Applications of Finite-State Transducers in Natural Language Processing

Lauri Karttunen

Xerox Research Centre Europe,
6, chemin de Maupertuis, F-38240 Meylan, France
karttunen@xrce.xerox.com http://www.xrce.xerox.com

Abstract. This paper is a review of some of the major applications of finite-state transducers in Natural Language Processing ranging from morphological analysis to finite-state parsing. The analysis and generation of inflected word forms can be performed efficiently by means of lexical transducers. Such transducers can be compiled using an extended regular expression calculus with restriction and replacement operators. These operators facilitate the description of complex linguistic phenomena involving morphological alternations and syntactic patterns. Because regular languages and relations can be encoded as finite-automata, new languages and relations can be derived from them directly by the finite-state calculus. This is a fundamental advantage over higher-level linguistic formalisms.

1 Introduction

The last decade has seen a substantial surge in the use of finite-state methods in many areas of natural language processing. This is a remarkable comeback considering that in the dawn of modern linguistics, finite-state grammars were dismissed as fundamentally inadequate. Noam Chomsky's seminal 1957 work, *Syntactic Structures* [3], includes a short chapter devoted to "finite state Markov processes", devices that we now would call *weighted finite-state automata*. In this section Chomsky demonstrates in a few paragraphs that

> English is not a finite state language. (p. 21)

In any natural language, a sentence may contain discontinuous constituents embedded in the middle of another discontinuous pair as in "If$_1$ ... either$_2$ ... or$_2$ ... then$_1$ ..." It is impossible to construct a finite automaton that keeps track of an unlimited number of such nested dependencies. Any finite-state machine for English will accept strings that are not well-formed.

The persuasiveness of *Syntactic Structures* had the effect that, for many decades to come, computational linguists directed their efforts towards more powerful formalisms. Finite-state automata as well as statistical approaches disappeared from the scene for a long time. Today the situation has changed in a fundamental way: statistical language models are back and so are finite-state

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 34–46, 2001.

automata, in particular, finite-state transducers. One reason is that there is a certain disillusionment with high-level grammar formalisms. Writing large-scale grammars even for well-studied languages such as English turned out to be a very hard task. With easy access to text in electronic form, the lack of robustness and poor coverage became frustrating. But there are other, more positive reasons for reasons for the renewed interest in finite-state techniques. In phonology, it was discovered rather early [6] that the kind formal descriptions of phonological alternations used by linguists were, against all appearances, finite-state models. In syntax, it became evident that although English as a whole is not a finite-state language, there are nevertheless subsets of English for which a finite-state description is not only adequate but also easier to construct than an equivalent phrase-structure grammar. Finally, considerable process has been made in developing special finite-state formalisms that are suited for the description of linguistic phenomena and, along with them, compilers that efficiently produce automata from such a description. The automata in current linguistic applications are typically much too large and complex to be produced by hand.

The following sections will cover these positive developments in more detail.

2 Finite-State Morphology

Morphology is a domain of linguistics that studies the formation of words. It is traditional to distinguish between *surface forms* and their analyses, called *lemmas*. The lemma for a surface form such as the English word `bigger` typically consists of the traditional dictionary citation form of the word together with terms that convey the morphological properties of the particular form. For example, the lemma for `bigger` might be represented as `big+Adj+Comp` to indicate that `bigger` is the comparative form of the adjective `big`.

There are two challenges in modeling natural-language morphology:

1. **Morphotactics**
   Words are typically composed of smaller units of meaning, called morphemes. The morphemes that make up a word must be combined in a certain order: `piti-less-ness` is a word of English but `*piti-ness-less` is not. Most languages build words by concatenation but some languages also exhibit non-concatenative processes such as interdigitation and reduplication [2].
2. **Morphological Alternations**
   The shape of a morpheme often depends on the environment: `pity` is realized as `piti` in the context of less, `die` as `dy` in `dying`.

The basic claim of finite-state approach to morphology is the relation between the surface forms of a language and their corresponding lemmas can be described can be modeled as a *regular relation*.[1] If the relation is regular, it can be defined using the metalanguage of regular expressions; and, with a suitable compiler, the regular expression source code can be compiled into a finite-state transducer that implements the relation computationally.

[1] Some writers prefer the term *rational relation*.

In the resulting transducer, each path (= sequence of states and arcs) from the initial to a final state represents a mapping between a surface form and its lemma, also known as the *lexical form*. For example, the information that the comparative of the adjective `big` is `bigger` might be represented in the English lexical transducer by the path in Figure 1 where the zeros represent epsilon symbols.[2]

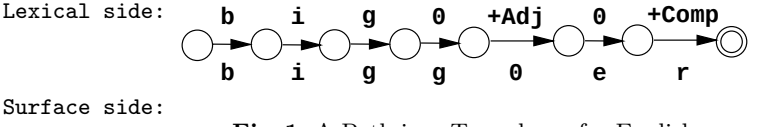


Fig. 1. A Path in a Transducer for English

For the sake of clarity, Figure 1 represents the upper and the lower side of the arc label separately on the opposite sides of the arc. In the rest of the paper, we use a more compact notation: the upper and the lower symbol are combined into a single label of the form `upper:lower` if the symbols are distinct. Identity pairs, e.g. `b:b`, are reduced to a single symbol. In standard notation, the path in Figure 1 is labeled as

`b i g 0:g +Adj:0 0:e +Comp:r`.

An important characteristic of the finite-state transducers built at Xerox is that they are inherently bidirectional: there is no privileged input side. The path in Figure 1 can be traversed matching either the form `bigger` to produce `big+Adj+Comp`, or vice versa. The same transducer can be used for analysis (surface input, "upward" application) or for generation (lexical input, "downward" application).

A single surface string can be related to multiple lexical strings. For example, a morphological transducer for French applied upward to the surface string `suis` may produce the four lexical strings shown in Figure 2. Ambiguity in the downward direction is also possible, as in the relation of the lexical string `payer+IndP+SG+P1+Verb` ("I pay") to the surface strings `paie` and `paye`, which are in fact alternate spellings in standard French orthography.

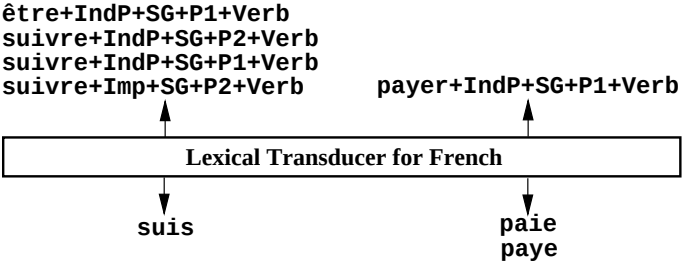


Fig. 2. Morphological Ambiguities

[2] The epsilon symbols and their placement in the string is not significant. We will ignore them whenever it is convenient.

At Xerox, such lexical transducers have been created for a great number of languages including most of the European languages, Turkish, Arabic, Korean, and Japanese. The source descriptions are written using notations [12,9, 1] that are helpful shorthands for ordinary regular expressions. The construction is commonly done by creating two separate modules: a lexicon description that defines the morphotactics of the language and a set of rules that define regular alternations such as the gemination of `g` and the epenthetical `e` in the surface form `bigger`. Irregular alternations such as `être:suis` are defined explicitly in the source lexicon. The separately compiled lexicon and rule networks are subsequently composed together as in Figure 3.

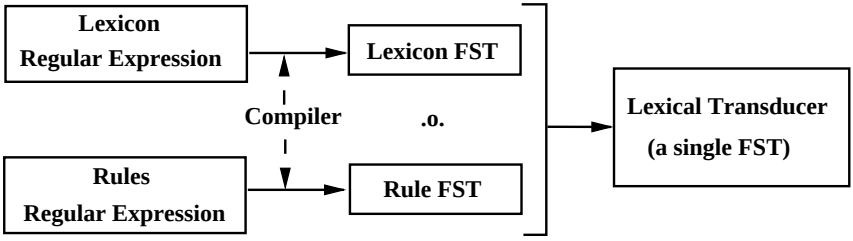


Fig. 3. Creation of a Lexical Transducer

Lexical transducers may contain hundreds of thousands, even millions, of states and arcs and an infinite number of paths in the case of languages such as German that in principle allow noun compounds of any length. The regular expressions from which such complex networks are compiled include high-level operators that have been developed in order to make it possible to describe constraints and alternations that commonly found in natural languages in a convenient and perspicuous way. We will describe them in the following sections.

3 Basic Expression Calculus

The notation used in this section comes from the Xerox finite-state calculus. It is described in detail in Chapter 2 of the forthcoming book by Beesley and Karttunen [1]. We use uppercase letters, `A`, `B`, etc., as variables over regular expressions. Lower case letters, `a`, `b`, etc., stand for symbols. There are two special symbols: **0**, the EPSILON symbol, that stands for the empty string and **?**, the ANY symbol that represents the infinite set of symbols in some yet unknown alphabet. The special meaning of `0`, `?`, and any other symbol can be canceled by enclosing the symbol in double quotes.

An atomic expression consisting of a symbol pair such as `a:b` denotes a relation containing the corresponding strings. An expression consisting of a single symbol such as `a` denotes the language consisting of "a" or, alternatively, the corresponding identity relation. The Xerox implementation intentionally does not distinguish between `a` and `a:a`.

Complex regular expressions can be built up from simpler ones by means of regular expression operators. Square brackets, `[]`, are used for grouping expressions. Because both regular languages and regular relations are closed under concatenation and union, the following basic operators can be combined with any kind of regular expression:

`A | B`	Union.
`A B`	Concatenation.
`(A)`	Optionality; equivalent to `[A | 0]`.
`A+`	Iteration; one or more concatenations of `A`.
`A*`	Kleene star; equivalent to `(A+)`.

Although regular languages are closed under complementation and intersection, regular relations are not [8]; thus the following operators can be combined only with expressions that denote a regular language.

`~A`	Complement
`\A`	Term complement; all single symbol strings not in `A`.
`A & B`	Intersection
`A - B`	Subtraction (minus)

Regular relations can be constructed by means of two basic operators:

`A .x. B`	Crossproduct
`A .o. B`	Composition

The crossproduct operator, `.x.`, is used only with expressions that denote a regular language; it constructs a relation between them. `[A .x. B]` designates the relation that maps every string of `A` to every string of `B`.

4 Containment, Restriction, Replacement, and Marking

The syntax (though not the descriptive power) of regular expressions can be extended by defining new operators that allow commonly used constructions to be expressed more concisely. A simple example of a trivial but convenient extension is the CONTAINMENT operator `$`.

`$A` $=_{def}$ `[?* A ?*]`

For example, `$[a | b]` denotes all strings that contain at least one "a" or "b" somewhere.

The addition of new operators can be more than just a notational convenience. A case in point is Koskenniemi's [16] RESTRICTION operator `=>`.

`A => L _ R`	Restriction; `A` only in the context of `L _ R`.

Here A, L and R may denote any regular language. This expression designates the language of strings that have the property that any string of A that occurs in them is immediately preceded by some string from L and immediately followed by some string from R. For example, `a => b _ c` includes all strings that contain no occurrence of "a", strings like "bac-bac" that completely satisfy the condition, but no strings like "ab". A special boundary symbol, `.#.`, is used to indicate the beginning or the end of the string. For example, `a => _ .#.` allows "a" only at the end of a string.

The advantage of the restriction operator is that it encodes in a compact way a useful condition that is difficult to express in terms of the more primitive operators. The definition of `[A => L _ R]` is shown below.

`A => L _ R` $=_{def}$ `[~[ [~[?* L] A ?*] | [?* A ~[R ?*]] ]]`

Another example of a useful high-level abstraction is the REPLACE operator `->`. As we will see shortly, there are many constructions involving this operator. The simplest variant is unconstrained, obligatory replacement:

`A -> B` Replacement of A by B.

Transducers compiled from `->` expressions are usually intended to be applied downward; they can of course be inverted and applied in the other direction. The component expressions, A and B, must denote regular languages but the expression as a whole denotes a relation. The `[A -> B]` relation maps any upper-language string to itself if the string contains no instance of A. Upper language strings that contain instances of A are paired with lower-language strings that are otherwise identical except that each A segment is replaced by some B string. The definition [10] of simple replacement is shown below.

`A -> B` $=_{def}$ `[ [~$[A - 0] [A .x. B]]* ~$[A - 0]]`

Two replace expressions linked with a comma indicate parallel replacement. For example,

```
a -> b, b -> a
```

yields a transducer that exchanges the two letters mapping "abba" to "baab".

High-level abstractions like `A => L _ R` and `A -> B` are conceptually easier to operate with than the logically equivalent but very complex primitive formulas, just as it is easier to write complex computer programs in a high-level language rather than in a logically equivalent assembly language.

Instead of replacing the strings of a language by other strings, it is sometimes useful just to mark them in some special way. In the Xerox calculus, an expression of the form

`A -> B ... C` Marking A by B and C.

yields a transducer that maps any upper language string to a lower-language string that is identical to it except that any instance of `A` is preceded by a string from `B` and followed by a string from `C`. Here `A`, `B` and `C` may denote any regular language. In practice, however, `B` and `C` are usually atomic expressions. For example, `a | e | i | o | u -> "[" ... "]"` yields a transducer that encloses vowels between square brackets leaving the rest of the text unchanged. The relation includes pairs such as

```
 i c e c r e  a m
[i]c[e]c r[e][a]m
```

4.1 Constraining Replacement and Marking

Replacement and marking can be constrained in many different ways: by context, by direction of application, by length of the pattern to be replaced or marked. The basic technique for compiling constrained replacement and marking transducers was invented in the early 1980's by Kaplan and Kay [7] for Chomsky-Halle-type rewrite rules [4]. It was also used very early for Koskenniemi's two-level rules [16,14,12]. The idea was finally explained in detail in Kaplan and Kay's 1994 paper [8]. There is now a rich literature on this topic [10,17,5,11,15,18]. The details vary but the basic method involves composing together a cascade of networks that introduce various auxiliary symbols into the input string, constrain their distribution, and finally eliminate the auxiliary alphabet. As there is no space to explore the compilation issue in a technical way, we will only explain the syntax of constrained replacement and marking expressions and give a few examples of the corresponding transducers without explaining how the expressions are compiled.

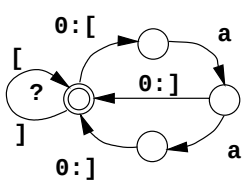
The transducers compiled from the simple replacement and marking expressions are in general ambiguous in the sense that a string in the upper language of the relation is paired with more than one lower-language string. For example, `a | a a -> "[" ... "]"` yields a marking transducer than maps the upper language string "aaa" into three different lower-language strings:

```
 a  a  a      a  a a      a a  a
 -  -  -      -  ---      ---  -
[a][a][a]    [a][a a]    [a a][a]
```

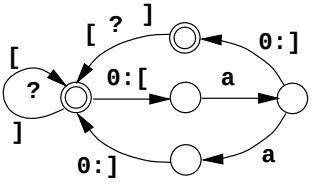
The `->` operator does not constrain the selection of the alternate substrings for replacement or marking. In this case, the upper language string can be factored or parsed in three different ways.

For many applications, it is useful to define another version of replacement and marking that in all such cases yields a unique outcome. The longest-match, left-to-right replace operator, `@->`, defined in [11], imposes a unique factorization on every input. The upper language substrings to be marked or replaced are selected from left to right, not allowing any overlaps. If there are alternate candidate strings starting at the same location, only the longest one is chosen. Thus `a | a a @-> "[" ... "]"` denotes a relation that unambiguously maps

"aaa" to "[aa][a]". The transducers corresponding to the `->` and `@->` variant of this expressions are shown in Figure 4.[3]




```
a | a a -> "[" ... "]"
```




```
a | a a @-> "[" ... "]"
```

Fig. 4. An Ambiguous and an Unambiguous Marking Transducer

Replacement and marking contexts can be specified using same notation as for restriction: `L _ R`, where `L` is the left context, `R` is the right context, and `_` marks the site of the upper language string that is replaced or marked. In the case of a restriction expression, the interpretation of context is self-evident because a restriction denotes a set of strings. This is not the case for replacement and marking. Replacement and marking expressions must specify whether `L` and `R` pertain to the upper or the lower side of the relation. The Xerox calculus provides specific markers `||`, `//`, \\ and \/ to distinguish between the four possible cases:

`|| L _ R`	L and R both on the upper side
`// L _ R`	L on the lower, R on the upper side
`\\ L _ R`	L on the upper, R on the lower side
`\/ L _ R`	L and R both on the lower side

To see the difference between, say `||` and `//`, versions let us consider two variants of a phonological rule that shortens a double "aa" in the context of another double "aa" in the preceding syllable. Here `C` represents any consonant.

Rule 1.	`a a -> a || a a C+ _`	(Slovak)
Rule 2.	`a a -> a // a a C+ _`	(Gidabal)

Vowel shortening is a very common type of morphological alternation under many different kinds of context conditions. Interestingly, in some languages such as Slovak the shortening depends on the lexical (upper side) context whereas in languages such as Gidabal (an Australian language), it is conditioned by the surface side.[4] The hypothetical lexical form "baacaadaafaa" would be realized quite differently in these two languages:

```
b a a c a a d a a f a a          b a a c a a d a a f a a
b a a c a   d a   f a            b a a c a   d a a f a
        Rule 1                           Rule 2
```

[3] The symbol `?` in an arc label represents an UNKNOWN symbol; in this case, any symbol other than `[`, `]`, and `a`. By convention, the leftmost state is the start state, final states are indicated by double circles.

[4] This example is due to Martin Kay (p.c.).

In a language like Slovak, the last three syllables would all shorten yielding "baacadafa" whereas a language like Gidabal would show the alternating pattern "baacadaafa".

The two replacement transducers compiled from Rule 1 and Rule 2 are shown in Figure 5.

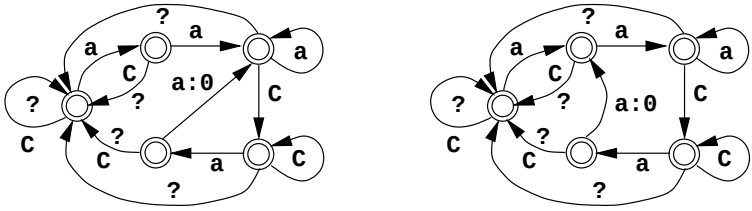


Fig. 5. Two Vowel-Shortening Rules

Contextual constraints may be combined with the directional left-to-right and longest match constraints. For example, if `C` and `V` stand for consonants and vowels, respectively, a simple syllabification rule may be expressed in the following way:

```
C* V+ C* @-> ... "-" || _ C V
```

This marking expression yields an unambiguous transducer that inserts a hyphen after each longest available instance of the `C* V+ C*` pattern that is followed by a consonant and vowel. The relation it encodes consists of pairs of strings such as

```
s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i              .
```

In this case, the choice between `||` and `//` makes no difference but the two other context markers, `\\` and `\/` could not be used here.

The syllabification transducer is a simple finite-state parser: it recognizes and marks instances of a regular language in a text. In the next section we will show a more sophisticated example of this kind.

5 Finite-State Syntax

Although the syntax of a natural language cannot in general be described by a finite-state, or even a context-free grammar there are many subsets of natural language that can be correctly described by very simple means, for example, names and titles, addresses, prices, dates, etc. In this section, we examine one such case in detail: a grammar for dates.

For the sake of illustration, let us consider here only one of several common date formats, expressions such as

```
                    Tuesday
July 25             Tuesday, July 25
July 25, 2000       Tuesday, July 25, 2000
```

In the following we assume that a date expression consists of a day of the week, a month and a date with or without a year, or a combination of the two. Note that this description of the syntax of date expressions presents the same problem we encountered in the `a | aa @-> a` example in the previous section. Long date expressions, such as "Tuesday, July 25, 200", contain smaller well-formed date expressions, e.g. "July 25", that should be ignored in the context of a larger date. In order to simplify the presentation, we stipulate that date expressions are contiguous strings, including the internal spaces and commas.

To facilitate the specification of the date language we first define some auxiliary terms and then use them to define a language of dates and a parser for the language. The complete set of definitions is shown below:

```
1To9  =  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
0To9  =  "0" | 1To9
Day   =  Monday | Tuesday | ..... | Saturday | Sunday
Month =  January | February | ..... | November | December
Date  =  1To9 | [1 | 2] 0To9 | 3 ["0" | 1]
Year  =  1To9 (0To9 (0To9 (0To9)))
AllDates = Day | (Day ", ") Month " " Date (", " Year)
```

From these definitions we can compile a small finite-state automaton, `AllDates`, with 13 states and 96 arcs that describes a language of about 30 million date expressions for the period from January 1, 1 to December 31, 9999.

A parser for the language can be compiled from the following simple regular expression.

```
AllDates @-> "[" ... "]"
```

It yields a transducer of 23 states and 321 arcs that marks maximal date expressions in the manner illustrated by the following text:

```
Today is [Tuesday, July 25, 2000] because yesterday was [Monday]
and it was [July 24] so tomorrow must be [Wednesday, July 26].
```

Because of the left-to-right, longest-match constraints associated with the `@->` operator, the transducer brackets only the maximal date expressions.

However, this regular-expression grammar is not optimal. The `AllDates` language includes a large number of syntactically correct but semantically invalid date expressions. For example, there is no "April 31, 2000", "February 29, 1900", or "Sunday, September 29, 1941". April only as 30 days in any year; unlike year 2000, year 1900 was not a leap year; and September 29, 1941 fell on a Monday.

All these three types of imperfections can be corrected within the finite-state calculus. For each of these three types on invalid dates we can define a regular language that excludes such expressions. By intersecting these constraint languages with the `AllDates` language, we can define a language that contains only semantically valid dates. Figure 6 illustrates the idea.

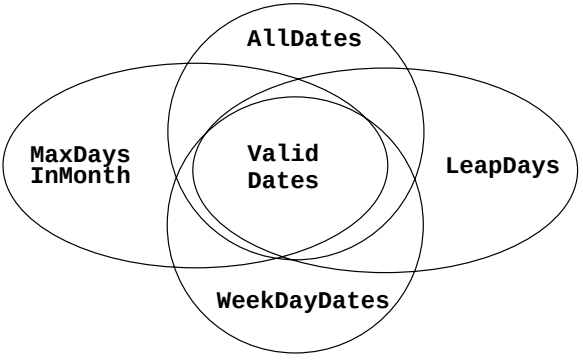


Fig. 6. Refinement by Intersection

We need three constraints:

`MaxDaysInMonth`	Restriction on the distribution of 30 and 31.
`LeapDays`	Restriction on February 29.
`WeakDayDate`	Restrictions on weekdays and dates

In fact, all the constraints can be expressed by means of the restriction operator `=>` defined in the previous section. For example, to build the leap day constraint we first need to define the language of leap years, that is the language of all numbers divisible by four but subtracting centuries such as 1900 that are not divisible by 400.

```
Even =      "0" | 2 | 4 | 6 | 8
Odd =       1 | 3 | 5 | 7 | 9
N =         1To9 0To9*
Div4 =      [(N) Even) ["0" | 4 | 8]] | [(N) Odd [2 | 6]]
LeapYears = Div4 - [[N - Div4] "0" "0"]
```

Here we first define `Div4` as the infinite set of natural numbers that are divisible by four. This set consists of two parts: numbers that end in 0, 4, or 8 possibly preceded by an even number and numbers that end in 2 or 6 preceded by an odd number. Finally, we define `LeapYears` as the set of numbers divisible by 4 subtracting centuries that are not multiples of 400. Note that the expression `[N - Div4] "0" "0"` denotes numbers with two final zeros that are preceded by a number that is not divisible by four. For example, it includes "1900" but not "2000". Because `LeapYears` is defined as `Div4` minus this set, it follows that the string "2000" is in the language but "1900" is not.

Once the language of leap years is defined, the distribution of "February 29" in date expressions can be constrained with the following simple restriction.

```
LeapDays = February " " 2 9 ", "  => _ LeapYears .#.
```

In other words: a date expression containing "February 29, " must terminate with a leap year. The boundary symbol, `.#.`, is necessary here to mark the end of the year string in order to rule out expressions like "February 29, 1969" which

would qualify if we were allowed to take into account only the first three digits since year 196 is a leap year in the Gregorian calendar.

The construction of the `WeakDayDate` constraint is not as trivial but not as difficult as it might initially seem. See [13] for details. Having constructed the auxiliary constraint languages we can define the language of valid dates as

```
ValidDates = AllDates & MaxDaysInMonth & LeapDays & WeekDayDates
```

The network contains 805 states, 6472 arcs, and about 7 million date expressions.

We could now construct a parser that recognizes only valid dates. But we actually can do something more interesting, namely, define a parser that recognizes all date expressions and marks them as valid, "[VD", or invalid, "[ID":

```
              ValidDates] @-> "[VD" ... "]" ,
[AllDates - ValidDates] @-> "[ID" ... "]"
```

This parallel replacement expression compiles into a 2699 state, 20439 arc transducer in about 15 seconds on a Sun workstation. The time includes the compilation of all the auxiliary expressions and constraints discussed above. The following example illustrates the effect of the transducer on a sample text.

```
The correct date for today is [VD Tuesday, July 25, 2000].
Today is not [ID Tuesday, July 26, 2000].
```

6 Conclusion

Although regular expressions and the algorithms for converting them into finite-state automata have been part of elementary computer science for decades, the restriction, replacement, and marking expressions we have focused on are relatively recent. They have turned out to be very useful for linguistic applications in particular for morphology, tokenization, and shallow parsing. Descriptions consisting of regular expressions can be efficiently compiled into finite-state networks, which in turn can be determinized, minimized, sequentialized, compressed, and optimized in other ways to reduce the size of the network or to increase the application speed. Many years of engineering effort have produced efficient runtime algorithms for applying networks to strings.

Regular expressions have a clean, declarative semantics. At the same time they constitute a kind of high-level programming language for manipulating strings, languages, and relations. Although regular grammars can cover only limited subsets of a natural language, there can be an important practical advantage in describing such sublanguages by means of regular expressions rather than by some more powerful formalism. Because regular languages and relations can be encoded as finite automata, they can be more easily manipulated than context-free and more complex languages. With regular expression operators, new regular languages and relations can be derived directly without rewriting the grammars for the sets that are being modified. This is a fundamental advantage over higher-level formalisms.

References

1. Kenneth R. Beesley and Lauri Karttunen. *Finite-State Morphology: Xerox Tools and Techniques*. Cambridge University Press, 2000. To appear.
2. Kenneth R. Beesley and Lauri Karttunen. Finite-state non-concatenative morphotactics. In Lauri Karttunen Jason Eisner and Alain Thériault, editors, *SIGPHON-2000*, pages 1–12, August 6 2000. Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology.
3. N. Chomsky. *Syntactic Structures*. Mouton, Gravenhage, Netherlands, 1957.
4. Noam Chomsky and Morris Halle. *The Sound Pattern of English*. Harper and Row, New York, 1968.
5. Edmund Grimley-Evans, George Anton Kiraz, and Stephen G. Pulman. Compiling a partition-based two-level formalism. In *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, 1996.
6. C. Douglas Johnson. *Formal Aspects of Phonological Description*. Mouton, The Hague, 1972.
7. Ronald M. Kaplan and Martin Kay. Phonological rules and finite-state transducers. In *Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting*, New York, December 27-30 1981. Abstract.
8. Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.
9. Lauri Karttunen. Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, CA, April 1993.
10. Lauri Karttunen. The replace operator. In *ACL'95*, Cambridge, MA, 1995. cmp-lg/9504032.
11. Lauri Karttunen. Directed replacement. In *ACL'96*, Santa Cruz, CA, 1996. cmp-lg/9606029.
12. Lauri Karttunen and Kenneth R. Beesley. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA, October 1992.
13. Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Journal of Natural Language Engineering*, 2(4):305–328, 1996.
14. Lauri Karttunen, Kimmo Koskenniemi, and Ronald M. Kaplan. A compiler for two-level phonological rules. Technical report, Xerox Palo Alto Research Center and Center for the Study of Language and Information, Stanford University, June 25 1987.
15. André Kempe and Lauri Karttunen. Parallel replacement in finite-state calculus. In *COLING'96*, Copenhagen, August 5–9 1996. cmp-lg/9607007.
16. Kimmo Koskenniemi. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki, 1983.
17. Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *ACL'96*, Santa Cruz, CA, 1996.
18. Gertjan van Noord and Dale Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam Germany, 1999.