Natural Language Engineering 1–24 © 1997 Cambridge University Press

Regular Expressions for Language Engineering

L. KARTTUNEN, J-P. CHANOD, G. GREFENSTETTE, A. SCHILLER

Rank Xerox Research Centre (RXRC), 6 Chemin de Maupertuis, 38240 Meylan, France

{Lauri.Karttunen,Jean-Pierre.Chanod,Gregory.Grefenstette,Anne.Schiller}
@grenoble.rxrc.xerox.com

(*Received 5 February1997*)

Abstract

Many of the processing steps in natural language engineering can be performed using finite state transducers. An optimal way to create such transducers is to compile them from regular expressions. This paper is an introduction to the regular expression calculus, extended with certain operators that have proved very useful in natural language applications ranging from tokenization to light parsing. The examples in the paper illustrate in concrete detail some of these applications.

1 Introduction

The use of finite state transducers for morphological analysis and generation (Karttunen *et al.* 1992; Karttunen 1994; Beesley and Karttunen 1997) is by now well established. It is less well known how far the techniques of finite state transformation of text can succeed in solving other natural language engineering problems beyond morphology. In this paper, we will present a number of applications of finite state technology to other language engineering problems, and describe part of the regular expression calculus that we have developed that make these applications possible. Although large efforts have been made to build local grammars (Silberztein 1993) without the help of a finite state calculus, the expressive power of a well-designed calculus makes it possible to create modular rule sets and lexical descriptions that are easy to update and maintain, accelerating the production of diverse engineering applications.

Our paper is structured in the following way: After a general introduction to the finite state calculus, special attention will be given to three finite state operators: restriction, replacement, and left to right, longest match replacement. These conceptually simple regular expression operators express in a concise way conditions whose explicit formulation would otherwise be complicated and unwieldy. Their introduction into the Xerox finite state calculus has spurred a number of language engineering applications, some of which are described below. A graded set of examples using these operators will be presented as a concrete illustration of their power and simplicity.

After this overview of the Xerox finite state calculus, we will present a number of

applications using this calculus over textual structures larger than individual words. In Section 4.1, we present tokenization applications. Subsection 4.1.1 presents a deterministic tokenizer used in our part of speech tagging suite. Nondeterministic tokenization, presented in subsection 4.1.3, maintains alternative tokenizations of the same input stream in a finite state structure to which further linguistic treatments, e.g. parsing, can be applied.

Subsection 4.2 presents two applications of light parsing over tagged text. Just as tokenization applies finite state rules and constraints over strings involving more than one word, these applications apply to units, here sentences, recognized by the part of speech tagger, which include more than one word. And just as transducers compiled from morphological rules insert or delete characters in words, these light parsers introduce syntactic markings within a tagged sentence, around sequences that fulfill the conditions specified in the finite state rules.

2 Introduction to the Finite-State Calculus

Finite state transducers that encode phonological and morphographical alternations are generally not created directly but are compiled from rules of some sort. The two-level rule formalism (Koskenniemi 1983) as well as classical phonological rewrite rules can be viewed as special kinds of regular expressions (Kaplan and Kay 1994) that extend the basic language with new operators and new types of expressions. These extensions make the formalism more suitable for particular applications but they do not affect its generative power. In this section we will discuss recent extensions to the regular expression calculus that enable us to create finite state transducers for syntactic processing from high level rules.

We start with a brief review of the basic regular expression calculus and proceed with the discussion of two special operators that were originally introduced for phonological and morphographical alternations but which have since proved useful in writing syntactic rules as well. These are the restriction operator `=>` and the replace operator `->`. We will then describe in more detail a variant of the latter, the left to right, longest match replacement `@->` recently introduced in (Karttunen 1996), and illustrate its application to syntactic description.

2.1 Simple regular expressions

The language of regular expressions is a formal language similar to formulas of Boolean logic. It has a simple syntax but the expressions can be arbitrarily complex. Like formulas of Boolean logic, regular expressions denote sets. We need to distinguish two kinds of sets: sets of *strings* and sets consisting of *pairs of strings*. We use the term *language* to refer to a set of simple strings and the term *relation* in talking about sets of string pairs. The terms *regular language* and *regular relation* refer to sets than can be described by a regular expression.

Regular languages and relations may be encoded as *finite state networks*. Languages are represented by simple automata, relations by transducers. Any regular expression can be compiled into a network that represents the corresponding lan-

guage or relation.[1] Because of the close connection between regular expressions and finite state networks, we often use the terms *regular* and *finite state* interchangeably in this paper.

Regular expressions contains two kinds of symbols: *unary symbols* and *symbol pairs*. Unary symbols (`a`, `b`, etc) denote strings, symbol pairs (`a:b`, `a:0`, `0:b`, etc.) represent pairs of strings. The simplest kind of regular expression contains a single symbol. For example, `a` denotes the set {"a"}. Similarly, the regular expression `a:b` denotes the singleton relation {<"a", "b">}. A regular relation may always be viewed as a mapping between two regular languages. The `a:b` relation is simply the crossproduct of the languages denoted by the expressions `a` and `b`.

In order to distinguish the two languages that are involved in a regular relation, we can call the first one the *upper* and the second one the *lower* language of the relation. Correspondingly, in the pair `a:b`, the first symbol, `a`, can be called the *upper symbol* and the second one, `b`, the *lower symbol*. The two components of a symbol pair are separated in our notation by a colon, `:`, without any whitespace before or after. To make the notation less cumbersome, we systematically ignore the distinction between the language `A` and the identity relation that maps every string of `A` to itself. Therefore, we also write `a:a` simply as `a`.

A transducer that encodes a regular relation can be applied in two ways: to map a string in the upper language to the corresponding string(s) in the lower language, or vice versa. There is no privileged direction of application. In the examples in later sections, the intended input language is always on the upper side and the output on the lower side of the relation.

Two regular expression symbols have a special interpretation: `0` (epsilon) and `?` (any). The epsilon symbol `0` denotes the empty string. `?` stands for any symbol that occurs in the same regular expression and for any unknown symbol. In a later section we introduce one more special symbol, the boundary marker `.#.`, to refer to the beginning or the end of a string in certain expressions.

The special meaning of any symbol may be turned off by prefixing it with the escape character `%` or by enclosing the symbol in double quotes. `%0`, and `"0"` are interpreted as ordinary zero digits and not as the special character epsilon. On the other hand, certain other strings receive a special interpretation within a doubly quoted string. For example, `"\n"` is interpreted as the newline character, `"\t"` as a tab following the C programming language conventions. Because whitespace is generally ignored, a space as a regular symbol must be prefixed with `%` or enclosed in double quotes: `% ` , `" "`.

Complex regular expressions can be built up from simpler ones by means of regular expression operators. 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:

[1] See `http://www.rxrc.xerox.com/research/mltt/fst/` for a demonstration of the Xerox regular expression compiler and for a more comprehensive description of the syntax and semantics of regular expressions.

`A	B`	Union.
`A B`	Concatenation.
`(A)`	Optionality; union with the empty string.
`A+`	Iteration; one or more concatenations of `A`.
`A*`	Kleene star; equivalent to `(A+)`.

Square brackets, `[]`, are used for grouping expressions. Thus `[A]` is equivalent to `A` while `(A)` is not. Note the following simple expressions:

`[]`	The empty-string language/identity relation
`?*`	The universal language/identity relation.

Although regular languages are closed under complementation and intersection, regular relations are not (see Kaplan and Kay 1994), thus the following operators can be combined only with expressions that denote a regular language.

`~A`	Complement (negation)
`A & B`	Intersection
`A - B`	Relative complement (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`. If `A` contains x and `B` contains y, the pair $<x, y>$ is included in the crossproduct.

Composition is an operation on relations that yields a new relation. `[A .o. B]` maps strings that are in the upper language of `A` to strings that are in the lower language of `B`. If `A` contains the pair $<x,y>$ and `B` contains the pair $<y, z>$, the pair $<x, z>$ is in the composite relation.

2.2 Defining New Operators

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 (1983) restriction operator `=>`, originally introduced for two-level phonological rules.

`A => B _ C`	Restriction; `A` only in the context of `B _ C`.

Here `A`, `B` and `C` 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 `B` and immediately followed by some string from `C`. 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". Reductionist finite state parsers (Koskenniemi *et al.* 1992; Voutilainen and Tapanainen 1993; Chanod and Tapanainen 1997) make frequent use of such constraints to exclude unwanted analyses.

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 => B _ C]` is shown below.

`A => B _ C` $=_{def}$ `[~[ [~[?* B] A ?*] | [?* A ~[C ?*]] ]]`

Clearly, high level abstractions like `[A => B _ C]` 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.

Note that the definition of the restriction operator `=>` given above contains three negations. Because regular relations are not closed under complementation, all the component expressions in `[A => B _ C]` must denote regular languages rather than relations. In the general regular expression calculus we do not allow expressions such as `a:0 => b _ c` that are well-formed two-level rules in Koskenniemi's formalism. In our two-level calculus all pair symbols are treated as atomic symbols and converted to real symbol pairs only after the compilation is finished (Karttunen and Beesley 1992).

Another example of a useful high level abstraction is the replace operator, `->`, that plays much the same role in the general regular expression calculus as `=>` and `<=` in two-level rules. This simplest type of replacement is unconstrained by contexts:

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

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 string to itself if the string contains no instance of `A`. Strings that contain instances of `A` are paired with copies that are otherwise identical except that each `A` segment is replaced by some `B` string. The exact definition of `A -> B` is shown below.

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

This relatively simple idea would be also rather cumbersome to express without the explicit replace `->` operator. The same is true of the other types of replace expressions (Kaplan and Kay 1994; Karttunen 1995) that constrain the operation by left and right contexts:

`A -> B || L _ R` Replacement of `A` by `B` in the context `L _ R`.

The two vertical bars, `||` in the above contexted replacement indicate that both contexts `L` and `R` pertain to the upper language of the relation. (See Karttunen 1995; Kempe and Karttunen 1996 for other variations.)

Another way of generalizing the replace operator is to make two or more replacements in parallel:

```
A1 -> B1, A2 -> B2
```

This is defined like the single `[A -> B]` replacement above except that we replace `~$[A - []]` by `~$[[A1 | A2] - []]` and `[A .x. B]` by the union of multiple crossproducts: `[[A1 .x. B1] | [A2 .x. B2]]`.

2.3 Left to right, Longest match Replacement

The transducers compiled from the simple replacement expression, `A -> B` are in general ambiguous in the sense that a string in the upper language of the relation may become paired with more than one string. This can happen even if the `B` language consists of a single string. For example `[a b | b | b a | a b a] -> x` maps the string "aba" into four different strings, as shown below.

```
a b a      a b a      a b a      a b a
---          -          ---      -----
 x  a      a x a      a  x         x
```

The reason is that the simple replacement relation does not constrain the selection of the alternate substrings for replacement. Here we get different results for "aba" depending on whether the replacement starts at the beginning or in the middle of the string. At both sites there are two alternative replacements to be made. Starting at the beginning, we may relace either "ab" or "aba". Starting in the middle, we can replace either "b" or "ba". The underlining show the four alternate factorizations of the input string.

For many applications, it is useful to define another version of replacement that in all such cases yields a unique outcome. The longest match, left to right replace operator, `@->`, defined in Karttunen (1996), imposes a unique factorization on every input. The replacement sites 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 replaced. Thus the `@->` operator allows only the last factorization in the figure above, mapping "aba" unambiguously to "x".

The effect of the left to right, longest match constraints is that every string in the upper language of `A @-> B` is uniquely parsed into a sequence of substrings that either belong or do not belong to `A`. We can take advantage of the unique factorization in more than one way. Instead of replacing the instances of `A` by a string from some other language, we may insert markers or brackets around the `A` strings to mark them as such.

To implement this idea, Karttunen (1996) introduced a special symbol `...` on the right-hand side of the replacement expression to mark the place around which the insertions are to be made. The general form of these marking expressions is

shown below. For the sake of generality we allow `B` and `C` to denote any regular language.

`A @-> B ... C` Left to right, longest match markup.

The corresponding transducer locates instances of `A` in the input string under the left to right, longest match regimen, copies the entire string unchanged except that the `B` and `C` strings are inserted around the selected `A` strings as markers. In effect, this transducer can be viewed as a parser that picks out maximal instances of the regular language `A`.

Just like simple replacement, the left to right longest match replacement can also be constrained by context and generalized for parallel replacement. For example

```
A1 @-> B1 ... C1, A2 @-> B2 ... C2
```

picks out the maximal instances of the union `[A1 | A2]` and marks them differently depending on whether they belong to `A1` or `A2`. We will make use of this specific possibility below in Section 3.2. The contexted version of `@->` makes its appearance in Section 4.1.2.

To start with a simpler example, let us assume that noun phrases consist of an optional determiner, `(d)`, any number of adjectives, `a*`, and one or more nouns, `n+`. The expression `(d) a* n+ @-> %[ ... %]` compiles into a transducer that inserts brackets around maximal instances of the noun phrase pattern. For instance, it maps "dannvaan" into "[dann]v[aan]".

```
  d a n n   v   a a n
  -------       -----
[ d a n n ] v [ a a n ]
```

Although the input string "dannvaan" contains many other instances of the noun phrase pattern, "n", "an", "nn", etc., the left to right and longest match constraints pick out just the two maximal ones.

This simple example demonstrates that finite state parsers can be compiled directly from regular expressions. In the next section we will present a more sophisticated example illustrating this technique.

3 A Grammar and a Parser for Date Expressions

It is well-known among linguists that the syntax of a natural language cannot in general be described by a finite state, or even a context free grammar. Nevertheless, 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. For some of these kinds of expressions, a finite state grammar may be more appropriate and easier to construct than an ordinary phrase structure or feature based grammar. In this section, we examine one such case in detail: a grammar for dates. As we demonstrated in the previous section, from the regular expression that defines the syntax of well-formed date strings we can directly derive a finite state transducer that marks them in a text.

For the sake of illustration, let us consider here only one of several common date formats, expressions that are of the type

Sunday
August 11
Sunday, August 11
August 11, 1996
Sunday, August 11, 1996

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 leads to the same problem we encountered in the previous example. Long date expressions, such as "Sunday, August 11, 1996", contain smaller well-formed date expressions, e.g. "August 11", 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 larger phrases. The complete set of definitions is shown below:

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

¿From these definitions we can compile a small finite state automaton (13 states, 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.

```
DateExpression @-> %[ ... %]
```

It yields a transducer of 23 states and 332 arcs that marks maximal date expressions in the manner illustrated by the following text.

```
Today is [Wednesday, August 28, 1996] because yesterday was [Tuesday]
and it was [August 27] so tomorrow must be [Thursday, August 29] and
not [August 30, 1996] as it says on the program.
```

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

However, as to correctness, this regular expression grammar suffers from a serious problem of overgeneration. The actual number of days in 9999 years is much smaller than the number of expressions in the language ($\approx$ 30 million). A large majority of the date expressions generated by the grammar refer to dates that do not exist. For example, expressions like "April 31, 1996" are similar to examples like "the present

king of France" that do not refer to anything real. It is an interesting challenge for finite state syntactic description to try to specify a sublanguage that contains all and only the semantically valid date expressions.

3.1 Eliminating Invalid Date Expressions

In this section we eliminate step by step all the different types of invalid date expressions using the operations of the regular expression calculus introduced earlier.

The easiest problem to correct is that the original date grammar allows expressions like "February 30" and "April 31" that exceed the maximum number of days for the month. Somewhat more challenging is the case of leap days. "February 29, 1996" and "February 29, 2000" are valid dates but "February 29, 1994" and "February 29, 1900" do not exist. The hardest problem is the dependency between the day of the week and the date. Because September 16, 1996 is in fact a Monday, the expressions "Tuesday, September 16, 1996", "Wednesday, September 16, 1996", etc. are invalid even if they occasionally occur in real texts.

Our solution is to construct a suitable constraint for each of these three kinds of invalid types of dates: `MaxDaysInMonth`, `LeapDays`, and `WeekDayDates`. Each of these constraint expressions denotes a language that excludes a particular type of invalid date but admits all other strings. We obtain the desired effect by intersecting the constraint languages with the original language of date expressions. The intersection of the four languages contains all and only the valid dates:

```
ValidDate =
  DateExpression & MaxDaysInMonth & LeapDays & WeekDayDates
```

The `MaxDaysInMonth` constraint is a language that includes all strings except the ones which contain a month name followed by an inappropriate number of days:

```
MaxDaysInMonth =
  ~$[ February " " 3 %0 |
      [February | April | June | September | November] " " 3 1 ]
```

Note that `~$[A]` denotes the complement (negation) of the set of strings that contain at least one instance of `A` somewhere.

In order to restrict "February 29" to leap years we need to do a little more work. Not all years divisible by four are leap years. Full centuries are not leap years unless they are divisible by 400. Consequently, year 1900 is not a leap year but year 2000 is. We need the following definitions:

```
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]]
LeapYear = 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 `LeapYear` 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 `LeapYear` is defined as `Div4` minus this set, it follows that 2000 is a leap year but 1900 is not.

Once the set of leap years is defined, the distribution of "February 29" in date expressions can be constrained with the following simple restriction. (As defined above, SP is a separator consisting of a comma and a space.)

```
LeapDays = February " " 2 9 SP   => _ LeapYear .#.
```

In other words: a date expression containing "February 29, " must terminate with a leap year. Note that 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 last problem, synchronization between the days of the week and the dates takes a little more work. In effect, we need to construct a complete calendar to determine the validity of any arbitrary date expression, say, "Friday, October 15, 1582", the day the Gregorian calendar was introduced in Catholic countries.

The task is easier than it first appears because the Gregorian calendar includes a perfect 400 year double cycle: after each 400 years we start the year on the same day and we are at the same point in the leap year sequence. Starting from that observation, it is simple to partition years into equivalence classes so that the years in each class begin on the same day of the week. Furthermore, we need to distinguish ordinary years from leap years that have one extra day. Thus we can define fourteen equivalence classes:

```
MonY = .... | 1973 | 1979 | 1990 | 2001 | ....
TueY = .... | 1974 | 1985 | 1991 | 2002 | ....
....
MonLY = .... | 1912 | 1940 | 1968 | 1996 | ....
TueLY = .... | 1924 | 1952 | 1980 | 2008 | ....
....
```

where `MonY`, `TueY`, etc. are ordinary years beginning with Monday, Tuesday, etc. respectively; `MonLY`, `TueLY`, etc. denote the corresponding classes for leap years.

The year classes can be constructed using the same sort of finite state arithmetic that we employed above to define the infinite set of leap years, but we will skip the mechanics here except for one interesting detail. In order to avoid listing years one by one, it is convenient to derive subclasses from other subclasses by addition. This can be achieved by an auxiliary transducer that maps any set of numbers x, y, z, ... to another set containing $x+1$, $y+1$, $z+1$, etc. Applied in the opposite direction, the same transducer subtracts 1.[2]

[2] A binary version of the adder/subtracter transducer is one of the examples discussed in `http://www.rxrc.xerox.com/research/mltt/fst/fsexamples.html`.

Another simple observation that greatly facilitates the calendar construction is that the weekday of the first day of the year determines the weekday of all the other days of the year. Again, we need fourteen equivalence classes:

```
D1 = January 1 | January 8 | .... | December 31
D2 = January 2 | January 9 | .... | December 25
....
D1L = January 1 | January 8 | .... | December 30
D2L = January 2 | January 9 | .... | December 31
....
```

where `D1` includes all the dates that fall on the same weekday as the first day of an ordinary year; `D2` contains all the dates that share the weekday with the second day, and so on. The `D1L`, `D2L`, etc. are the corresponding classes for leap years.

To construct the final filter for the perfect Gregorian calendar we just need to combine the information about what weekday begins what year with the equivalence classes for the dates within a year. For example, for an ordinary Monday year (`MonY`), Mondays fall on January 1, January 8, ..., and December 31. But if the year begins on Sunday (`SunY`), then Mondays fall on January 2, January 9, etc., up to December 25. Similarly for all the fourteen equivalence classes for years. The complete constraint for Mondays, `MondayDates` is expressed by the following regular expression.

```
MondayDates =
[ Monday => _ (SP  [ D1 (SP  MonY) | D1L (SP  MonLY) |
                     D2 (SP  SunY) | D2L (SP  SunLY) |
                     D3 (SP  SatY) | D3L (SP  SatLY) |
                     D4 (SP  FriY) | D4L (SP  FriLY) |
                     D5 (SP  ThuY) | D5L (SP  ThuLY) |
                     D6 (SP  WedY) | D6L (SP  WedLY) |
                     D7 (SP  TueY) | D7L (SP  TueLY) ]) .#. ]
```

The language denoted by the expression above includes all strings that do not contain any instance of "Monday" and all strings that contain "Monday" in an environment that conforms to the restriction: the end of the string ("Monday"), any month and a date without a year ("Monday, December 5"), or a month and a date followed by an appropriate year ("Monday, January 1, 1996"). If a month and a date are followed by a year, the month and date class, `D1`, `D1L`, `D2`, etc., must correlate with the year class, `MonY`, `MonLY`, `SunY`, etc.

The complete `WeekDayDates` constraint is simply the intersection of the similar constraints for all the seven weekdays:

```
WeekDayDates = MondayDates & TuesdayDates & WednesdayDates &
               ThursdayDates & FridayDates & SaturdayDates &
               SundayDates
```

We have now completed the task of extracting the language of valid dates from the set of all date expressions; it is the intersection of the four languages we have defined:

```
ValidDate = DateExpression & MaxDaysInMonth &
            LeapDays & WeekDayDates
```

If we consider the finite state network that encodes the language, we can see that the number of expressions is considerably reduced since there are only about about 3.7 million days in between year 1 and year 9999. On the other hand, the network itself is much larger than the unconstrained `DateExpression` automaton: 1346 states, 21006 arcs.

In any case, we can now easily derive a transducer that marks all and only the valid dates:

```
ValidDate @-> %[ ... %]
```

With the Xerox regular expression compiler, the entire computation creating the date-parsing finite state transducer takes about 10 seconds on a powerful workstation (Sun Ultra 1).

3.2 Discriminating Date Parser

Although it is desirable to distinguish valid dates from invalid dates, it may not be useful in practice to recognize only the valid dates. Because of errors and misprints, real text corpora contain a fair number of invalid dates. Instead of ignoring all but the valid ones, it may be more practical to accept all date expressions and simply use different tags to mark the distinction between valid and invalid dates.

Once we have defined both the language of all date expressions, `DateExpression`, and the set of valid ones, `ValidDate`, it is simple to pick out the set of invalid dates: `[DateExpression - ValidDate]`. Using the notion of parallel replacement introduced earlier, we can easily construct a parser that recognizes maximal instances of the general class of date expressions but tags them in two ways depending on which subclass the expression belongs to. The discriminating date parser is defined by the regular expression below.

```
[ [DateExpression - ValidDate] @-> "[ID " ... %] ,
  ValidDate @-> "[VD " ... %] ]
```

This parallel replacement expression compiles into a 1338 state, 20862 arc transducer in about 15 seconds on a Sun Ultra 1. 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 Monday, September 16, 1996]. There
is an error in the program. Today is not [ID Tuesday, September 16,
1996].
```

The string "Monday, September 16, 1996" gets marked with the `"[VD "` tag because it is a valid date. Replacing "Monday" by "Tuesday" makes the date invalid but the parser still recognizes the expression "Tuesday, September 16, 1996" as a date expression, albeit as an invalid one. Because the longest match constraint is calculated with respect to the language of all date expressions, the invalid date "Tuesday, September 16, 1996" gets selected over "Tuesday, September 16, 19" which happens to be valid, although the Gregorian calendar of course was not yet in use in the year 19 AD.

To conclude this section, let us recapitulate the main points. The purpose of this exercise was to demonstrate that

- Finite state parsers can be constructed directly from regular expressions for nontrivial languages.
- There are regular subsets of natural language, such as the language of dates, for which finite state description is not only feasible but more appropriate and easier to construct than the equivalent phrase structure grammar.
- Regular languages and relations can be modified directly with the finite state calculus to obtain new languages and relations without rewriting the grammars that describe them.

It is of course possible to use phrase structure rules, attribute-value matrices, type hierarchies, categorial grammar, and other powerful formalisms to represent leap years, years starting on a Monday, and dates that fall on the same weekday as January 1. However it is not apparent that these powerful formalisms give us any advantage over the simple finite state techniques we have used. On the contrary, it seems that the manipulations required to accomplish such tasks with more powerful formalisms would be at least as complicated and ad hoc as the simple finite state techniques employed here. Note that a phrase structure grammar of valid dates requires cross-classification, which leads to a large number of rules and nonterminal categories.

If the language to be described is in fact regular, there may be a significant advantage in describing it by means of a regular grammar instead of using a more powerful grammar formalism. In the example just discussed, we were able to obtain the language of invalid dates by subtracting one regular language from another one. There would be no straightforward method to achieve a similar result starting with a phrase structure grammar since context free languages in general are not closed under complementation.

The next sections will add more examples supporting this general line of argument and present other applications of the finite state calculus to natural language processing.

4 Applications of the finite state calculus

4.1 Tokenization

One of the very first steps in any natural language processing system is applying a *tokenizer* to the input text. A tokenizer is a device that segments an input stream into an ordered sequence of *tokens*, each token corresponding to an inflected word form, a number, a punctuation mark, or other kind of unit to be passed on to subsequent processing. If the output never contains alternative segmentations for any part of the input, the tokenizer is called *deterministic*. Deterministic tokenization is commonly seen as an independent preprocessing step unambiguously producing items for subsequent morphological analysis.

In our approach, tokenization is an integral part of language processing, which can

be adapted to the needs of the subsequent analysis steps. Depending on the following steps, one might want to invoke different tokenization algorithms. In section 4.1.1, we describe a deterministic tokenizer which is useful for stochastic part of speech disambiguation. Then in Section 4.1.3 we sketch a situation in which we might need nondeterministic tokenization and describe how that is achieved.

4.1.1 Deterministic Tokenization

The simplest kind of deterministic tokenizer is an *unambiguous*[3] transducer that splits the input stream into a unique sequence of tokens, one per line, taking into account some general character classes but not using any language-specific information:

```
Letter =     [ A | B | C |  ......  | x | y | z ]
WhiteSpace = " " | "\t" | "\n"
Other =      ? - Letter - WhiteSpace
```

With these definitions, we can compile a simple tokenizing transducer that inserts newlines to mark token boundaries from the following regular expression. It represents the composition of three simple replace relations, as explained below.

```
WhiteSpace+ @-> " "
.o.
Letter+ @-> ... "\n", Other+ @-> ... "\n"
.o.
" " -> [] || .#. | "\n"  _
```

The first relation reduces strings of whitespace characters into a single blank using longest match replacement. The second inserts a newline as a token boundary after longest matches of letter sequences and other non-whitespace sequences. The third formula, a contexted replace expression, denotes a relation that eliminates any initial space and all spaces that follow a token boundary. The composite single transducer consists of 5 states and 170 arcs. It is unambiguous: every sequence of input characters is mapped into a unique sequence of tokens.

Exactly the same tokenization can also be obtained by compiling the three replacements separately and applying the resulting transducers in a sequence, each modifying the output of the previous one. This step by step approach, a cascade of transductions, will be discussed in section 4.1.2.

Extending this simple tokenizer, we can create more sophisticated tokenizer transducers which are language-specific. In order to divide the input text into sentences and words recognized by subsequent processing steps, these tokenizers need to know about abbreviations, conjoined clitics, and multiword expressions that contain internal spaces, hyphens, and other special symbols. A lexicon compiled as a finite

[3] The most efficient transducer is not only unambiguous but *sequential* as well. A sequential transducer produces a unique mapping without ever exploring any alternatives (Mohri 1996). An unambiguous transducer can be sequentialized if all local ambiguities can be resolved with the help of a limited amount of lookahead. Tokenizing transducers are in general of this type.

state language (`TokenLexicon` below) containing such exceptional tokens can be included in the middle line of the preceding formula.

```
[Letter+ | TokenLexicon ] @-> ... "\n", Other+ @-> ... "\n"
```

Because we use the longest match operator, any spaces and punctuation characters in the entries of `TokenLexicon` are mapped to themselves and do not count as token boundaries. For example, if the lexicon includes the strings "-tu", "l' ", and "à côté de", a French tokenizer would split the string "Vois-tu l'arbre à côté de la maison?" *'Do you see the tree next to the house?'* into the following tokens (token-bounding newlines are written as ||):

`Vois||-tu||l'||arbre||à côté de||la||maison||?||`

Multiword tokens are of several types:

- adverbial expressions ("all of a sudden", "a priori", "to and fro")
- prepositions ("in front of", "in spite of", "with respect to")
- dates ("January 2nd"), time expressions ("2:00 am")
- proper names ("New York", "Rank Xerox")
- and other units

A word of warning is in order. An unambiguous tokenizer analyzes every multiword expression consistently as a single token, even if the component words should be separated in a given context. For example, if "in general" is included in the multiword lexicon, then it will be tokenized as a unit in both of the examples: "in general, he ..." and "in general meetings". Therefore, multiword lexicons used for unambiguous tokenization must be carefully and conservatively designed.

4.1.2 Tokenization by step by step transduction

If the multiword lexicons are very large, their compilation into a single transducer may lead to time and space problems. Moreover, different NLP applications may require different multiword lexicons. For these reasons, it may be advantageous to use another approach, based on multiple transducers that apply in a sequence to yield exactly the same end result as the single tokenizer just described. In general, such a sequence consists of

- a *basic tokenizer* which segments any sequence of input characters into simple tokens (i.e. no multiword units) and
- one or several *multiword staplers* which identify multiwords and group them together as single units.

The basic tokenizer is compiled as described in the previous section. The multiword staplers are built in the following way. We first define a multiword language, `MWL`, containing the units to be recognized. The definition assumes the basic tokenization has already been done; the internal word separator is a newline instead of a space. For example, if the multiword expressions consist of "ad hoc" and "and so on", we define the language as follows:

```
MWL = [a d "\n" h o c | a n d "\n" s o "\n" o n ]
```

In order to define a relation that staples together the MWL expressions, it is useful to start with some auxiliary definitions.

```
BEG = ["<<"]   END = [">>"]   BND = [BEG | END]   LIM = ["\n" | .#.]
```

The **BEG** and **END** brackets are markers for the multiword string. The **LIM** expression is used to check the surrounding context making sure that the beginning and the end of the candidate multiword expression are not part of some other token. The stapler is composed from the three auxiliary relations below:

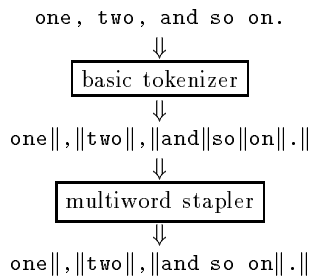
```
Identify = [~$[BND] .o. [MWL @-> BEG ... END || LIM _ LIM ]]
Staple =   ["\n" -> " " || BEG ~$[BND] _ ~$[BND] END]
Cleanup =  [BND -> []]
```

The **Identify** relation wraps the multiword expressions in **MWL** inside a pair of auxiliary brackets, << >>, under the left to right, longest match regimen imposed by **@->** and under the constraint that the multiword string is properly delimited.[4] The **Staple** relation converts every internal newline in a marked region into a space leaving the final one unchanged. The **Cleanup** relation eliminates the auxiliary brackets.

The multiword stapler for the **MWL** expressions is the composition of the three relations defined above:

```
Stapler = [Identify .o. Staple .o. Cleanup]
```

The sequential application of the basic tokenizer and the multiword stapler is illustrated in the figure below. As before, we use || to represent the newline symbol in order to save space.



The basic tokenizer can of course be composed with the multiword staplers to form a single, larger transducer if increasing the speed of application is more important than the size of the network.

If the stapling of some multiword expressions is made optional, the tokenization becomes nondeterministic because the multiword interpretation of the **MWL** string is an alternative to the sequence of single word tokens produced by the basic tokenizer. The next section discusses some cases where it is advantageous not to use deterministic tokenization.

[4] The same logic could be encoded without the contexted version of the **@->** operator but the details would be quite a bit more complicated. The problem is to avoid errors such as picking out "ad hoc" as a multiword in the middle of "bad hock".

4.1.3 Nondeterministic Tokenization

The deterministic treatment of multiword expressions as single tokens is problematic because many such expressions have alternate analyses in different contexts. For instance, the string "de même" in French can be treated as a single token, meaning *similarly*, or a sequence of two independent tokens: the preposition "de" *of* followed by the adjective "même" *same*. If the unambiguous tokenizer makes a wrong choice, it may lead to a parse failure or incorrect semantic interpretation. In such cases, a cautious tokenizer produces alternative segmentations postponing the decision to a later processing stage.

With the techniques just introduced, it is easy to make a tokenizer that produces alternative segmentations for some strings. We start by creating a special multiword lexicon for strings such as "de même" that should be analyzed either as a single token or as a sequence of tokens. If we are using the step by step approach in the previous section, we introduce into the cascade a second, ambiguous stapler transducer that optionally adjusts the output of the basic tokenizer for these potential multiword items. This optional stapler is defined exactly like the unambiguous stapler except that we include the universal identity relation, `?*`, to allow for any string to be mapped to itself.

```
OptionalStapler = [[Identify .o. Staple .o. Cleanup] | ?*]
```

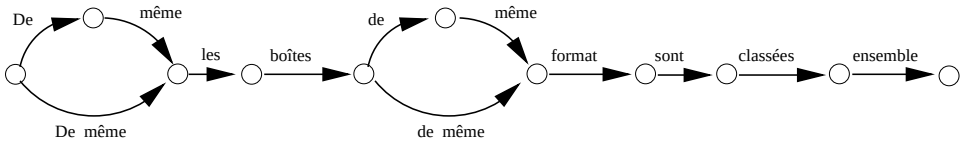
This optional stapler maps the output of the basic tokenizer, "de||même||", into both "de||même||" (identity) and "de même||" (stapled).

Making tokenization nondeterministic solves one problem but introduces another one. The subsequent stages of processing have to deal with the ambiguous representations of the input. This problem was first addressed in the context of a constraint-based finite state parser for French (Chanod and Tapanainen 1996; Chanod and Tapanainen 1997) that builds a finite state network for the input sentence that represents not only the alternative tokenizations but also all the additional ambiguities arising from the morphosyntactic analysis of the tokens. Each path through the network represents one possible tokenization and one possible morphosyntactic analysis for each token. Because of alternative tokenizations, the paths in general do not have the same number of tokens.

At this level, some of these paths can be quickly eliminated by syntactic constraints. Syntactic constraints are expressed as regular expressions, typically containing the restriction operator, and compiled to networks. These automata are intersected with the sentence network to prune out unwanted readings. In particular, they remove unacceptable tokenization schemes, unless the ambiguity is syntactically acceptable.

For instance, the sentence: "De même les boîtes de même format sont classées ensemble" *'Similarly the boxes of same format are classified together.'* is ambiguous at the tokenization level, because of the ambiguous string "de même". This leads to four different paths in the input network, as far as tokenization is concerned[5]:

[5] There are actually many more paths, as the input network represents not only tokenization ambiguity but also syntactic ambiguity.



However, after syntactic analysis, there remains only one analysis where the first "de même" is recognized as a multiword adverbial, while the second one is decomposed into two independent tokens:

```
de même  +Adv +Cap +MWE                  +Adverbial
le       +InvGen +PL +Def +Det           +NounPrMod
boîte    +Fem +PL +Noun                  +Subject
de                                       +Prep
même     +InvGen +SG +Adj                +NounPrMod
format   +Masc +SG +Noun                 +PPObj
être     +IndP +PL +P3 +Verb +Copl +Auxi +MainV
classer  +PaPrt +Fem +PL +Verb           +PastPart
ensemble +Adv                            +Adverbial .
```

This is due to the syntactic constraints that reject unwanted analyses, including incorrect tokenizations. For instance, the path where the two occurrences of "de même" are split into two tokens is rejected by several constraints, among which is the following constraint:

```
Prep  =>  _
      Coord Prep |                                          (a)
      ~$[ NounHead | VerbHead | Prep ] PPObj                (b)
      ~$[ NounHead | VerbHead | Prep ] [ Inf | PresPart ]   (c)
      Adverbial |                                           (d)
      NounPsMod                                             (e)
```

This constraint allows the `Prep` symbol to appear only in the given five contexts that describe the possible continuations for a preposition in a French sentence:

(a) before a coordinated preposition
(b) before a PPOBj without prior head nouns or head verb
(c) before an infinitival or participial verb
(d) before an adverbial
(e) before an adjective on the condition that it is a noun postmodifier.

None of these contextual constraints accepts the sequence `[Prep NounPrMod Det Subject]` on the path corresponding to "`De`||`même`||`les`||`boîtes`". This path is then eliminated from the input sentence network.

If, for a given sentence, two ambiguous tokenization paths are syntactically acceptable, they are both preserved after intersection with the constraint networks. This is what happens with the sentence: "Je pense bien qu'il parle" where "bien que" is ambiguous (meaning *although* as a multiword token). The sentence can be

read either as lit.: "I think well that he speaks", i.e. *I do think that he speaks* or *I think although he speaks*, which leads to two remaining analyses (paths) in the output sentence network.

4.2 Light Parsing by Marking and Filtering Transducers

The preceding subsections have shown applications of the finite state calculus to parts of natural language processing such as tokenization and morphological analysis used by part of speech tagging. Earlier, Section 3 showed that finite state parsers can be created for some subsets of natural language, such as correct and incorrect dates. Here we present a full scale light parser built from the finite state calculus described in Section 2.

For some large scale text applications, such as terminology extraction, lexicography, or information retrieval, a parser must recognize recurring lexical syntactic patterns and their variants. The parser need be no more powerful than is necessary to recognize these patterns. Many such parsers (Joshi 1961; Debili 1982; Grefenstette 1992; Abney 1991; Appelt *et al.* 1993) employ recognizers over part of speech tagged text by first marking contiguous patterns such as noun and verb groups, then marking heads within groups, and then extracting patterns between noncontiguous heads. Some of these parsers mix non-finite-state procedures with finite state recognizers, but we show here that the entire parser can be built within a finite state framework.

A finite state transducer that introduces extra symbols into an input string can be considered as a *finite state marker*. As seen in Section 2.3, the longest match operator can be used to introduce marks around nominal groups and verbal groups, and simple transducers can be used to mark head words within these groups. A *finite state filter* is a transducer which outputs only certain parts of the input string, setting all the other parts of the string to epsilon and possibly introducing a filter-indicating label. Schematically a filtering transducer has the following structure[6]:

```
0:RelationLabel  [ LEFTCONTEXT .x. [] ]
   token  [ MIDDLE .x. [] ]
   token  [ RIGHTCONTEXT .x. [] ]
```

Transducers implementing finite state markers can be composed with transducers implementing filters to create a finite state parser which can extract and label a wide variety of n-ary syntactic dependency relations between words.

We have created a finite state light parser in the following way: (1) using the longest match and replacement operators, transducers are created which identify contiguous noun group and verbal group boundaries; (2) labeling transducers are described which mark the nominal or verbal heads within each group; and (3)

[6] First described in a different notation in (Debili 1982, p. 99). Recall that `0` represents the empty string. Therefore, `0:a` introduces "a" into the lower-side string, and `B .x. []`, the cross-product of the language `B` with the empty-string language, eliminates the `B` strings from the output.

filtering transducers are defined which extract and label the syntactic relations between words within and across group boundaries.

4.2.1 Phrasal Mark-up

Rule-based descriptions of phrase contours can be done in a variety of ways, the light parser described here uses a finite state calculus representation of part of speech precedence matrix (Debili 1982) phrasal descriptions. Here is a simple example, a regular expression describing a French nominal phrase using the restriction operator to implement the rows and columns of such a precedence matrix:

```
NominalPhrase =
      [ [ Art   => _ [ Noun  ]] &
        [ Noun  => _ [ PAdj | Prep | .#. ]] &
        [ PAdj  => _ [ PAdj | Prep | .#. ]] &
        [ Prep  => _ [ Art  | Noun ]] &
        [ [ Art | Noun ] [ Art | Noun | Padj | Prep ]* ] ]
```

The expression states that a nominal phrase can contain an article followed by a noun, a noun followed by a postposed adjective or by a preposition or nothing (`.#.` means the end of an expression), a postposed adjective followed by another adjective or by a preposition, etc. The nominal group must begin with an article or a noun. We can derive from such a definition a phrase marking transducer using the directed replace operator `@->` and the three dots (`...`) symbol to mark the insertion points around the longest instances of these groups in each tagged sentence present in input.

```
MarkNGroup = [NominalPhrase @-> "[NG " ... " NG]"]
```

Such phrasal patterns can easily be written for different languages or different tagsets or with different noun-phrase definitions in a given language, e.g. for terminology extraction the determiners might be omitted in the definition of the noun phrase pattern. Schiller (1996) describes a language-independent architecture for finite state noun phrase extraction built on our finite state tools and taggers.

The phrasal markup, however, can also be used as a step towards further analysis leading to syntactic function extraction, as shown below.

4.2.2 Head Marking

Once the group markers are inserted by the transducers described above into the tagged text, another transducer places *head labels* before certain classes of words within the groups. These additional labels indicate that the words appear in specified contexts and make the task of writing *syntactic function filters* easier and shorter.

Supposing that `NOUN` matches words tagged as nouns, `PREP` prepositions and `CC` conjunctions, nominal heads are marked as being modified by prepositions (`*P`) or as independent noun phrase heads (`*N`) by the transducers produced by the following regular expression:

```
[] -> "*N" || "[NG" _ NOUN [ ~$[ NOUN | "NG]" ] ] [ "NG]" | CC ]
.o.    "*N" -> "*P" || [ [$ PREP] &  ~$[ PREP | "NG]" ] _
```

These expressions state that the empty string `[ ]` is replaced with the symbol `*N` in front of a noun that is not followed by another noun in the same noun group. This transducer is composed with another which replaces the `*N` label with a `*P` when the label is preceded by a preposition in the same noun group with no intervening prepositions.

Verbal heads are similarly marked with aspect labels inside verbal groups. For example, `*PasV` marks verb heads in passive constructions.

Applying the phrasal marking transducers and the head markers over tagged text inserts the following markings into an example sentence (hiding the part of speech tags for legibility):

```
[NG Significant *N correlations NG] [VG were *PasV obtained VG] [NG
between the maternal and fetal glucose *P levels and the maternal and
fetal ffa *P levels NG] .
```

4.2.3 Syntactic Function Extraction

Having introduced nominal and verbal group delimiters and labeled heads within these nominal and verbal groups makes writing regular expression filters easier. For example, a filter that extracts passive subjects can be written as

```
0:"PassiveSubj>" [ ?* .x. [] ] "*N" NOUN
  [ ~ $[ "[NG" | "VG]" ] .x. [] ] "*PasV" VERB [ ?* .x. [] ]
```

Applying this filter to the following examples, extracts

```
PassiveSubj> correlations  obtained
```

A finite state light parser is constructed by aligning the phrase marking transducer, the head marker transducer, and a union of filtering transducers into one cascade. We constructed (Grefenstette 1997) and ran such a light finite state parser (which included 16 other filters describing other syntactic patterns) over the first megabyte of AP news from 1988, and randomly chose one hundred output sentences. On a SPARC-20, tagging the 164,000 words took about 15 seconds of real time, inserting noun and verb phrase boundaries via nonoptimized marking transducers took about 2 minutes, marking heads inside boundaries took about 3 minutes 12 seconds, and applying the union of 17 different filters took about 11 minutes. In all, this is about 10,000 words per minute. Manual evaluation showed that 80% of 50 randomly chosen `PassiveSubj` relations from these 8000 sentences were correct. These numbers can be improved by introducing more complicated filters, but already they provide useful indications of subcategorizations, see for example the passive subjects extracted for the word "killed": "people" (5 times), "seaman" (2), "villagers", "vendors", "teen-agers", "soldiers", "rebels", "pilot", "patient" etc.

5 Conclusion

We have presented numerous examples illustrating the application of the regular expression calculus to language engineering tasks ranging from tokenization to

lightweight syntactic analysis. There are many other types of applications that we have not discussed, because of space or because they are already well known, such as morphological analysis by lexical transducers. In particular, we would have liked to include a fuller discussion of rule-based disambiguation to illustrate the use of replace expressions in systems such as the Brill tagger (Brill 1992; Roche and Schabes 1995) and the constraint grammar parser (Karlsson *et al.* 1995).

Although regular expressions and the algorithms for converting them into finite state automata have been part of elementary computer science for decades, the restriction and replacement expressions we have focused on are recent. They have turned out to be very useful for linguistic applications. 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 well-established practice in finite state morphology. Our examples in this paper provide ample evidence of its utility in other areas of language engineering.

6 Acknowledgements

We would like to thank Kenneth R. Beesley and Annie Zaenen for their editorial advice. We also thank Annie Zaenen for many discussions about the feasibility of a regular grammar for valid date expressions and Pasi Tapanainen for his contributions to the sections on finite state disambiguation. We acknowledge Ronald M. Kaplan and Martin Kay, our Xerox colleagues, for the pioneering work on the approach we are taking in this paper.

References

Abney, Steven. 1991. Parsing by chunks. In Abney, Steven *et al.* (eds.), *Principle-Based Parsing*, Kluwer Academic Publishers, Dordrecht.

Appelt, Douglas E., Hobbs, Jerry R., Bear, John, Israel, David, and Tyson, Mabry. 1993. FASTUS: A finite-state processor for information extraction from real-word text. *Proceedings of IJCAI-93*, Chambery, France.

Beesley, Kenneth R. and Karttunen, Lauri. 1997. Finite-State Morphology. *Technical Report* (forthcoming). RXRC Grenoble.

Brill, Eric. 1992. A simple rule-based part-of-speech tagger. *Proceedings of ANLP-92*, Trento, Italy.

Chanod, Jean-Pierre and Tapanainen, Pasi. 1996. A Non-Deterministic Tokenizer for Finite-State Parsing. *ECAI-96 workshop on Extended Finite State Models of Language*, Budapest.

Chanod, Jean-Pierre and Tapanainen, Pasi. 1997. Finite-State Based Reductionist Parsing for French. Kornai, András (ed.), *Extended Finite State Models of Language.* Cambridge University Press, forthcoming

Debili, Fathi. 1982. *Analyse Syntaxico-Semantique Fondée sur une Acquisition Automatique de Relations Lexicales-Semantiques.* Ph.D. dissertation, University of Paris XI, France.

Grefenstette, Gregory. 1992. Use of syntactic context to produce term association lists for text retrieval. *Proceedings of SIGIR'92*, Copenhagen, Denmark. ACM.

Grefenstette, Gregory. 1997. Light Parsing as Finite-State Filtering. In Kornai, András (ed.), *Extended Finite State Models of Language.* Cambridge University Press, forthcoming

Grefenstette, Gregory and Tapanainen, Pasi. 1994. What is a word, what is a sentence? Problems of tokenization. *Proceedings of COMPLEX-94*, pp. 79–87, Budapest.

Joshi, Aravind. 1961 Computation of Syntactic Structure. In *Advances in Documentation and Library Science*, vol. III, part 2. Interscience Publishers.

Kaplan, Ronald M. and Kay, Martin. 1994. Regular Models of Phonological Rule Systems. *Computational Linguistics*, **20**: 331–378.

Karlsson, Fred, Voutilainen, Atro, Heikkilä, Juha and Anttila, Arto, (eds.), 1995. *Constraint Grammar: a language-independent system for parsing unrestricted text.* Mouton de Gruyter, Berlin and New York.

Karttunen, Lauri and Beesley, Kenneth R. 1992. Two-Level Rule Compiler. Technical Report. ISTL-92-2. October 1992. Xerox Palo Alto Research Center. Palo Alto, California.

Karttunen, Lauri. 1994. Constructing Lexical Transducers. *Proceedings of COLING-94* **I**, pp. 406–411, Kyoto, Japan.

Karttunen, Lauri. 1995. The Replace Operator. *Proceedings of ACL-95*, pp. 16–23, Boston, Massachusetts.

Karttunen, Lauri. 1996. Directed Replacement. *Proceedings of ACL-96*, Santa Cruz, California.

Karttunen, Lauri, Kaplan, Ronald M., and Zaenen, Annie. 1992. Two-level morphology with composition. *Proceedings of COLING-92*, **I**, pp. 141–148, Nantes, France.

Kempe, André and Karttunen, Lauri. 1996. Parallel Replacement in the Finite-State Calculus. *Proceedings of COLING-96*, **2** pp. 622–627. Copenhagen, Denmark.

Koskenniemi, Kimmo. 1983. Two-level Morphology. A General Computational Model for Word-Form Recognition and Production. Department of General Linguistics. University of Helsinki, Finland.

Koskenniemi, Kimmo, Tapanainen, Pasi, and Voutilainen, Atro. 1992. Compiling and using finite-state syntactic rules. *Proceedings of COLING-92*, **I**, pp. 156–162, Nantes, France.

Mohri, Mehryar. 1996. On Some Applications of Finite-State Automata Theory to Natural Language Processing. *Natural Language Engineering, 1:1.*

Palmer, David D. and Hearst, Marti A. 1994. Adaptive sentence boundary disambiguation. *Proceedings of ANLP-94*, pp. 78–83, Stuttgart, Germany.

Roche, Emmanuel and Schabes, Yves. 1993. Deterministic Part-of-Speech Tagging. *Computational Linguistics*, 21(2), 227–253.

Salton, Gerald, Zhao, Zhongnan, and Buckley, Chris. 1990. A simple syntactic approach for the generation of indexing phrases. Technical Report 90–1137, Department of Computer Science, Cornell University.

Schiller, Anne. 1996. Multilingual Finite-State Noun Phrase Extraction. *ECAI-96 Workshop on Extended Finite State Models of Language* Budapest.

Silberztein, Max. 1993. *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX*. Masson, Paris, 1993.

Voutilainen, Atro and Tapanainen, Pasi. 1993. Ambiguity resolution in a reductionistic parser. *Proceedings of EACL-93*, pp. 394–403, Utrecht.