | Plural(Fruit).l;
1.6 Kb. 17 states, 19 arcs, 8 paths.
fst[1]:
```

In this case we union the singular forms in `Fruit` with their plural counterparts on the lower side, `.l`, of the `Plural(Fruit)` network. The result is shown in Figure 3.

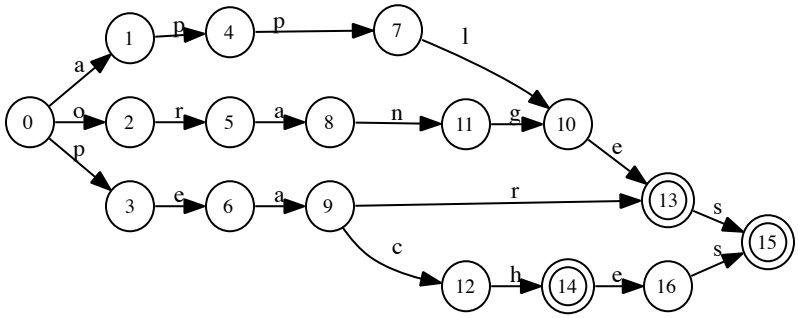

Figure 3: Fruit|Fruits

As the final variant of this example, we construct a network that maps singular nouns to themselves followed by the code `+NSg` and plural nouns to the corresponding singular form followed by the code `+NPl`. This operation yields a LEXICAL TRANSDUCER, that is, a morphological analyzer/generator for our fruit nouns.

```
fst[0]: regex Fruit "+NSg":0 | Plural(Fruit) "+NPl":0;
3.3 Kb. 18 states, 22 arcs, 8 paths.
```

The result is shown in Figure 4.

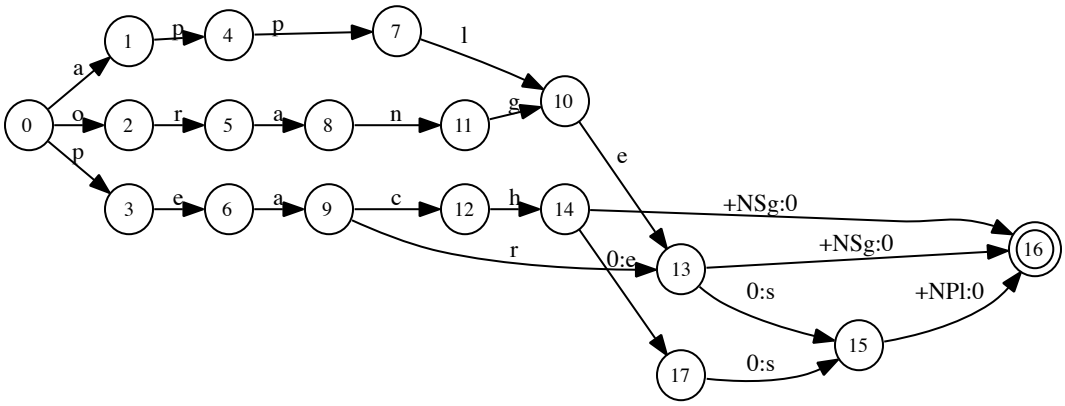

Figure 4: Lexical transducer

A lexical transducer can be thought of as a mapping between two languages. The "upper" language" consists of canonical dictionary forms of words and their morphological codes. The "lower" language contains the corresponding inflected surface forms.

```
fst[1]: print upper-words
apple+NPl
apple+NSg
orange+NPl
orange+NSg
peach+NPl
peach+NSg
pear+NPl
pear+NSg

fst[1]: print lower-words
apples
apple
```

```
oranges
orange
peaches
peach
pears
pear
```

The command `save` saves all the networks that are currently on the stack in binary format into a file. Assuming that what we have on the stack is the network in Figure 4 on our stack, we save it to a file called "fruit-lex.fst":

```
fst[1]: save fruit-lex.fst
Opening output file 'fruit-lex.fst'
Closing 'fruit-lex.fst'
```

1.1.1 Insert symbols

If `Fruit` has been defined as a network, the symbol `"@I.Fruit@"` is interpreted as a reference to that network. The initial and final `@`-signs indicate that `"@I.Fruit@"` is a special kind of a "flag" symbol. The `I` is an INSERT operator. The runtime functions treat `"@I.Fruit@"` as an instruction to enter into the `Fruit` network and exit it if a successful match is found. In regular expressions `"@I.Fruit@"` is like any other multi-character symbol, it has a special meaning only in the runtime code. To make it convenient to create insert symbols, the FST regular expression language has a function `Ins(X)` that expands to `"@I.X@"`. So in a regular expression, `Ins(Fruit)` is equivalent to `"@I.Fruit@"`.

Insert symbols are particularly useful in definitions that contain several instances of the same large component. For example, because the set of first names in English is the same as the set of middle names, a definition such as

```
define PersonName FirstName (" " FirstName) " " LastName;
```

includes two copies of the FirstName network. Because the middle name component is optional, `PersonName` includes strings such as *John Adams* and *John Henry Adams.* A more concise definition,

```
define PersonName Ins(FirstName)  (" " Ins(FirstName)) " " LastName;
```

does not physically incorporate the network of first names at all. When the `apply` routine encounters an arc labeled with an insert symbol, it "pushes" into the defined network and, on success, "pops" back to continue the match on the same level as before.

The definitions of terms referred to by an insert flag may themselves contain inserts, as the following example shows.

```
define D {the} | {this};
define N {city} | {girl} | {side} | {ocean};
define P {with} | {of} | {from} | {on};
define NP Ins(D) " " Ins(N)  (" " Ins(PP));
define PP Ins(P) " " Ins(NP);
```

Here the definition of NP refers to the definitions of D, P and PP by way of insert symbols, and the definition of PP refers back to the definition of NP, and vice versa. The PP definition includes strings such as *with the girl* and *on this side* that contain a simple noun phrase, and it also includes strings such as *with the girl from the city*, *on this side of the ocean* and *with the girl from the city on the ocean* where the NP contains one or more embedded PPs. This is an example of an RTN, a RECURSIVE TRANSITION NETWORK.

Insert symbols are properly handled by just a few network operations such as concatenation and union. Their utility is in enabling the `apply` routines to operate on smaller networks. The NP network defined above consists of just five arcs.

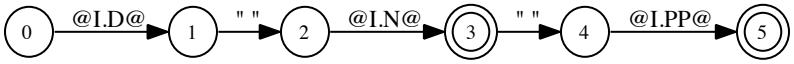

Figure 5: NP network

The FST command `eliminate flag` also works on insert symbols. For example, `eliminate flag D`, removes the `@I.D@` arc from the NP by "splicing in" a copy of the D network.

```
fst[0]: push NP
fst[1]: eliminate flag D
```

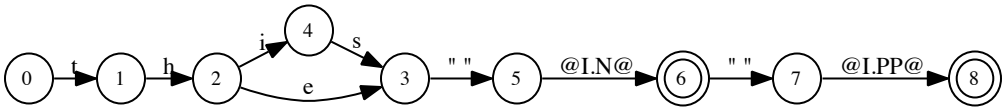

Figure 6: NP network after elimination of @I.D@

Because the definitions of NP and PP refer to each other, in this case neither one can be eliminated without resurrecting the other.

Although the NP definition in the above example is recursive, the same NP language can also be defined using iteration. The last element in the definition of NP, `(WS Ins(PP))`,

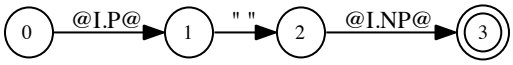

Figure 7: PP network

could be replaced by `[WS Ins(P) WS Ins(D) WS Ins(N)]*`. Any example of "tail recursion" such as this one can be replaced by iteration showing that language is in fact regular (finite-state). But recursive transition networks can represent non-regular context-free languages as the following example shows.

```
define WS " ";
define Det {a} | {the} | {an} ;
define CN  {cheese} | {cat} | {mouse} ;
define VTr {chased} | {ate} ;
define ObjRel ({that} WS) Ins(NP) WS Ins(VTr);
define Rel ObjRel;
define Nom Ins(CN) (WS Ins(Rel));
define NP Ins(Det) WS Ins(Nom);
```

In this example, the `NP` language includes strings such as *the mouse that the cat chased* as well as *the cheese that the mouse the cat chased ate*. This last phrase shows that the definition of `ObjRel` introduces an embedded `NP` in the center of another `NP`. This kind of recursion cannot be converted to iteration. Because the APPLY routine in FST correctly interprets insert symbols in such cases, it can recognize context-free languages that are not regular.

The `save` command that writes a network into a file looks for insert symbols that are in the network's alphabet and saves any defined networks along with the one on the stack.

```
fst[0]: push NP
fst[1]: save np.net
```

The `load` command puts the `NP` network on the stack and restores all the definitions it depends on.

```
fst[0]: load np.net
Redefined 'Det': 872 bytes. 5 states, 5 arcs, 3 paths.
Redefined 'Nom': 2.3 Kb. 4 states, 3 arcs, 2 paths.
Redefined 'CN': 1.2 Kb. 10 states, 11 arcs, 3 paths.
Redefined 'Rel': 2.6 Kb. 9 states, 8 arcs, 1 path.
Redefined 'NP': 2.3 Kb. 4 states, 3 arcs, 1 path.
Redefined 'VTr': 1.1 Kb. 9 states, 9 arcs, 2 paths.
fst[1]:
```

1.1.2 List and list membership symbols

A LIST in FST is a set of symbols. For example,

```
fst[0]: list Vowel a e i o u;
```

binds `Vowel` to the alphabet consisting of these five symbols. In regular expressions list symbols are interpreted as the union of the list members. Thus

```
fst[0]: regex Vowel;
```

and

```
fst[0]: regex a | e | i | o | u;
```

are equivalent expressions in FST.
It is often convenient to define lists using Perl-like range expressions. For example,

```
list Letter "A-Za-z" ;
```

defines `Letter` as the set of 52 ASCII letters. A range expression such as `"A-Za-z"` must be in double quotes. The regular expression compiler interprets `"A-Za-z"` as the union of the symbols in the range. Range expressions may be of the form `"\uHHHH-\uHHHH"` where `H` are hexadecimal numbers. For example, the range `"\u0391-\u03A1\u03A3-\u03A9"` includes all uppercase Greek letters, equivalently, in UTF-8 mode `"A-PΣ-Ω"`.[3]

The FST interface is build on top of an underlying finite-state library called `cfsm`. Table 1 below enumerates fifteen symbol lists that are predefined as part of the initialization of `cfsm`. They include symbols in Latin-1 and its extension in Microsoft's CP 1252 ("Windows Latin-1").

In addition to being interpreted as predefined unions, list symbols have another use that takes advantage of the fact that list symbols denote symbol alphabets rather than networks. If `Vowel` is defined as a list, any of the symbols on the list matches the special symbol `"@L.Vowel@"` in the runtime routines of FST. The interpretation given to `@L.Vowel@` is "any member of the list `Vowel`." The opposite symbol, `"@X.Vowel@"`, means "any symbol, but excluding all members of the list `Vowel`." The `"@L. @"` and `"@X. @"` constructions are similar to `[ ]` and `[^ ]` in Perl regular expressions except that lists may contain multi-character symbols.[4][5] The function `Lst(Vowel)` is equivalent to `"@L.Vowel@"`, `Exc(Vowel)`

[3]The range of uppercase Greek letters must be defined as two subranges because the lowercase letter σ has an alternate word-final form that has no uppercase counterpart. There is no `"\u03A2"`.

[4]Insert and list symbols are similar in appearance to the FLAG DIACRITICS discussed in Chapter 7 of the Beesley & Karttunen book but the similarity stops there. Flag diacritics are a special type of epsilon symbols that check the value of an attribute. Insert symbols refer to networks. List symbols either match or don't match a given input symbol.

[5]Perl `[ ]` and `[^ ]` expressions can contain both character ranges and single characters, e.g. `[a-zA]`, while FST range expressions cannot. It is possible in FST to put multiple ranges together between double quotes, e.g. the FST `"a-zA-Z0-9"` is equivalent to the Perl `[a-zA-Z0-9]`.

Name	Description
all	the union of the other lists
alnum	alphanumeric symbols
alpha	alphabetic symbols
ascii	ASCII symbols
cntrl	control characters
digit	0-9
graph	non-whitespace symbols
lower	lowercase alphabetic symbols
punct	punctuation symbols
upper	uppercase alphabetic symbols
space	whitespace symbols
windows	the 27 non-Latin-1 symbols in CP 1252
labr	left angle bracket
rabr	right angle bracket
delim	punctuation, brackets and other delimiter symbols

Table 1: Predefined Lists

is the same as `"@X.Vowel@"`. The advantage of "list flags" such as `"@L.Vowel@"` is that they reduce the size of networks. For example, a network that encodes any sequence of two vowels can be defined concisely as `Lst(Vowel)^2`. This regular expression compiles into a network consisting of two arcs and three states.

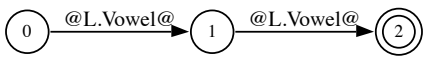

Figure 8: The language of vowel pairs encoded by `Lst(Vowel)^2`

An equivalent network defined as `Vowel^2` in Figure 9 has 10 arcs. Like insert symbols, list symbols can be eliminated from a network. The command `eliminate flag Vowel` converts the network in Figure 8 to the one in Figure 9.

When a network that contains list symbols is written into a file with the `save` command, any user-defined lists such as `Vowel` are saved with the network. When the network is loaded from the file, the list definitions are automatically restored.

```
fst[0]: regex Exc(Vowel) Lst(Vowel) Exc(Vowel);

fst[1]: save cvc.net
fst[1]: pop
fst[0]: undefine all
```

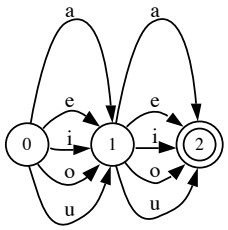

Figure 9: The language of vowel pairs encoded as `Vowel^2`

```
Undefined list 'Vowel'.

fst[0]: load cvc.net
Defined list Vowel
fst[1]:
```

All the list definitions a network depends on are saved on the property list of the network along with any regular expression used to compile it and the version numbers of FST and the cfsm library it contains.

```
fst[1]: write properties
DEFINITION: " Exc(Vowel) Lst(Vowel) Exc(Vowel);"
TOOLVERSIONS: ( ( "fst" "2.14.11" ) ( "libcfsm" "2.22.7" ) )
DEFINED_LISTS: ( ( "Vowel" "a" "e" "i" "o" "u" ) )
```

Instead of defining `Vowel` as a list, we could alternatively define `Vowel` as a network and use an insert symbol, `"@I.Vowel@"`, to encapsulate the union of vowel symbols:

```
define Vowel a | e | i | o | u;
```

Whenever there is a choice of defining something as a list or as a network, it is more efficient to define it as a list. Lists are internally represented as a vector of symbols. It is faster to check list membership than to find a matching arc in a state of a network.

Insert and list symbols are useful in networks that are applied to text. Only a few calculus functions such as concatenation, union and composition know about list symbols. All other FST algorithms treat them as ordinary symbols with no special meaning. The earlier definition of `Amount`

```
fst[0]: define Amount "$" [digit^{1,3} - ["0" ?*]] ("." digit^2);
```

subtracts any sequence starting with a zero from digit sequences ranging from one to three in length. Replacing `digit` by `"@L.digit@"` in this expression would not yield the intended result because the FST subtraction operator, `-`, does not know about list membership symbols. The expression `[Lst(digit)^{1,3} - ["0" ?*]]` is equivalent to

`"@L.digit@"^{1,3}` because a path starting with `"@L.digit@"` does not start with a zero as far as subtraction is concerned. Consequently, the subtraction operation has no effect in this case.

1.1.3 Functions

Functions are regular expression constructs that encapsulate a regular expression containing the arguments, evaluate it, and return the resulting network. For example, the command

```
define CrossP(X) [X %. X ];
```

defines a function that duplicates its argument and marks the center with a period. Given this definition of `CrossP(X)`, the regular expression compiler interprets `CrossP(p i g)` as `p i g . p i g`.[6] The PigLatin example below shows how function definitions can use other defined functions to produce complex behavior.

```
# This script defines the function PigLatin() that translates
# English strings to Pig Latin. For example, PigLatin(p i g)
# yields the string igpay.

# We need a list of consonants Cons and a list of vowels Vow.
list Cons b c d f g h j k l m n p q r s t v w x y z;
list Vow a e i o u;

# CrossP(X) builds a crossproduct of X marking the center
# with a period.
define CrossP(X) [X %. X];

# AddW(X) adds an initial w to the duplicated word if it begins
# with a vowel
define AddW(X) [ X .o. %. -> {.w} || _ Vow];

# DelCons(X) deletes any initial consonants from X.
define DelCons(X)[X .o. Cons+ @-> 0 || .#. _ ] ;

# TailToAy(X) replaces any trailing vowels of X with ay;
define TailToAy(X) [X .o. Vow ?* @-> {ay} || %. Cons* _ ];
```

[6] The function definition is bound to a multicharacter symbol consisting of the function name and the opening left paren, `CrossP(` in this case. There must not be any whitespace between the name and the left paren as they are parsed as a single symbol.

```
# DelMiddle(X) Deletes any periods in X.
define DelMiddle(X) [X .o. %. -> 0];

# PigLatin(X) applies the functions defined above in the given
# order and extracts the lower side of the result.

define PigLatin(X) [DelMiddle(TailToAy(DelCons(AddW(CrossP(X)))))].l;

read regex PigLatin(p i g);
```

A function can have any number of parameters, separated by commas. For example,

```
define Wrap(X, Y) Y X Y;
```

yields a function that maps `Wrap({abc}, %")` into `"abc"`. Every parameter must occur at least once in the regular expression defining the function. It is also possible to define a function with no arguments. The difference between

```
define Foo A B;
```

and

```
define Foo() A B;
```

is that in the first case `A` and `B` are evaluated at the point where the symbol `Foo` is defined, whereas in the case of `Foo()`, `A` and `B` are evaluated when a regular expression containing `Foo()` is compiled.

There is a large number of predefined functions for case conversion, symbol manipulation and alphabet-to-network coercions. They are listed in Table 2. The optional `side` argument is either `U` for upper, `L` for lower or `B` for both sides. The side argument makes a difference only for transducers. The default is `B`. Here are some examples of built-in case conversions and symbol manipulations. The argument to a case conversion function can be any regular expression. If `Fruit` is defined as a network, `UpCase(Fruit)` uppercases all its words.

```
UpCase({new york}) ==> NEW YORK
UpCase(a:b)        ==> A:B
UpCase(a:b, L)     ==> a:B
OptUpCase(a:b)     ==> A:B, a:b
Cap({new york})    ==> New York
OptCap({new york}) ==> New York, New york, new York, new york
Explode(NP)        ==> N P (two single-character symbols)
Implode(N,P)       ==> NP (a multicharacter symbol)
```

The case-conversion functions work on all alphabets that make a distinction between upper and lowercase letters, including Greek, Cyrillic, etc.

UpCase(X [,side])	Returns a network that recognizes the all-uppercase versions of X. If `side` U or L is specified, then only the indicated side of the transducer is uppercased.
OptUpCase(X [,side])	Same as for UpCase(X [,side]) but optionally uppercases each symbol in X.
DownCase(X [,side])	Returns a network that recognizes the all-lowercase versions of X. If `side` U or L is specified, then only the indicated side of the transducer is lowercased.
OptDownCase(X [,side])	Same as for DownCase(X [,side]) but optionally lowercases each symbol in X.
Cap(X [,side])	Same as UpCase(X [,side]) but with initial-uppercasing only, with all other characters lowercased.
OptCap(X [,side])	Same as Cap(X, [,side]), but optionally.
AnyCase(X [,side])	Returns a network that contains X with all case variations.
Explode(X [,Y [, Z ...]])	Returns a network consisting of a single path that converts any multicharacter symbols into a sequence of single symbols and concatenates them.
Implode(X [, Y [, Z ...]])	Returns a network with a single arc labeled by a symbol that compacts the arguments into one label. For example `Implode(a, b, cd)` produces a network consisting of an arc labeled `abcd`.
Sigma(X)	Returns a network consisting of the sigma alphabet of X. For example `Sigma(a:b c:0 0:d a:b)` returns a network equivalent to `[a|b|c|d]`.
Labels(X)	Returns a network consisting of the label alphabet of X. For example `Labels(a:b c:0 0:d a:b)` returns a network equivalent to `[a:b|c:0|0:d]`.
Lit(X)	Returns X as a literal symbol even if it has a definition.
Ins(X), Lst(X), Exc(X)	Return `"@I.X@"`, `"@L.X@"` and `"@X.X@"`, respectively.
EndTag(X)	Returns `"</X>"`:0

Table 2: Predefined Functions

As we shall see below, labeled patterns for pattern matching must be terminated with a label consisting of a multicharacter symbol that looks like an XML end tag on the upper side, and epsilon (the empty string) on the lower side. The symbol manipulation functions can be useful in manufacturing such labels. For example, the `EndTag(X)` function

```
define EndTag(X) Implode("</", X, ">"):0 ;
```

converts `EndTag(Person)` into the label `</Person>:0`, which is exactly what is needed to mark the end of a labeled pattern. The argument of `EndTag(X)` is interpreted literally even if the symbol has a definition.

An example of a useful user-defined function, `Plural(X)`, is given below and as an appendix.

```
define Irregular [{child}:{children} | {man}:{men} | {woman}:{women} |
                  {ox}:{oxen} | {bacterium}:{bacteria} |
                  {corpus}:{corpora} | {criterium}:{criteria} |
                  {curriculum}:{curricula} | {datum}:{data} |
                  {genus}:{genera} | {medium}:{media} |
                  {memorandum}:{memoranda} | {stratum}:{strata} |
                  {phenomenon}:{phenomena} | {deer} | {fish} |
                  {offspring} | {sheep} | {foot}:{feet} |
                  {goose}:{geese} | {tooth}:{teeth} |
                  {louse}:{lice} | {mouse}:{mice} |
                  {crisis}:{crises} | {thesis}:{theses} |
                  {oasis}:{oases} | {hypothesis}:{hypotheses}];
define AddE f | s | {sh} | {ch} | x | z |
            {acco}|{alvo}|{ando}|{ango}|{anjo}|{argo}|{asco}|{asso}|
            {atto}|{cano}|{cebo}|{digo}|{echo}|{ecko}|{egro}|{endo}|
            {ento}|{esco}|{esto}|{etto}|{falo}|{fico}|{hako}|{halo}|
            {hero}|{hobo}|{ildo}|{illo}|{ingo}|{into}|{irdo}|{lago}|
            {lico}|{mago}|{mato}|{mino}|{nado}|{nkgo}|{otto}|{pedo}|
            {rado}|{rago}|{tato}|{tico}|{tigo}|{ucco}|{uito}|{utgo}|
            {veto}|{viso}|{zebo};
define Vowel a | e | i | o | u ;
define FtoV {cal}|{dwar}|{el}|{hal}|{hoo}|{kni}|{lea}|{li}|{loa}|{scar}|
            {sel}|{shea}|{shel}|{thie}|{whar}|{wi}|{wol};
define Regular [..] -> %.    || _ .#.                .o.
                 %. -> {es}  || AddE _ .#.           .o. # buses, heroes
               y %. -> {ies} || \Vowel  _ .#.        .o. # ladies
                 %. -> s     || _ .#.                .o. # days, pianos
                 f  -> v     || .#. FtoV _ {es} .#. ;    # wolves
define Plural(X) [[X .o. Irregular] .P. [X .o. Regular]].l;
```

The definition of `Irregular` lists a number of irregular pairs such as *mouse:mice*. The definition of `Regular` compiles into a transducer that maps English words to their regular plurals: *cat:cats*, *bus:buses*, *hero:heroes*, *lady:ladies*, *wolf:wolves*, and so on. If `X` is a regular

word, `Plural(X)` is its regular plural form but if `X` is one of the listed irregular words, `Plural(X)` will be its irregular plural. For example, `Plural({child})` is *children* but `Plural({wife})` is *wives*. This is the effect of the PRIORITY UNION operator, `.P.`, that prefers the mapping `[X .o. Irregular]` if it is non-empty and falls back on the mapping `[X .o. Regular]` only if `X` has no irregular plural, that is, when the composition of `X` with the `Irregular` relation yields a null result.

The `Plural(X)` function can be applied not just to individual words but to a list of words of any size. For example,

```
fst[0]: regex Plural({child}|{wife}|{ship}|{sheep}|{crisis}|{ferry});
fst[1]: print words
wives
ferries
ships
sheep
crises
children
```

1.1.4 Optimizations

The internal representation of states and arcs in FST can be modified to reduce the size of the network or to improve the speed of applications.

arc optimization The FST command `optimize net` runs a heuristic algorithm that tries to reduce the number of arcs, possibly at the cost of creating more states. The effect varies greatly depending on the arcs/states ratio of the original net. Dense networks can sometimes be dramatically reduced in size by encoding the arc space in a different way. In the case shown below, arc optimization reduces the size of the network from 25 megabytes to 3 megabytes.

```
fst[1]: print size
25.0 Mb. 6980 states, 1032936 arcs, Circular.
fst[1]: optimize net
OPTIMIZER INPUT SIZE: 25.0 Mb. 6980 states, 1032936 arcs, Circular.
20844 arcs reordered.
OPTIMIZER OUTPUT SIZE: 3.0 Mb. 11370 states, 108245 arcs, Circular.
  Arcs: -89.5% (-924691)
  States: +62.9% (+4390)
3.0 Mb. 11370 states, 108245 arcs, Circular.
```

The command `unoptimize net` restores an optimized network into its original configuration and size. Arc optimization is lossless and has no effect on the behavior of runtime

applications.[7]

In the case of very sparse networks such as `Fruit` in Figure 1, arc optimization has no effect at all. Sometimes it has only a modest effect, as illustrated below.

```
fst[0]: regex [a|b]* a [a|b];
896 bytes. 4 states, 8 arcs, Circular.
fst[1]: optimize net
OPTIMIZER INPUT SIZE: 896 bytes. 4 states, 8 arcs, Circular.
0 arcs reordered.
OPTIMIZER OUTPUT SIZE: 848 bytes. 4 states, 6 arcs, Circular.
  Arcs: -25.0% (-2)
  States: +0.0% (+0)
848 bytes. 4 states, 6 arcs, Circular.
```

Figure 10 shows the network in its standard representation.

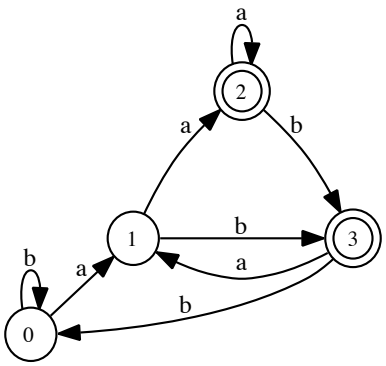

Figure 10: `[a|b]* a [a|b]` before arc optimization

Figure 11 shows the optimized result. As the latter figure shows, the optimization algorithm discovers that two arcs in states 2 and 3 can be eliminated by introducing one special EPSILON arc in each state. Thus the net gain is two arcs. The special epsilon symbol is displayed as *ALTCHAIN* by the `print sigma` and `print net` commands. An optimized network can be saved and loaded from a file without losing the optimization.

When an optimized network is applied, the memory footprint can be reduced even further by eliminating the *ALTCHAIN* arcs altogether. The runtime algorithms, `apply`

[7]The heuristic algorithm invoked by the `optimize net` command has never been publicly disclosed although it has been used in products for decades. When the idea was invented and implemented by Ronald M. Kaplan in the 1980s, PARC decided not to apply for a patent but to retain the algorithm as a "trade secret." The fundamental insight is that reducing the number of arcs is often more valuable than minimizing the number of states because in some storage representations the arcs of a network commonly take up more memory than the states. The classic textbook algorithms for minimization by Brzozowski [2] and Hopcroft [3] aim to reduce the number of states.

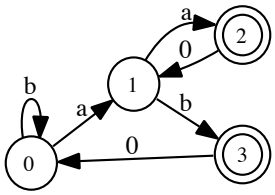

Figure 11: `[a|b]* a [a|b]` after arc optimization

`up/down` and `pmatch` do this automatically. The command `share arcs` does the same on the FST command line. In the shared arcs configuration, the two outgoing arcs in state 1 are shared with state 2 and the two outgoing arcs in state 0 are also in state 3. As Figure 11 shows, states 0 and 3 are equivalent in terms of what arcs they have and where the arcs lead to. The same goes for states 1 and 2.

```
fst[1]: share arcs
fst[1]: print size
800 bytes. 4 states, 4 arcs, Circular.
```

In this case we have eliminated four of the original eight arcs without adding any new states. In the case of our first example, we get another reduction of size by 10747 arcs down to 2.6 megabytes, an order of magnitude below the original size.

```
fst[1]: share arcs
fst[1]: print size
2.6 Mb. 6980 states, 97498 arcs, Circular.
```

state vectorization In the standard representation, a state consists of a list of arcs. An arc has a label, a pointer to a destination state and a pointer to the next arc. Most FST algorithms work only on the standard representation. The command `vectorize net` eliminates arc lists and replaces them with a vector of destination pointers that is accessed directly by symbol labels. Vectorization improves the speed of runtime applications but state vectors typically take up more space than arc lists. By default, vectorization is applied to states that have 50 or more outgoing arcs and other states retain their arc lists. The following example illustrates the effect of vectorization on one large but not very dense network.

```
fst[1]: print size
12.2 Mb. 156406 states, 300326 arcs, Circular.
fst[1]: vectorize net
```

```
Vectorized 25 states out of 156406 total.
12.6 Mb. 156406 states, 298433 arcs.
```

Because only 25 states had 50 or more outgoing arcs, the size of the network did not change much. The command `unvectorize net` restores a network to its standard representation.

```
fst[1]: unvectorize net
Unvectorized 25 states out of 156406 total.
12.2 Mb. 156406 states, 300326 arcs, Circular.
```

If we lower the setting of `vectorize-n`, more states are affected and the size of the network grows.

```
fst[1]: set vectorize-n 10
fst[1]: vectorize net
Vectorized 4553 states out of 156406 total.
71.4 Mb. 156406 states, 235621 arcs
```

Table 3 illustrates the effect of the two optimization methods on the size of a network and the speed of the application. The network in question is a "sentence breaking" transducer that introduces sentence boundaries. Its large size is due to the long lists of abbreviations, numerical expressions and other expressions that are exceptions to general punctuation rules. The application speed was measured on a 710K text file.

Representation	States/Arcs	Size	Speed
Standard, no optimization	5302/596017	7.2 Mb	23 sec.
Arc optimization	8364/83301	1.1 Mb	12 sec
State vectorization	5302/5312	6.9 Mb	2 sec

Table 3: Network size vs. speed of application

Because this network is very dense, vectorizing actually reduces its size from the original 7.2 Mb.

A vectorized network cannot be saved into a file without first unvectorizing it.

1.2 FST as a runtime tool

A network that is on the FST stack can be applied to text. In this section we introduce two kinds of apply commands, simple `apply up/down` and `pmatch` (a.k.a `apply patterns up`).

1.2.1 apply

If we have just defined a network, say `Fruit`, we can push a copy of it to the stack.

```
fst[0]: push Fruit
fst[1]:
```

The `apply` command comes in two versions: `apply up` and `apply down`. If the network on the stack is a simple network compiled from a list of words like the `Fruit` network, both `apply` commands have the same effect. If the input word matches a path in the network, the word is echoed, if the input word is not recognized the output is `???`. The two `apply` commands can be abbreviated to `up` and `down`.

```
fst[1]: apply up apple
apple
fst[1]: down peach
peach
fst[1]: up plum
???
fst[1]: apply down plum
???
```

The command `pop` pops the top network off the stack, `clear` empties the entire stack.

```
fst[1]: pop
fst[0]:
```

Another way to start applying a network is to load it from a file:

```
fst[0]: load fruit-lex.fst
fst[1]:
```

Because this network, shown in Figure 4, is a transducer, the two `apply` commands have a different effect. The `apply up` command tries to match the input string against the lower side of the network and, if the word is recognized, the output is the upper side of the matching path. The `apply down` command works in the opposite way.

```
fst[1]: apply up apples
apple+NPl
fst[1]: apply down apple+NPl
apples
fst[1]: down peach+NPl
peaches
```

To put it in linguistic terms, if the network is a lexical transducer of the kind described in [1], `apply up` gives all the analyses of an inflected surface form and `apply down` generates one or more surface forms from a canonical dictionary form and a morphological code. But this is not a limitation on the kind of output that `apply up/down` command produces. Finite-state transducers can encode any kind of regular string-relation. For example, the `numbers.script` in the appendix creates a transducer that maps the numbers from *1* to *99* to the corresponding English numerals from *one* to *ninety-nine*, and vice versa. If we launch FST with the command `fst -l numbers.script`, it creates the numbers/numerals transducer and applies it to a couple of test cases.

```
fst -l numbers.script
Defined 'OneToNine': 2.3 Kb. 25 states, 32 arcs, 9 paths.
Defined 'TeenTen': 1.8 Kb. 19 states, 23 arcs, 6 paths.
Defined 'Teens': 2.8 Kb. 34 states, 42 arcs, 10 paths.
Defined 'TenStem': 2.0 Kb. 21 states, 27 arcs, 8 paths.
Defined 'Tens': 3.8 Kb. 49 states, 64 arcs, 80 paths.
Defined 'OneToNinetyNine': 6.0 Kb. 81 states, 113 arcs, 99 paths.

push OneToNinetyNine

apply down 75
seventy-five

apply up thirteen
13

Closing file numbers.script...
```

The command `apply up < numerals.txt` applies the network to the file *numerals.txt*. The file is processed line-by-line. The input lines and the output lines are echoed to the console.

```
fst[1]: apply up < numerals.txt

twelve
12

fourteen
14

seventy-five
75
```

The command `apply up < numerals.txt > output.txt` directs the output into a file. If an `apply up/down` command is entered without an input word or a file directive, FST enters into a loop prompting for input from the console. The input `END;` stops the loop and brings back the FST command line. In the example below we compile the `Amount` expression in the previous section leaving the resulting network on the stack for immediate application.

```
fst[0]: regex "$" [digit^{1,3} - ["0" digit ?*]] ("." digit^2);
1.9 Kb. 8 states, 54 arcs, 101000 paths.
fst[1]: apply up
apply up> $1
$1
apply up> $0.15
$0.15
apply up> $99.05
$99.05
apply up> 20
???
apply up> END;
fst[1]:
```

1.2.2 pmatch

The `pmatch` command (equivalently `apply patterns up`) is like `apply up` in that it takes input from the FST command line or a file and sends the output either to the console or into a file. The input from the console is processed line by line, but an input file is processed as a continuous stream of characters.

A network for pattern matching contains one or more patterns, unioned together, written using the standard FST syntax. Typically, but not necessarily, the patterns are *labeled*. A labeled pattern for pmatch ends with a label that has a multicharacter symbol on the upper side that looks like an XML end tag, e.g. `</foo>`, and epsilon on the lower side. In the example below we construct a pattern-matching network for recognizing prices and fruits. The function calls `EndTag(Amount)` and `EndTag(Fruit)` create the labels `"</Amount>":0` and `"</Fruit>":0`. Such labels are not required but are useful to indicate the type of entity that has been recognized.

```
fst[0]: list 1to9 "1-9";
fst[0]: define Amount ["$" | €] ["0" | Lst(1to9) Lst(digit)^{0,2}]
                      ("." Lst(digit)^2);
fst[0]: define Fruit {apple} | {orange} | {pear} | {peach};

fst[0]: regex Fruit EndTag(Fruit) | Amount EndTag(Price);
```

```
fst[1]: pmatch An apple costs $0.50 today.
An <Fruit>apple</Fruit> costs <Price>$0.50</Price> today.
```

The default output from `pmatch` marks each instance of a pattern with an initial and a final XML tag provided that the patterns have a final end tag. The corresponding start tag is created on the fly. In the default mode, parts of the input that are not part of any pattern are echoed into the output unchanged. The output behavior of `pmatch` is controlled by the eight FST variables in Table 4.

VARIABLE	DEFAULT	DESCRIPTION
count-patterns	OFF	count all pattern matches
delete-patterns	OFF	replace the matching string by an initial tag
extract-patterns	OFF	delete anything that does not match a pattern
locate-patterns	OFF	locate start and end positions of each match
mark-patterns	ON	mark matching strings with XML tags
max-context-length	254	maximal context length
max-recursion	5000	maximum recursion in applying a pattern
need-separators	ON	check for separators at starts and ends

Table 4: `pmatch` variables

The values of these variables can be set on the FST command line. For example, if `count-patterns` is set to ON `pmatch` prints statistics at the end of the input on how many instances of each pattern it found.

```
fst[1]: set count-patterns on
fst[1]: pmatch An apple costs $0.50 today.
An <Fruit>apple</Fruit> costs <Price>$0.50</Price> today.
Pattern # of matches
------------------------
<Fruit>      1
<Price>      1
------------------------
Total:       2
```

The following examples illustrate how the variable settings affect the output behavior of `pmatch`.

```
fst[1]: set count-patterns off
fst[1]: set delete-patterns on

fst[1]: pmatch An apple costs $0.50 today.
```

```
An <Fruit> costs <Price> today.

fst[1]: set delete-patterns off
fst[1]: set extract-patterns on

fst[1]: pmatch An apple costs $0.50 today.
<Fruit>apple</Fruit>
<Price>$0.50</Price>

fst[1]: set mark-patterns off
fst[1]: pmatch An apple costs $0.50 today.
apple
$0.50

fst[1]: set mark-patterns on
fst[1]: set locate-patterns on

fst[1]: pmatch An apple costs $0.50 today.
3|5|apple|<Fruit>
15|5|$0.50|<Price>
```

When the variable `locate-patterns` is set to ON, `pmatch` reports the beginning byte position of each match (starting with 0) and the number of bytes matched. If the input is UTF-8 encoded, the number of bytes may be larger than the number of visible symbols on the screen. For example, if we in our example change the currency from dollars to euros, the byte count comes out differently.

```
pmatch An apple costs €0.35 today.
3|5|apple|<Fruit>
15|7|€0.35|<Price>
```

The length of *€0.35* is 7 bytes because the UTF-8 encoding of the Euro symbol € contains three bytes.

The variable `need-separators` controls where `pmatch` can start and finish matching a pattern. With the default setting ON, the `pmatch` assumes that a pattern that starts and ends with a number should be surrounded by non-digits. It does not start matching a pattern that begins with an alphabetic symbol unless it has just seen a non-alphabetic symbol. For example, in the following case we get no matches. The `apple` pattern cannot start and end in the middle of *crabapples* and the three-digit dollar amount `$100` cannot end in the middle of *$1000*.

```
fst[1]: pmatch The crabapples cost $1000.
The crabapples cost $1000.
```

An ill-advised move of setting `need-separators` to OFF removes this context constraint.

```
fst[1]: set need-separators off
fst[1]: pmatch The crabapples cost $1000.
The crab<Fruit>apple</Fruit>s cost <Price>$100</Price>0.
```

The next section presents more sophisticated ways of controlling the context of a match.

The variable `max-recursion` is a limit on the size of a pattern. The `pmatch` algorithm discussed in the next section is recursive. Although `pmatch` does not call itself recursively but maintains its own internal recursion stack, it can still run out of memory in trying to match patterns that are unbounded in length.

The function of the `max-context-length` variable is explained in Section 2.2.

2 Pattern matching with FST

This section of the tutorial begins with an informal description of the FST pattern matching algorithm. The following sections give examples of matching with complex patterns, methods for constraining the context for valid matches, and for modifying the output.

2.1 The `pmatch` algorithm

The `pmatch` algorithm is based on four simple ideas.

1. The left-to-right longest-match principle.
2. Only the end of a pattern needs to be labeled.
3. Multiple patterns can be matched in parallel.
4. The same string can be a match for more than one pattern.

If an input symbol matches the first symbol of a pattern the `pmatch` algorithm fetches the next symbol from the input and continues as long as it can follow a path in the pattern network. If a valid match is found, the `pmatch` algorithm produces an output and continues starting at the end of the matching string. If the match is unsuccessful, `pmatch` moves one symbol to the right in the input and tries again. The left-to-right longest-match principle entails that a match is valid only if a longer match cannot be found. We illustrate this principle with the network constructed below and shown in Figure 12.

```
fst[0]: list 1to9 "1-9";
fst[0]: regex ["$" | €] ["0" | Lst(1to9) Lst(digit)^{0,2}]
              ("." Lst(digit)^2);
```

For example, when `pmatch` applies the `Price` pattern depicted in Figure 10 to the input *The apples cost $100*, it starts with the first letter of input but does not get anywhere because in state 0 of the network there is no transition for *T*. Because of the

`need-separators` condition, the next starting point for a match is at the first letter of *apples*. All the input up to that point is echoed to the output when the variable `extract-pattern` is OFF. Because there is no transition for *a* in state 0, `pmatch` echoes the input to the output symbol by symbol until it reaches the dollar sign in *$100*. The path

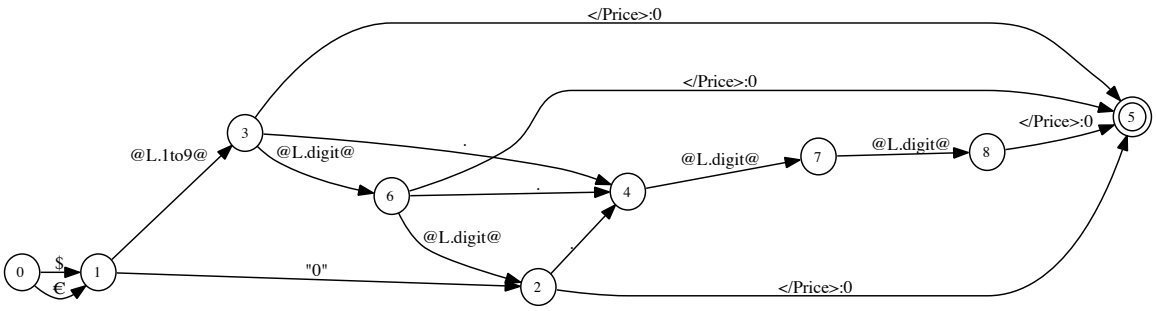

Figure 12: The Price pattern

`0-$-1-@L.1to9@-3-</Price>:0-5`, that is, the string *$1*, would be a match for the pattern except that it fails the `need-separators` condition. In any case, even with that constraint turned off, `pmatch` would not output `<Price>$1</Price>` because it finds a longer path, `0-$-1-@L.1to9@-3-@L.digit@-6-</Price>:0-5` matching the string *$10*, and finally the path `0-$-1-@L.1to9@-3-@L.digit@-6-@L.digit@-2-</Price>:0-5` for the string *$100*. Whenever `pmatch` finds a longer match for the input, it discards any previous shorter result.

When `pmatch` finds that the same string matches two or more patterns, it gives all the analyses nested inside one another. For example, the network in Figure 13 tags the string *1/4* both as a date and as a fraction. Both analyses are shown in the output.

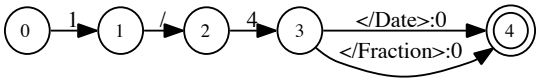

Figure 13: Ambiguity

```
fst[0]: regex {1/4} [EndTag(Date) | EndTag(Fraction)];
fst[1]: pmatch 1/4
<Date><Fraction>1/4</Fraction></Date>
```

This example shows the utility of marking only the end of a pattern with a closing XML tag and generating the initial tag on the fly. If the pattern network was defined as in Figure 14, the network would be larger and `pmatch` would have to process the string *1/4* twice.

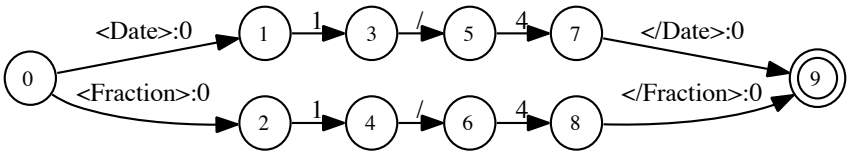

Figure 14: Wasteful encoding of ambiguity

Some pattern networks may contain spurious ambiguities. For example, if the arc from state 1 to state 3 in Figure 12 had the label `@L.digit@` instead of `@L.1to9@`, `pmatch` would discover that the input *$0* matches two paths, `0-$-1-@L.digit@-3-</Price>:0-5` and `0-$-1-"0"-2-</Price>:0-5`. In a case like this, `pmatch` disregards the spurious ambiguity and returns a single result: `<Price>$0</Price>` instead of a redundantly nested output `<Price><Price>$0</Price></Price>`. If a string is already marked for being a match for a particular pattern, `pmatch` will ignore any subsequent matches for the same end tag. In this case `apply up` gives two identical results but `pmatch` discards the duplicate match.

```
fst[0]: regex ["$" € ] ["0" | Lst(digit)^{0,3}]
              ("." Lst(digit)^2) EndTag(Price);
fst[1]: apply up $0
$0</Price>
$0</Price>
fst[2]: pmatch $0
 <Price>$0</Price>
```

The `pattern.script` in the appendix is a simple example of how various pattern components can be compiled and unioned into a single network . The script compiles lists of actors, movies and dictators from text files and two regular expression files for phone and social security numbers. Assuming that we have the following small files.

```
# Actor.txt
Charlie Chaplin
Chaplin
Grace Kelly
Paul Newman
Gary Grant
```

```
# Dictator.txt
Sadam Hussein
Adolf Hitler
Hitler
Stalin
Mussolini
```

```
# Movie.txt
The Great Dictator
To Catch a Thief
North by Northwest
The Sting
Limelight
```

```
#PhoneNum.re
("(" Lst(digit)^3 ")" (" ")) Lst(digit)^3 "-" Lst(digit)^4 ;
```

```
#SocSecNum.re
Lst(digit)^3 "-" Lst(digit)^2 "-" Lst(digit)^4 ;

#input.txt
My social security number is 123-45-6789. You can call me at (650) 812-
4567. Grace Kelly and Gary Grant starred in "To Catch a Thief".
Charlie chaplin played Hitler in The Great Dictator.
```

The script compiles the patterns, saves the result into a file and applies it to a `input.txt` with the `locate-patterns` set to ON. The script can be run from the Unix command line by starting FST with the optional `-f` flag that executes the commands in a file and exits.

```
% fst -f pattern.script
62|15|(650) 812-4567|<PhoneNum>
79|11|Grace Kelly|<Actor>
95|10|Gary Grant|<Actor>
118|16|To Catch a Thief|<Movie>
137|15|Charlie Chaplin|<Actor>
160|6|Hitler|<Dictator>
170|18|The Great Dictator|<Movie>
```

As this example shows, a pattern matching transducer may modify its input. The `pattern.script` compiles the `Actor.txt` source file with the command

```
define Actors OptCap(@txt "Actor.txt", L);
```

The function invocation `OptCap(@txt "Actor.txt", L)` makes the capitalization of actor names optional on the input side (lower) but retains capitalization on the output side. The transducer includes a path that recognizes the input *chaplin* mapping it to *Chaplin* as well as a path that matches and outputs *Chaplin* literally. The transducer is FUNCTIONAL in the lower-to-upper direction, it maps any input to at most one output.

If we replaced the `L` argument of `OptCap` by `B` meaning "both sides" the result would be a non-functional transducer. It would recognize both *chaplin* and *Chaplin* and map each of them to both *chaplin* and *Chaplin*. The behavior of `pmatch` is unpredictable in such cases. If the transducer is non-functional, `pmatch` will produce only one output, the first one it finds under a particular ordering of the arcs. Other outputs for the same input with the same output tag are ignored. This behavior of `pmatch` may be changed in a future version of the application.

The second input modification made in `pattern.script` comes from the composition of the original network with a replace rule:

```
Pattern .o. [..] (->) "\n" || "-" _ ;
```

This rule inserts optionally a newline character after a dash on the input side. This allows a line break after a dash, that is, an optional `0:"\n"` label in the pattern. The result is that the phone number *(650) 812-4567* is written into the output without showing the line break that is nevertheless recorded in the byte count.

2.2 Context constraints

A pattern definition may include constraints on the context. A constraint is a condition that has to be met for a string to count as a valid match for a pattern. For example, the ambiguous pattern for tagging *1/4* as a date or as a fraction in Figure 13 could be tightened by adding the requirement that *1/4* can be interpreted as a fraction only when followed by a preposition such as *of*. This can be implemented by adding a right-context condition to the Fraction tag: `EndTag(Fraction) RC({ of})`. The expression `RC({ of})` is a shorthand notation for `"*RIGHT_CONTEXT*" { of})`. It means that the fraction match is valid only when followed by a space and *of*.

But this is not quite enough. We also need to prevent the *1/4* from being interpreted as a date in this context. This requires a negative context: `EndTag(Date) NRC({ of})` where `NRC({ of})` abbreviates `"*NON_RIGHT_CONTEXT*" { of})`. Figure 15 shows the resulting structure.

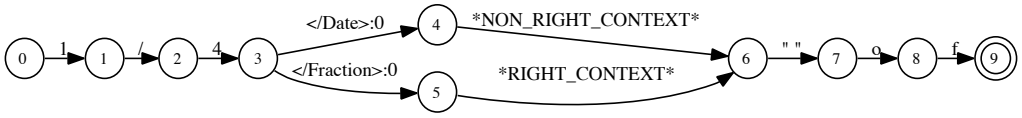

Figure 15: Right context constraints

When `pmatch` reaches a closing XML tag that does not lead to a final state, it expects to find a context condition. In the case of positive or negative right-context condition it will scan the input further trying to find a match in the constraint network, a single word in this case. If the match succeeds and the starting symbol was `"*RIGHT_CONTEXT*"`, we have a valid match. If the starting symbol was `"*NON_RIGHT_CONTEXT*"`, success in matching the constraint is a failure. The network in Figure 15 now always gives an unambiguous result:

```
fst[1]: pmatch 1/4 of the people
<Fraction>1/4</Fraction> of the people

fst[1]: pmatch 1/4 or before
<Date>1/4</Date> or before
```

We can also constrain the left context of a match. To make sure that *1/4* is not misanalyzed as a date, we could require that it be preceded by an expression such as *due on*, *night of*, or more generally as `DayPeriod { of }` where `DayPeriod` is defined as `[{morning}|{evening}|{eve}|{night}]`. This is expressed as `LC(DayPeriod { of } | {due on })`. Because matching the left context proceeds backwards from the start of the potential match, the `LC` function actually reverses its argument producing the structure in Figure 16 that presents the symbols in the order they would be found going right-to-left.

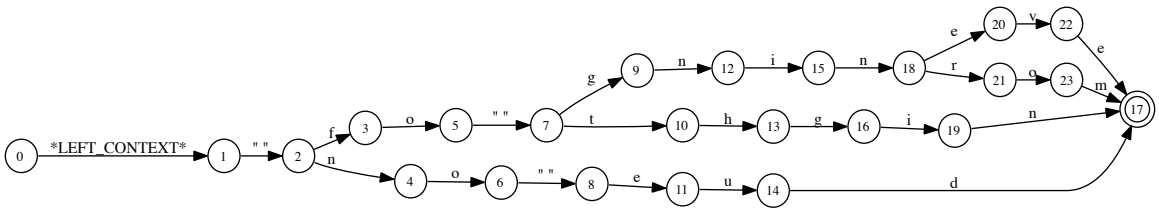

Figure 16: Left context in reverse

Instead of compiling the `DayPeriod` definition directly into the context constraint, it would be desirable to use an insert symbol `Ins(DayPeriod)`, that is, `"@I.DayPeriod@"`, but FST does not currently support the use of insert symbols in context constraints. Insert symbols are interpreted properly in the body of a pattern definition.

We also need negative left-context constraints. For example, *1/4* should not be tagged as a date when preceded by a word such as *under*. This is expressed as `NLC({under })`. The `NLC` function introduces a `"*NON_LEFT_CONTEXT*"` symbol. Like the `LC` function, `NLC` reverses its argument because the matching works right-to-left from the start of a potential match.

Simple context conditions can be combined into Boolean conditions with `AND` and `OR` functions. For example, the expression below stipulates that *1/4* cannot be a date if it is either preceded by *under* or followed by *of*. That is, to qualify as a date, *1/4* has to satisfy both the constraint that it is not preceded by *under* and that it is not followed by *of*.

```
fst[0]: regex {1/4} EndTag(Date) AND(NLC({under }), NRC({ of}));
```

The situation is the opposite for the competing `Fraction` interpretation. Any occurrence of *1/4* that is either preceded by *under* or followed by *of* should be tagged as a fraction.

```
fst[0]: regex {1/4} EndTag(Fraction) OR(LC({under }), RC({ of}));
```

Figure 17 shows the result of combining the two constraints into a single network. The `pmatch` test below indicates that the constraints have the desired effect.

```
fst[0]: regex {1/4} [EndTag(Date)     AND(NLC({under }), NRC({ of}))|
                     EndTag(Fraction)  OR( LC({under }),  RC({ of}))];
```

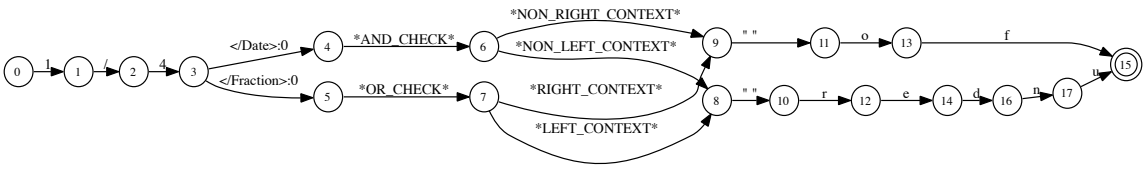

Figure 17: Date and fraction constraints for *1/4*

```
3.3 Kb. 18 states, 20 arcs, 4 paths.

fst[1]: pmatch under 1/4 voters said
6|3|1/4|<Fraction>

fst[1]: pmatch the bill is due on 1/4 next year
19|3|1/4|<Date>

fst[1]: pmatch  under 1/4 inch layer
6|3|1/4|<Fraction>

fst[1]: pmatch just 1/4 of the applications
5|3|1/4|<Fraction>
```

This example demonstrates that the FST can encode any context constraint on regular languages that is expressible in propositional logic. In a manner similar to De Morgan's Law, negation is pushed down to the `NLC` and `NRC` constraints: $\neg(p \vee q)$ is equivalent to $(\neg p \wedge \neg q)$.

Table 5 sums up the function symbols for context constraints. The `.r` symbol is the reverse operator needed for the left-context constraints.

RC(X)	Returns "*RIGHT_CONTEXT*" X
NRC(X)	Returns "*NON_RIGHT_CONTEXT*" X
LC(X)	Returns "*LEFT_CONTEXT*" X.r
NLC(X)	Returns "*NON_LEFT_CONTEXT*" X.r
AND(C1, C2)	Returns "*AND_CHECK*" [C1 | C2]
OR(C1, C2)	Returns "*OR_CHECK*" [C1 | C2]

Table 5: Context-constraint functions

Checking the left context of a potential match presupposes that `pmatch` retains in memory the input that it has already processed. How much that is depends on how far

back the longest left-context constraint has to look. For practical reasons FST puts a limit on the amount of memory devoted to remembering past history and on how far to the right from the the end of a potential match FST has to look to determine whether the match is valid. The default setting of `max-context-length` is 256. It can be reset from FST command line. If the limit is exceeded, FST reports the error and rejects the match.

2.3 Embedded pattern components

As the `pattern.script` illustrates, a pattern may refer to networks by insert symbols without physically encoding them in the pattern network. These networks may in turn contain insert symbols of their own. Returning to the example in Section 1.1.1, we can declare a top-level pattern that explicitly refers only to the `NP` network.

```
define D {the} | {this} | {a} | {an};
define N {city} | {girl} | {side} | {ocean};
define P {with} | {of} | {from} | {on};
define PP Ins(P) " " Ins(NP);
define NP Ins(D) " " Ins(N)  (" " Ins(PP)) ;

regex Ins(NP) EndTag(NP);

fst[1]: pmatch she is a girl
she is <NP>a girl</NP>

fst[1]: pmatch she is a girl from a city
she is <NP>a girl from a city</NP>

fst[1]: pmatch she is a girl from a city on this side of the ocean
she is <NP>a girl from a city on this side of the ocean</NP>
```

In the above example only the top level network has an end tag. If we instead append the `EndTag(NP)` label to the definition of `NP` itself, `pmatch` marks the beginning and end of each nested `NP` string.

```
define NP Ins(D) " " Ins(N)  (" " Ins(PP)) EndTag(NP);

push NP

fst2]: pmatch a girl
<NP>a girl</NP>

fst[2]: pmatch a girl from the city
```

```
<NP>a girl from <NP>the city</NP></NP>

fst[2]: pmatch a girl from the city on the ocean
<NP>a girl from <NP>the city on <NP>the ocean</NP></NP></NP>
```

If all the components of the NP definition had end tags, `pmatch` would produce a full parse tree:

```
fst[1]: pmatch a girl from the city
<NP><D>a</D> <N>girl</N> <PP><P>from</P> <NP><D>the</D> <N>city</N></NP></PP></NP>
```

When `pmatch` routine enters a network referenced by an insert symbol, it takes note of its starting location. If the embedded network ends with an end tag, `pmatch` wraps a pair of XML tags around the entire string. Only the top-level pattern can be constrained by the context.

3 Conclusion

Although it is possible to use FST for context-free parsing, it is most likely less efficient in that role than a chart parser would be. FST is best applied to matching patterns that are regular languages. Compared to other similar tools, the main asset of FST is that it can compile any number of patterns into a single network, optimize it for speed or size, and apply all the patterns in parallel in a single run. The networks can be created with the help of a regular expression calculus that is more expressive than the regular expression formalism found in tools such as `perl` or `python`.

References

[1] Kenneth R. Beesley and Lauri Karttunen. *Finite State Morphology.* CSLI Publications, Palo Alto, CA, 2003.

[2] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1963.

[3] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

Acknowledgements

Thanks to Kenneth R. Beesley, Ronald M. Kaplan and John T. Maxwell III for improvements of this document. Special thanks to Paul Meurer for helping to debug `pmatch`.

Appendix

Plural script

```
# This script defines the function Plural(X) that maps an English word
# to its plural form. Irregular singular:plural pairs are listed directly
# Regular plurals are produced by the cascade of five replace rules.
# The irregular and regular forms are combined by the priority-union
# operator, .P., that chooses an irregular form if one exists. If no
# irregular form exists for a particular word, a regular plural is
# produced.

define Irregular [{child}:{children} | {man}:{men} | {woman}:{women} |
                  {ox}:{oxen} | {bacterium}:{bacteria} |
                  {corpus}:{corpora} | {criterium}:{criteria} |
                  {curriculum}:{curricula} | {datum}:{data} |
                  {genus}:{genera} | {medium}:{media} |
                  {memorandum}:{memoranda} | {stratum}:{strata} |
                  {phenomenon}:{phenomena} | {deer} | {fish} |
                  {offspring} | {sheep} | {foot}:{feet} |
                  {goose}:{geese} | {tooth}:{teeth} |
                  {louse}:{lice} | {mouse}:{mice} |
                  {crisis}:{crises} | {thesis}:{theses} |
                  {oasis}:{oases} | {hypothesis}:{hypotheses}];

define AddE f | s | {sh} | {ch} | x | z |
            {acco}|{alvo}|{ando}|{ango}|{anjo}|{argo}|{asco}|{asso}|
            {atto}|{cano}|{cebo}|{digo}|{echo}|{ecko}|{egro}|{endo}|
            {ento}|{esco}|{esto}|{etto}|{falo}|{fico}|{hako}|{halo}|
            {hero}|{hobo}|{ildo}|{illo}|{ingo}|{into}|{irdo}|{lago}|
            {lico}|{mago}|{mato}|{mino}|{nado}|{nkgo}|{otto}|{pedo}|
            {rado}|{rago}|{tato}|{tico}|{tigo}|{ucco}|{uito}|{utgo}|
            {veto}|{viso}|{zebo};

define Vowel a | e | i | o | u ;

define FtoV {cal}|{dwar}|{el}|{hal}|{hoo}|{kni}|{lea}|{li}|{loa}|{scar}|
            {sel}|{shea}|{shel}|{thie}|{whar}|{wi}|{wol};

define Regular [..] -> %.    || _ .#.                .o.
                 %. -> {es}  || AddE _ .#.           .o. # buses, heroes
               y %. -> {ies} || \Vowel  _ .#.        .o. # ladies
                 %. -> s     || _ .#.                .o. # days, pianos
                 f  -> v     || .#. FtoV _ {es} .#. ;    # wolves

define Plural(X) [[X .o. Irregular] .P. [X .o. Regular]];
```

Numbers script

```
# This script constructs a transducer that maps the English numerals
# "one", "two", "three", ..., "ninety-nine", to the corresponding
# numbers "1", "2", "3", ... "99".

define OneToNine [1:{one} | 2:{two} | 3:{three} | 4:{four}  | 5:{five} |
                  6:{six} | 7:{seven} | 8:{eight} | 9:{nine}];

define TeenTen [3:{thir} | 5:{fif} | 6:{six}|
                7:{seven} | 8:{eigh} | 9:{nine}];

define Teens [1:0 [{0}:{ten} | 1:{eleven} | 2:{twelve}|
                  [TeenTen | 4:{four}] 0:{teen}]];

define TenStem [2:{twen} | TeenTen | 4:{for}];

# TenStem is followed either by "ty" paired with a zero
# or by "ty-" mapped to an epsilon and folowed by a one
# number. Note that {0} means zero and not epsilon

define Tens [TenStem [{0}:{ty} | 0:{ty-} OneToNine]];

define OneToNinetyNine [ OneToNine | Teens | Tens ];

# Let's put the result on the stack for testing.

echo
echo push OneToNinetyNine
push OneToNinetyNine
echo
# echo print random-upper
# print random-upper 5
# echo
# echo print random-lower
# print random-lower 5
# echo
echo apply down 75
down 75
echo
echo apply up 13
up thirteen
echo
```

Pattern script

```
# This script compiles lists of actors, movies, and dictators from
# text files, compiles two regular expressions from files and unions
# them into a single pattern matching network. Calling OptCap(X, L)
# allows and corrects missing initial capitalization. The composition
# of the Pattern with the rule inserts an optional newline character
# after a dash on the input side. This allows a line break after a
# dash, that is, an optional 0:"\n" label in the pattern.

set verbose off

define Actors OptCap(@txt "Actor.txt", L);

define Movies OptCap(@txt "Movie.txt", L);

define Dictators OptCap(@txt "Dictator.txt", L);

define Pattern Ins(Actors) EndTag(Actor) |
               Ins(Dictators) EndTag(Dictator) |
               Ins(Movies) EndTag(Movie) |
               @re "PhoneNum.re" EndTag(PhoneNum) |
               @re "SocSecNum.re" EndTag(SocSecNum) ;

regex Pattern .o. [..] (->) "\n" || "-" _ ;

save patterns.fst

set locate-patterns on

pmatch < input.txt
```