An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing

Gertjan van Noord[1] and Dale Gerdemann[2]

[1] University of Groningen vannoord@let.rug.nl
[2] University of Tübingen dg@sfs.nphil.uni-tuebingen.de

Abstract. Finite-state techniques are widely used in various areas of Natural Language Processing (NLP). As Kaplan and Kay [12] have argued, regular expressions are the appropriate level of abstraction for thinking about finite-state languages and finite-state relations. More complex finite-state operations (such as contexted replacement) are defined on the basis of basic operations (such as Kleene closure, complementation, composition).
In order to be able to experiment with such complex finite-state operations the FSA Utilities (version 5) provides an *extendible* regular expression compiler. The paper discusses the regular expression operations provided by the compiler, and the possibilities to create new regular expression operators. The benefits of such an extendible regular expression compiler are illustrated with a number of examples taken from recent publications in the area of finite-state approaches to NLP.

1 Introduction

Finite-state techniques are widely used in various areas of Natural Language Processing (NLP). As Kaplan and Kay [12] have argued, regular expressions are the appropriate level of abstraction for thinking about finite-state languages and finite-state relations. More complex finite-state operations (such as contexted replacement) are defined on the basis of basic operations (such as Kleene closure, complementation, composition).

For instance, context sensitive rewrite rules have been widely used in several areas of natural language processing, including syntax, phonology and speech processing. Johnson [11] has shown that such rewrite rules are equivalent to finite state transducers under the assumption that they are not allowed to rewrite their own output. An algorithm for compilation into transducers was provided by Kaplan and Kay [12]. Improvements and extensions to this algorithm have been provided by Karttunen [13] [15] [14] and Mohri & Sproat [19]. Such algorithms take as their input regular expressions for the strings to be replaced and the left and right contexts, and produce a finite-state transducer. In other words, such an algorithm provides a new regular expression operator.

Many different variants of replacement operators have been proposed, depending on whether rewrite rules are interpreted left to right, right to left or in parallel; whether rewrite rules are required to use longest, shortest or all matches; whether rules are obligatory or optional; whether contexts should match the input side or the output side of the transductions etc. For this reason, it is crucial to be able to experiment with each of the various proposals in a flexible way.

Version 5 of the FSA Utilities [25] is an extended, rewritten and redesigned version of the FSA Utilities toolbox previously presented at the first WIA [24]. The FSA Utilities toolbox has been developed as a platform for experimenting with finite-state approaches in natural language processing. For this reason, the FSA Utilities toolbox is implemented in SICStus Prolog (cf. also section 5).

FSA5 provides a very flexible *extendible* regular expression compiler. Below, we present the basic regular expression operations provided by the compiler, and the possibilities to create new regular expression operators. We illustrate the exendible regular expression compiler with a number of examples taken from recent publications in the area of finite-state approaches to NLP.

2 Regular Expressions

`[]`	empty string
`[E1,E2,...En]`	concatenation of `E1, E2 ...En`
`{}`	empty language
`{E1,E2,...En}`	union of `E1, E2 ...En`
`E*`	Kleene closure
`E^`	optionality
`~E`	complement
`E1-E2`	difference
`$ E`	containment
`E1 & E2`	intersection
`?`	any symbol
`A:B`	pair
`E1 x E2`	cross-product
`A o B`	composition
`domain(E)`	domain of a transduction
`range(E)`	range of a transduction
`identity(E)`	identity transduction
`inverse(E)`	inverse transduction

Table 1. Basic regular expression operators in FSA5.

Table 1 gives an overview of the basic regular expression operators provided by FSA5. Apart from the standard regular expression operators and extended regular expression operators for regular languages, the tool-box also provides regular expression operators for regular relations. For example, the expression

```
{a:b,b:c,c:a}*
```
(1)

is the transducer which rewrites each `a` into a `b`, each `b` into a `c`, and each `c` into an `a`. Consider furthermore a transducer which removes each `b`, but which leaves each non-`b` in place:

```
{b:[],? -b}*                                                        (2)
```

In this example[1], the expression `? -b` is any symbol except `b`. An expression `Expr` denoting a regular language is automatically coerced in the context in which a transducer is expected into `identity(Expr)`. Here, `? -b` is automatically coerced into `identity(? -b)`, because it is unioned with a transducer. Composing the examples 1 and 2:

```
{a:b,b:c,c:a}* o {b:[],? -b}*                                       (3)
```

yields a transducer which removes each `a`, and transduces each `b` to a `c`, and each `c` to an `a`. For instance, the input `abcabcabc` yields `cacaca`.

In FSA5, such a regular expression could be turned into a transducer using the command:

```
% fsa -r '{a:b,b:c,c:a}* o {b:[],? -b}*' >  ex1.fa                  (4)
```

In this case, the resulting automaton is written to the file `ex1.fa` in FSA5 format. There are options to produce automata in many different formats, including formats for other finite-automata tool-boxes such as AT&T's `fsm` program [18] and various visualization formats (including `dot`, `vcg`, `daVinci`, LaTeX and `postscript`). Other interesting formats are as a Prolog or C program implementing the transduction.

FSA5 can also be used interactively. In that case a graphical user interface is provided from which regular expressions can be input. The resulting automata are then displayed on the screen, and the resulting automata can be tested with sample inputs. The availability of such a graphical user interface in combination with various visualization tools has enabled the use of FSA5 in teaching [3]. For more information on these and other possibilities refer to the FSA Home Page: `http://www.let.rug.nl/vannoord/Fsa/`. The FSA Home Page includes an on-line demo.

3 Extendible Regular Expression Operators

The regular expression compiler can be extended with new regular expression operators by providing one or more files defining these operators. The definitions are essentially of two types. In both cases, the actual definitions are written in (often very simple) Prolog. On the one hand, operators can be defined in terms of existing regular expression operators. On the other hand, regular expression operators can be defined by providing a direct implementation on the underlying automata. Many researchers prefer the first style. For instance, Kaplan & Kay [12] (p. 376) argue:

[1] For technical reasons a space is required after each occurrence of the `?` meta-symbol.

> The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs—the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine. The only hope of success in this domain lies in developing an appropriate set of high-level algebraic operators for reasoning about languages and relations and for justifying a corresponding set of operators and automata for computation.

Paradoxically, Mohri & Sproat improve upon Kaplan & Kay's algorithm by taking precisely the opposite approach. Their algorithm is primarily presented in terms of manipulations upon states and transitions within automata. One could perhaps translate Mohri & Sproat's algorithm into a high-level calculus, but a great deal of efficiency would be lost in the process. It is a testimony to the flexibility of FSA5, that these two approaches can both be implemented and combined (cf. section 4.3).

New operators in terms of existing operators. A regular expression operator is defined as a pair `macro(ExprA,ExprB)` which indicates that the regular expression `ExprA` is to be interpreted as regular expression `ExprB`. For example, simple nullary regular expression operators (equivalent to abbreviatory devices found in tools such as `lex` and `flex`), can be defined as in the following example:

```
macro( vowel, {a,e,i,o,u} )
```
(5)

indicating that the operator `vowel/0` can be understood by assuming that every occurrence of `vowel` in a regular expression is textually replaced by `{a,e,i,o,u}`.

The same mechanism is used to define n-ary operators, exploiting Prolog variables. For instance, the containment operator `containment(Expr)` is the set of all strings which have as a sub-string any of the strings in `Expr`. This could be defined as follows:[2]

```
macro(containment(Expr), [? *,Expr,? *])
```
(6)

Naturally, operators defined in this way can be part of the definition of other operators. For instance, the operator `free(A)` is the language of all strings which do not have any of the strings in `A` as a substring. This can be defined as:

```
macro(free(A), ~containment(A))
```
(7)

[2] Note that this operator is standardly available in FSA5. Many of the built-in operators in FSA5 are defined using the same technique.

We have found it useful to define boolean operators using this mechanism. In fact, if we use the universal language to stand for *true* and the empty language to stand for *false*, then the standard operators for intersection and union correspond to conjunction and disjunction:

```
macro(true,? *).                                                    (8)
macro(false,{}).
```

With these definitions we get the expected properties:

```
true & true   = true     {true,true}   = true                       (9)
true & false  = false    {true,false}  = true
false & true  = false    {false,true}  = true
false & false = false    {false,false} = false
```

The macros for true and false can also be used to define a conditional expression in the calculus. The operator `coerce_to_boolean` maps the empty language to the empty language, and any non-empty language to the universal language:

```
macro(coerce_to_boolean(E),                                        (10)
        range(E o (true x true))).

macro(if(Cond,Then,Else),
     {  coerce_to_boolean(Cond) o Then,
       ~coerce_to_boolean(Cond) o Else  }).
```

Various interesting properties of automata have been implemented which yield boolean values, such as the predicates *is_equivalent/2* for recognizers, and *is_functional/1* and *is_subsequential/1* for transducers (using the algorithms described in for instance [23]).

Regular expression operator definitions can also be recursive. The following example demonstrates furthermore that definitions can take the operands of the operator into account. The operator `set(List)` yields the union of the languages given by each of the expressions in the list `List`; `union(A,B)` is a built-in operator providing the union of the two languages `A` and `B`:

```
macro(set([]),'{}').                                               (11)
macro(set([H|T]),union(H,set(T))).
```

We can also exploit the fact that these definitions are directly interpreted in Prolog by providing Prolog constraints on such rules. This possibility is used in [7] to define a longest-match concatenation operator which implements the leftmost-longest capture semantics required by the POSIX standard (cf. section 4.4).

A simple example is a generalization of the operator `free`. Suppose we want to define an operator `free(N,Expr)` indicating the set of strings which do not contain more than N occurrences of Expr. This can be done as follows:

```
macro(free(N,X),~ [? *|List]) :-                                    (12)
    free_list(N,X,List).

free_list(0,X,[X,? *]).
free_list(N0,X,[X,? *|T]) :-
    N0 > 0, N is N0-1, free_list(N,X,T).
```

Another example is an implementation of the N-queens problem: how to place N queens on an N by N chess-board in such a way that no queen attacks any other queen. For any N we can create a regular expression generating exactly all strings of solutions. A solution to the N-queen problem is represented as a string of N integers between 1 and N. An integer i at position j in this string indicates that a queen is placed on the i-th column of the j-th row.

```
macro(n_queens(N), sigma(N)*                                        (13)
                 & length(N)
                 & columns(N)
                 & diagonals(N)
                 & reverse(diagonals(N)))
```

The operator n_queens(N) is defined as the intersection of a number of constraints. The first constraint, sigma(N)*, indicates that a solution must be a string of integers between 1 and N. The second constraint indicates that the length of the string must be N. The remaining constraints ensure that queens do not attack each other. The definition of length illustrates once more the use of Prolog to create a regular expression; the definition of sigma/1 uses the set operator defined previously.

```
macro(length(N),List) :- length(List,N), fill_qm(List).             (14)

fill_qm([]).
fill_qm([? | T]) :- fill_qm(T).

macro(sigma(N),set(L)):-
    findall(C,between(1,N,C),L).

between(N,_,N).
between(N0,N,I) :-
    N1 is N0+1, N1 < N+1, between(N1,N,I).
```

The complete program is given in the appendix. For instance, the expression n_queens(5) produces the automaton in figure 1.

The mechanism described sofar to define new regular expression operators is already quite powerful. As another illustration, consider the problem of compiling a given finite automaton into a regular expression. This problem becomes trivial

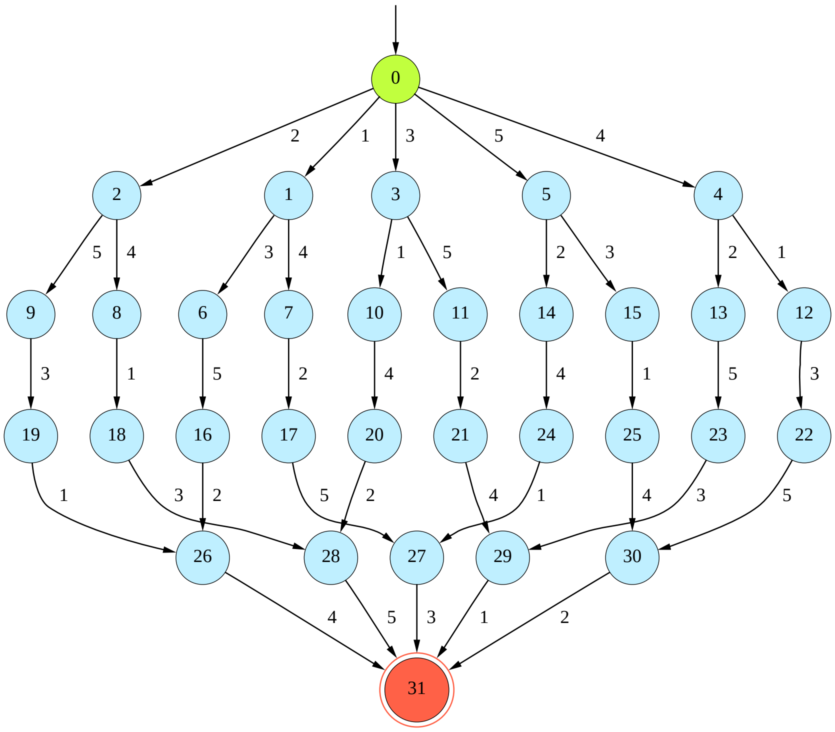

Fig. 1. Solution to the 5-queens problem

if we allow the introduction of new operators. Here is the definition of an operator 'fa/5' which describes an automaton as a listing of its components:

```
macro(fa(Sigma,States,Initials,Finals,Transitions),          (15)
  range(          % state-sym-state triples:
         ([[States,Sigma]*,States]
                & % no non-transition triples:
          free([States,Sigma,States]-Transitions)
                & % start in start-state:
          [Initials,? *]
                & % end in final state:
          [? *,Finals]
         )      o % get rid of state names:
         [[States x [],?]*,States x []]
       ))
```

As an example, the automaton given in figure 2.16 of [10] (given in figure 2) can be specified as:

```
fa({0,1},{q1,q2,q3},{q1},{q2,q3},                             (16)
     {[q1,0,q2],[q1,1,q3],[q2,0,q1],
      [q2,1,q3],[q3,0,q2],[q3,1,q2]})
```

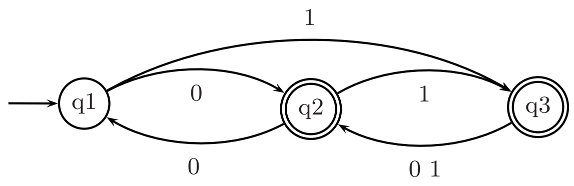

Fig. 2. Example automaton from figure 2.16 of [10].

Direct implementation of new operators. Some operators are more easily defined in terms of the underlying automaton. For instance, the operator `reverse(X)` is the set of all strings `Y` such that the reversal of `Y` is in `X`. In case the operand `X` is constructed by means of standard regular expression operators it would be possible to define the reverse operator recursively in terms of the various forms that `X` can take; in FSA however `X` could be constructed by means of various user-defined operators as well. Therefore this approach is not applicable. However, the operation is trivial to define in terms of the underlying automaton: each of the transitions needs to be swapped, final states become start states and vice versa. The (simplified) definition is given as follows:

```
rx(reverse(Expr),Fa) :-                                    (17)
    fsa_regex:rx(Expr,Fa0), reverse_fa(Fa0,Fa).

reverse_fa(Fa0,Fa) :-
    fsa_data:start_states(Fa0,Finals),
    fsa_data:final_states(Fa0,Starts),
    fsa_data:transitions(Fa0,Trans0),
    reverse_transitions(Trans0,Trans),
    fsa_data:construct_fa(Starts,Finals,Trans,Fa).

reverse_trans([],[]).
reverse_trans([trans(A,B,C)|T0],[trans(C,B,A)|T]) :-
    reverse_trans(T0,T).
```

As is typical in such definitions, the `fsa_regex:rx` predicate is used to construct an automaton for a given regular expression. The `fsa_data` module provides a consistent interface to the internal representation of automata. Its predicates can be used to select relevant parts of an automaton (such as start states, final states and transitions) and to construct automata on the basis of such parts.

4 Regular Expression Operators in NLP

This section illustrates the flexibility and the power of the FSA5 extendible regular expression compiler on the basis of a number of examples taken from recent publications in the field of NLP.

4.1 Lenient Composition.

In a recent paper, Karttunen [16] has provided a new formalization of Optimality Theory in terms of regular expressions. Optimality theory [20] is a framework for the description of phonological regularities which abandons rewrite rules. Instead, a universal function called GEN is proposed which maps input strings non-deterministically to many different output strings. In addition, a set of ranked universal constraints rule out many of the phonological representations generated by GEN. Some constraints can be conflicting. Therefore it might be impossible for a candidate string to satisfy all constraints. A string is allowed to violate a constraint as long as there is no other string which does not violate that constraint.

Procedurally, this mechanism can be understood as follows. Firstly, an input is mapped to a set of candidate output strings. This set of strings is then passed on to the most important constraint. This constraint removes many of the candidate strings. The remaining strings are passed on to the next important constraint, and so on. If the application of the constraint would remove all remaining candidate strings, then no strings are removed (constraints are *violable*). In the simplest case, only a single string survives all of the constraints. If none of the strings satisfy a given constraint, then the strings survive with the least number of violations of that constraint.

Karttunen formalizes GEN as a regular relation. Each of the constraints is itself a regular language allowing only the strings which satisfy the constraint (unless no strings satisfy the constraint). If the constraints were to be combined using ordinary composition, then the set of outputs would often be empty. Therefore, instead of composition Karttunen introduces an operation of *lenient_composition* which is closely related to a notion of defaults.

Informally, the *lenient_composition* of `S` and `C` is the composition of `S` and `C`, except for those elements in the domain of `S` that are not mapped to anything by `S o C`. Thus, it enforces the constraint `C` to those strings in `S` which have an output that satisfies the constraint:

```
macro(priority_union(Q,R), {Q, ~domain(Q) o R}).                (18)
macro(lenient_composition(S,C), priority_union(S o C,S)).
```

Here, `priority_union` of two transductions `Q` and `R` is defined as the union of `Q` and the composition of the complement of the domain of `Q` with `R`; i.e. we obtain all pairs from `Q`, and moreover for all elements not in the domain of `Q` we apply `R`. Lenient composition of `S` and `C` is defined as the priority union of the composition of `S` and `C` (on the one hand) and `S` (on the other hand); i.e. we obtain the composition of `S` and `C` and moreover for all inputs for which that composition is empty we retain `S`.

Consider the example

```
lenient_composition({b x [b,b],a x [b,b]*},[b,b,b]*)            (19)
```

The input transducer maps an `a` to an even number of `b`'s, and it maps a `b` to two `b`'s. If this transducer is leniently composed with the requirement that the result must be a string of `b`'s divisible by 3, then the resulting transducer maps b to two `b`'s, as before (since the constraint cannot be satisfied for any map of the input `b`), and it maps an `a` to a string of `b`'s which is divisible by 6.

Karttunen illustrates the method by providing a formalization of the syllabification analysis in Optimality Theory. This formalization has been implemented in FSA5 and is given in the appendix.

4.2 Priority Union for lexical analysis

Another application of the priority union operator is in spell checking. As in [4] we consider a finite-automaton approach. Suppose we are given a dictionary in the form of a transducer. The transducer will map each word to its lexicographic description. A spell checker attempts to find, for a given word, the lexicographic description of the word which is closest to a word in the dictionary according to some distance function. As in many spell checkers we assume Levenshtein distance: the minumum number of substitutions, deletions and insertions that is required to map a string into another. In FSA all strings with a Levenshtein distance of 1 can be defined as follows; here `X` can be thought of as the dictionary, `lev1(X)` is the Levenshtein-1 closure of the dictionary:

```
macro(lev1(X), { subs(X), del(X), ins(X) })                    (20)
```

The operators `subs/1, del/1` and `ins/1` are built-in. The expression `subs(X)` stands for all pairs (x, y) such that (x', y) is in the relation defined by `X` and x' can be formed from x by a single substitution. The insertion and deletion operators are defined likewise.

In contrast to [4] we want to obtain the candidates with minimal distance. For instance, if we attempt to lookup `book` then we don't want to get the description of `cook` as a result. This can be defined using the priority union operator as follows:

```
macro(spell1(X), priority_union(X, lev1(X)))                   (21)
```

For instance, applying `spell1` to a dictionary consisting of the identity transducer over the words *book, look, lock, oak* would map each of these words to itself, and in addition it would map a form such as *wook* to the set *book, look* and a form such as *ook* to the set *book, look, oak.*

We can define expressions for any given radius α. For example, the case which treats $\alpha = 2$ is given by:

```
macro(spell2(X), priority_union(spell1(X), lev1(lev1(X))))    (22)
```

4.3 The replace operator.

In [19] a variant of the replace operator is implemented which is more efficient than previous implementations provided by Kaplan and Kay [12] and Karttunen [13]. This improved version crucially depends on the possibility of manipulating the transitions and states of the underlying automata directly. The replacement of expression `Phi` into `Psi` in the context of `Left` and `Right` is written `replace(Left,Phi,Psi,Right)`. In the left-to-right interpretation, this operator can be defined as the following cascade:

```
macro(replace(L,Phi,Psi,R),                                     (23)
    r(R) o f(Phi) o replace(Phi,Psi) o l1(L) o l2(L))
```

This definition and the definitions of the auxiliary operators are closely modelled on those given in [19]. The auxiliary operators are defined in the appendix.

A typical example of the use of the replace operator is provided by the past tense endings of Dutch regular verbs. In Dutch, the singular past tense is formed by the `-de` and `-te` suffixes. If the previous phoneme is voiced, the suffix `-de` must be used; in order circumstances the `-te` suffix is appropriate. This phenomenon can be analysed by assuming an underlying, abstract, `-Te` suffix. The `T` is then transformed into a `d` or `t` depending on context. The rule can be defined as follows (the `+` indicates a morpheme boundary):

```
macro(tkofschip,                                              (24)
      replace([{k,f,s,[c,h],p,t,x},+],'T',t,e)
                                o
                  replace(+,'T',d, []))
```

4.4 Leftmost-longest contexted replacement.

In [7] a leftmost-longest match contexted replacement operator

```
lml(T,Left,Right)
```

is defined which ensures that the transducer `T` is applied in contexts `Left` and `Right`, using a leftmost-longest match strategy. One application of such an operator is finite-state parsing (chunking), [1, 5, 8, 22]. In finite-state parsing, sets of context-free rules are collected into levels. Typically there is a finite number of such levels, and these levels are ordered. First each of the rules of the first level apply. The result is then input to the second level, etc. Note that rules cannot work on their own output, unless the same rule is placed in several levels.

In the following example we will not use the contexts; therefore lml/1 is defined as:

```
macro( lml(T), lml(T,[],[]) )                                 (25)
```

This operator ensures that the transducer T is applied to a string at all possible positions, using a left-to-right left-most longest match policy.

In this particular example we will assume that the input to the finite-state parser is a tagged sentence: each word is represented by a category, an opening bracket, the word itself, and a closing bracket. A rule with a given left hand side and right hand side will look for the sequence of elements described by the right hand side and wrap the result inside left hand side brackets. In general, the macro `-->` can be defined as the following transducer (this macro disallows the case that the daughters is the empty string):

```
macro((A --> Ds), [[] x [A,'['], Ds-[], []:']'])              (26)
```

We use the macro `d(Expr)` for elements in the right hand side of rules; the macro `dw(Expr)` is similar but is used for pre-terminals, to refer to specific words.

```
macro(d(Cat),      [Cat, '[':'(',free(']'),']':')']).         (27)
macro(dw(Cat,Word),[Cat, '[':'(',   Word,  ']':')']).
```

Note that the brackets (introduced by an earlier level) are replaced here by other brackets in order to ensure that these brackets cannot be used in later levels again; in other words at any given level we can only 'see' the top-most constituents (yet, the full parse tree can be recoved using the 'invisible' brackets).

Using these two macro's a rule to recognize basic noun phrases is:

```
np --> [d(art)^,d(num)^,d(adj)*,d(n)+]                              (28)
```

A level of rules can now simply be defined as the replacement operator applied to the union of these rules. For instance, the following is a level recognizing multi-word-units (for instance, the Dutch phrase 'ten opzichte van' is comparable to the English phrase 'with respect to'):

```
macro(mwu,lml({                                                     (29)
  (p --> [dw(p,ten),dw(n,opzichte),dw(p,van)] ),
  (p --> [dw(p,in),dw(n,verband),dw(p,met)]   ),
  (p --> [dw(p,in),dw(n,plaats),dw(p,van)]    ) }))
```

Finally, we use composition to combine a number of such levels. Thus, the following expression defines a simple noun-phrase chunker:

```
macro(np_chunker,                                                   (30)
  mwu o lml(( adj --> [d(adv), d(adj)]))
      o lml((  np --> [d(art)^,d(num)^,d(adj)*,d(n)+]))
      o lml((  pp --> [d(p),d(np)]))
      o lml((  np --> [d(np),d(pp)+])))
```

For example, one of the sentences from the Eindhoven corpus [6] is chunked as in figure 3.

5 Implementational Issues

The regular expression compiler is defined in SICStus Prolog. This choice was motivated because the FSA Utilities toolbox has been developed as a platform for experimenting with finite-state approaches in natural language processing. Prolog allows for rapid prototyping of new techniques and variations of known techniques. The drawback is that the CPU-time requirements increase in comparison with an implementation based on C or C++. In [26] it is shown that the implementation of the determinizer is typically about 2 to 5 times slower in FSA Utilities than in AT&T's `fsm` library ([18]); the FSA Utilities toolbox contains a variant of the determinization algorithm for input automata with large amounts of ϵ-moves. In such cases FSA Utilities is often much faster. The implementation of the minimization algorithm [9, 2] is up to three times faster than the implementation described in [17], which was shown to be much faster than the corresponding implementations in Fire Lite [27] and Grail [21].

Regular expressions are read and parsed using the Prolog parser (i.e. regular expressions are read in as Prolog terms), exploiting the inherent flexibility of

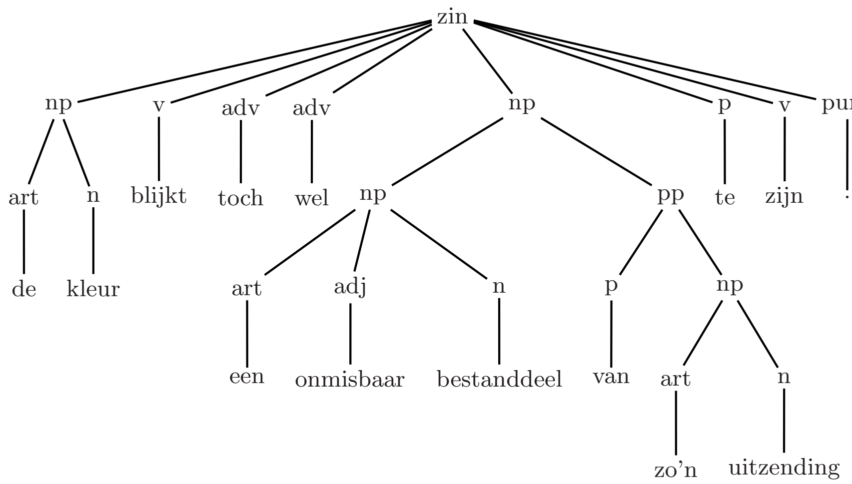

Fig. 3. Application of NP chunker

this parser (such as the possibility to declare that new operators may be written using operator syntax; therefore we can write `E1 o E2` instead of `o(E1,E2)`). The constructed term is straightforwardly compiled into a corresponding finite automaton using a simple top-down recursive-descent procedure.

This mechanism implies that in order to construct an automaton for a regular expression such as `[a*,b,c^,d+]` automata are constructed for each of the sub-expressions. For regular expressions which are constructed solely using such simple operators, more efficient automaton construction algorithms are known. We have not implemented these algorithms because of the desire to be able to treat user-defined operators. A possible improvement could be to have the compiler identify which parts of an expression are simple enough to be treated by a more efficient specialized algorithm.

The compiler supports caching of sub-expressions. If the cache facility is switched on, then the result of each sub-expression that is encountered will be cached for later re-use. This can increase efficiency for the compilation of a single expression, but it is especially useful in an interactive session where the user gradually alters the regular expression; typically a large part of the expression remains the same and interactive response time can be much more attractive.

The caching facility can also be used selectively. The `cache(Expr)` operator can be used to cache the result of the compilation of a specific regular expression `Expr`. For instance, in example 20 the expression `lev1(X)` is defined as { `subs(X), del(X), ins(X)` }. If we write instead:

```
macro(lev1(X), {subs(cache(X)),del(cache(X)),ins(cache(X))})
```
(31)

then `X` will be compiled only once.

The compiler supports a number of other operators which have an effect on the underlying automata, but not on the corresponding language or relation. For instance, the operator `determinize(Expr)` can be used to ensure that the resulting automaton is determinized. Similar operators provide a simple interface to various minimization algorithms provided by FSA5.

Furthermore, certain operators can be used for the sole purpose of obtaining a side-effect. One example was the `cache/1` operator discussed above. The operator `spy(Expr)`, for instance, can be used to request that the compiler provides progress information on the compilation of the expression `Expr` (size of the result, and CPU-time required to obtain the result). Such progress information is crucial for a better understanding of the sources of complexity of particular expressions.

Concluding Remarks

We have presented the extendable regular expression compiler of FSA5. We have shown that the functionality and flexibility provided by the toolbox can be used to experiment with a variety of finite-state techniques in natural language processing, including applications in phonology, morphology and syntax.

References

[1] Steven Abney. Partial parsing via finite-state cascades. In John Carroll, editor, *Workshop on Robust Parsing; Eight European Summer School in Logic, Language and Information*, pages 8–15, 1995.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Gosse Bouma. A modern computational linguistics course using dutch. In *EACL 99: Computer and Internet Supported Education in Language and Speech Technology. Proceedings of a Workshop sponsored by ELSNET and The Association for Computational Linguistics*, Bergen Norway, 1999.

[4] Christian S. Calude, Kai Salomaa, and Sheng Yu. Metric lexical analysis. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam Germany, 1999.

[5] Jean-Pierre Chanod and Pasi Tapanainen. A robust finite-state grammar for French. In John Carroll, editor, *Workshop on Robust Parsing*, Prague, 1996. These proceedings are also available as Cognitive Science Research Paper #435; School of Cognitive and Computing Sciences, University of Sussex.

[6] P. C. Uit den Boogaart. *Woordfrequenties in geschreven en gesproken Nederlands*. Oosthoek, Scheltema & Holkema, Utrecht, 1975. Werkgroep Frequentie-onderzoek van het Nederlands.

[7] Dale Gerdemann and Gertjan van Noord. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway, 1999.

[8] Gregory Grefenstette. Light parsing as finite-state filtering. In *EACI 1996 Workshop Extended Finite-State Models of Language*, Budapest, 1996.

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

[10] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[11] C. Douglas Johnson. *Formal Aspects of Phonological Descriptions*. Mouton, The Hague, 1972.

[12] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379, 1994.

[13] Lauri Karttunen. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, M.I.T. Cambridge Mass., 1995.

[14] Lauri Karttunen. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[15] Lauri Karttunen. The replace operator. In Emannual Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 117–147. Bradford, MIT Press, 1997.

[16] Lauri Karttunen. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara, 1998.

[17] George Anton Kiraz and Edmund Grimley-Evans. Multi-tape automata for speech and language systems: A prolog implementation. In Derick Wood and Sheng Yu, editors, *Automata Implementation. Second Internation Workshop on Implementing Automata, WIA '97*, pages 87–103. Springer Lecture Notes in Computer Science 1436, 1998.

[18] Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley. A rational design for a weighted finite-state transducer library. In *Automata Implementation. Second International Workshop on Implementing Automata, WIA '97*. Springer Verlag, 1998. Lecture Notes in Computer Science 1436.

[19] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[20] Alan Prince and Paul Smolensky. Optimality theory: Constraint interaction in generative grammar. Technical Report TR-2, Rutgers University Cognitive Science Center, New Brunswick, NJ, 1993. MIT Press, To Appear.

[21] D. Raymond and D. Wood. The grail papers. Technical Report TR-491, University of Western Ontario, Department of Computer Science, London Ontario, 1996.

[22] Emmanuel Roche. Parsing with finite-state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 241–281. MIT Press, Cambridge, 1997.

[23] Emmanuel Roche and Yves Schabes. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass, 1997.

[24] Gertjan van Noord. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*, pages 87–108. Springer Verlag, 1997. Lecture Notes in Computer Science 1260.

[25] Gertjan van Noord. FSA Utilities (version 5), 1998. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/˜vannoord/Fsa/.

[26] Gertjan van Noord. The treatment of epsilon moves in subset construction. In *Finite-state Methods in Natural Language Processing*, Ankara, 1998. cmp-lg/9804003. Accepted for *Computational Linguistics*.

[27] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.

A Syllabification in Optimality Theory

This is the implementation of Karttunen's formalization of syllabification in Optimality Theory.

```
%% Karttunen's X -> L ... R. Every X is ‘bracketed’ with L and R.
macro(dots(X,L,R), [[free(X), [[] x L, X, [] x R]]*, free(X)]).
%% Karttunen's A => L R. Every A must occur in context L _ R.
macro(restrict(A,L,R), ~[? *,A,~[R,? *]] & ~[~[? *,L],A,? *]).
macro(cons,{b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,z}).
macro(lbr,{'O[', 'D[', 'X[', 'N['}).
macro(input,{cons,vowel}*).
macro(parse,   dots(cons,{'O[','D[','X['},']')
            o dots(vowel,{'N[','X['},']')).
macro(overparse,[([] x [lbr,']'])^,dots({cons,vowel},[],[lbr,']']^)]).
macro(onset,['O[', cons^, ']']).
macro(nucleus,['N[', vowel^, ']']).
macro(coda,['D[', cons^, ']']).
macro(unparsed,['X[', {cons,vowel}, ']']).
macro(syllable_structure,ignore([onset^,nucleus,coda^],unparsed)* ).
macro(gen, input o overparse o parse o syllable_structure).
macro(have_ons,restrict('N[', onset, [])).
macro(nocoda,free('D[')).
%% 'parse' is used twice in Karttunen 98; we use parsed(N) where N is
%% the maximum number of occurrences of X
macro(parsed(N), free(N,'X[')).
macro(fillnuc, free(['N[', ']'])).
macro(fillons, free(['O[', ']'])).
:- op(403,yfx,lc).
macro(R lc C, lenient_composition(R,C)).
macro(syllabify,gen lc have_ons lc nocoda lc fillnuc lc parsed(0) lc
        parsed(1) lc parsed(2) lc parsed(3) lc parsed(4) lc fillons ).
```

B Mohri & Sproat Replace Operator

Implementation in FSA5 of the contexted replacement operator of [19].

```
macro(r(R),reverse(marker(1,[sigma*,reverse(R)],[>]))).
macro(f(F),reverse(marker(1,[{sigma,>}*,reverse([ignore(F,{>}),>])],
                          ['<1','<2']))).
macro(l1(L), sloppy_ignore(marker(2,[sigma*,L],'<1'),{'<2':'<2'})).
macro(l2(L),marker(3,[sigma*,L],'<2')).
```

```
macro(replace(Phi,Psi), {{sigma,'<2':'<2', > :[]},
               ['<1':'<1',ignore(Phi,{'<1','<2',> }) x Psi,> :[]]}*).
macro(sigma,? - {'<1','<2',>}).

rx(marker(Type,Expr,C),Fa) :-
    fsa_regex:rx(identity(determinize(Expr)),Fa0), mark(Type,C,Fa0,Fa).

mark(1,Ins,Fa0,Fa) :-  %% Ins: symbols to be inserted
    fsa_regex:add_symbols(Ins,Fa0,Fa1), fsa_data:symbols(Fa1,Sig),
    fsa_data:start_states(Fa1,Starts),  fsa_data:transitions(Fa1,Trs0),
    fsa_data:final_states(Fa1,Fins),    fsa_data:all_states(Fa1,AllSts),
    ordsets:ord_subtract(AllSts,Fins,NFins0),
    add_ins(Fins,Ins,NFins,NFins0,Trs,Trs1),
    replace_trs_sf(Trs0,Trs1,Fa0),
    fsa_data:rename_fa(Sig,Starts,NFins,Trs,[],Fa).

replace_trs_sf([],[],_).
replace_trs_sf([trans(A0,B,C)|T0],[trans(A,B,C)|T],Fa):-
    ( fsa_data:final_state(Fa,A0) -> A=q(A0) ; A=A0 ),
    replace_trs_sf(T0,T,Fa).

add_ins([],_,F,F) --> [].
add_ins([F0|Fs],Ins,[q(F0)|NewF0],NewF) -->
    add_ins0(Ins,F0), add_ins(Fs,Ins,NewF0,NewF).

add_ins0([],_F) --> [].
add_ins0([Sym|Syms],F) --> [trans(F,[]/Sym,q(F))], add_ins0(Syms,F).

mark(2,Del,Fa0,Fa) :-  %% Sym is a symbol to be deleted
    fsa_regex:add_symbols([Del],Fa0,Fa1),
    fsa_data:copy_fa_except(transitions,Fa1,Fa2,Trs0,Trs),
    fsa_data:copy_fa_except(final_states,Fa2,Fa,Fins,AllSts),
    fsa_data:all_states(Fa0,AllSts),
    add_deletions(Fins,Del,Trs1,Trs0),  sort(Trs1,Trs).

add_deletions([],_) --> [].
add_deletions([F|Fs],Del) --> [trans(F,Del/[],F)], add_deletions(Fs,Del).

mark(3,Del,Fa0,Fa) :- %% Del is a symbol to be deleted
    fsa_regex:add_symbols([Del],Fa0,Fa1),
    fsa_data:copy_fa_except(transitions,Fa1,Fa2,Trs0,Trs),
    fsa_data:copy_fa_except(final_states,Fa2,Fa,Fins,AllSts),
    fsa_data:all_states(Fa0,AllSts),
    ordsets:ord_subtract(AllSts,Fins,NonFins),
    add_deletions(NonFins,Del,Trs1,Trs0),  sort(Trs1,Trs).

%% As defined by Mohri & Sproat. This should be done differently,
%% ignore is not defined for transducers.
macro(sloppy_ignore(A,B),ignore0(A,B)).
```

C N-queens Problem

```
macro(free(Expr), ~containment(Expr)).
macro(sigma(N),set(L)):- findall(C,fsa_util:between(1,N,C),L).
macro(columns(N),Ints) :- columns(1,N,Ints).

%% don't use ordinary operator syntax, since this file is read-in with
%% regular expression operator precedences active.
columns(N,N,free([N,? *,N])).
columns(N0,N,free([N0,? *,N0]) & Ints) :-
    N0<N, is(N1,+(N0,1)), columns(N1,N,Ints).

macro(diagonals(N), I) :- diagonals(1,N,I).

diagonals(N0,N,I) :- is(N,N0+1),!, diagonals_n(1,N0,N,I).
diagonals(N0,N,I0 & I) :- diagonals_n(1,N0,N,I0),
    is(N1,+(N0,1)), diagonals(N1,N,I).

diagonals_n(N0,Br,N,I0) :- is(N,+(N0,Br)),!, diagonal(N0,Br,I0).
diagonals_n(N0,Br,N,I0 & I):-
    diagonal(N0,Br,I0), is(N1,+(N0,1)), diagonals_n(N1,Br,N,I).

diagonal(N0,Br,free([N0,length(MidN),N])) :-
    is(N,+(N0,Br)), is(MidN,-(Br,1)).
```