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
[2] University of Tübingen

June 23, 1999

Abstract

Finite-state techniques are widely used in various areas of Natural Language Processing (NLP). As Kaplan and Kay (1994) 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 more 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 (1994) 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 more 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 (1972) has shown that such rewrite rules are equivalent to finite state transducers under the standard assumption that they are not allowed to rewrite their own output. An algorithm for compilation into transducers was provided by Kaplan and Kay (1994). Improvements and extensions to this algorithm have been provided by Karttunen (1995) (1997) (1996) and Mohri & Sproat (1996). 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 (van Noord, 1998) is an extended, rewritten and redesigned version of the FSA Utilities toolbox previously presented at the first WIA(van Noord, 1997). FSA5

`[]`	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.

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

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)
```

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

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 (Mohri, Pereira, and Riley, 1998) 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 (Bouma, 1999). 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 (Kaplan and Kay, 1994) (p. 376) argue:

> 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.

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)
```

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([]),'{}').                                                     (8)
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 Gerdemann and van Noord (1999) to define a longest-match concatenation operator which implements the longest-capture semantics required by the POSIX standard.

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]) :-                                         (9)
    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)*  & length(N) & columns(N) &                 (10)
                   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

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

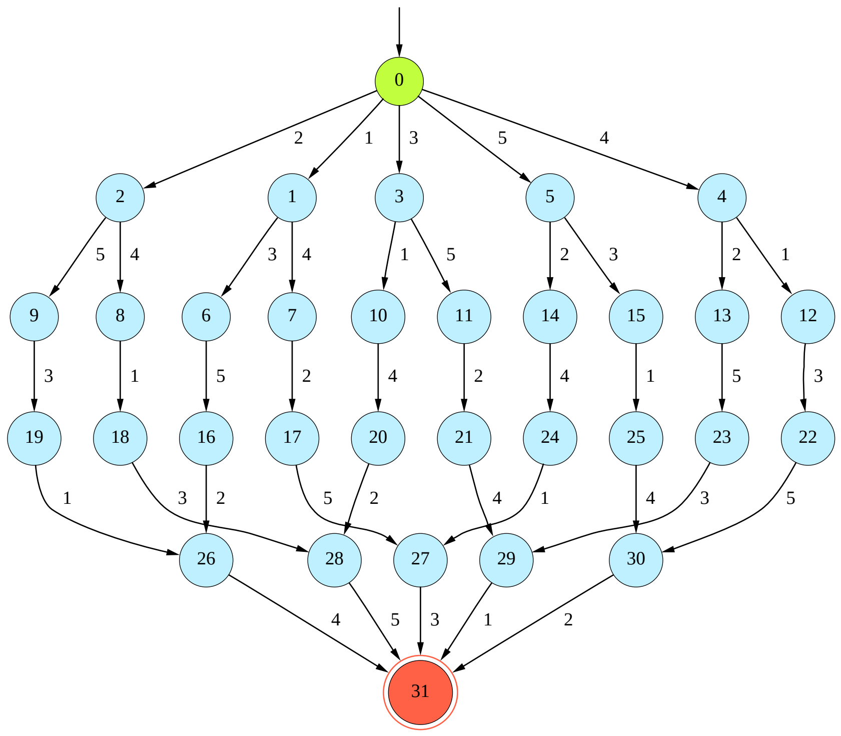


Figure 1: Solution to the 5-queens problem

defined previously.

```
macro(length(N),List) :- length(List,N), fill_qm(List).                    (11)

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.

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`. It is difficult to define this operator in terms of other operators. However, the operation is trivial 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) :-                                                   (12)
    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.

Lenient Composition. In a recent paper, Karttunen (1998) has provided a new formalization of Optimality Theory in terms of regular expressions. Optimality theory (Prince and Smolensky, 1993) 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 together with a set of ranked universal constraints which rule out many of the phonological representations generated by GEN. The constraints eliminate all but the best output. Some constraints can be conflicting. Therefore it might be impossible for a candidate string to satisfy all constraints. The best string is the string which violates the least important 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. It removes many of the candidate strings. The remaining strings are passed on to the next important constraint, and so on. 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. 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}).                          (13)
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]*)                    (14)
```

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.

The replace operator. In Mohri and Sproat (1996) a variant of the replace operator is implemented which is more efficient than previous implementations provided by Kaplan and Kay (1994) and Karttunen (1995). This improved version crucially depends on the possibility to manipulate 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),                                             (15)
    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 Mohri and Sproat (1996). 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,                                                        (16)
      replace('T':t,[{k,f,s,[c,h],p,t,x},+],e) o replace('T':d, +, [])
```

Left-most longest-match replacement with backreferencing. In Gerdemann and van Noord (1999) a left-most longest match replacement operator replace(T,Left,Right) is defined which ensures that the transducer T is applied in contexts Left and Right. One application of such an operator is finite-state parsing (chunking), (Abney, 1995; Chanod and Tapanainen, 1996; Grefenstette, 1996; Roche, 1997). 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 replace/1 is defined as:

```
macro( replace(T), replace(T,[],[])).                                   (17)
```

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 include the empty string):

```
macro((A --> Ds), [[] x [A,'['], Ds-[], []:']']).                    (18)
```

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, '[':'(', ~ $ ']',']':')'] ).   %$                (19)
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)+]                               (20)
```

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:

```
macro(mwu,replace({(prep --> [dw(prep,{'Ten',ten}),                  (21)
                              dw(n,opzichte),
                              dw(prep,van)]         ),
                   (prep --> [dw(prep,{'In',in}),
                              dw(n,verband),
                              dw(prep,met)]         ),
                   (prep --> [dw(prep,{'In',in}),
                              dw(n,plaats),
                              dw(prep,van)]         )
                   })).
```

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

```
macro(np_chunker, mwu                                                (22)
                  o
   replace(( adj --> [d(adv), d(adj)]))
                  o
   replace(( np -->  [d(art)^,d(num)^,d(adj)*,d(n)+]))
                  o
   replace(( pp -->  [d(prep),d(np)]))
                  o
   replace(( np -->  [d(np),d(pp)+]))
```

For example, one of the sentences from the Eindhoven corpus (den Boogaart, 1975) is chunked as in figure 2.

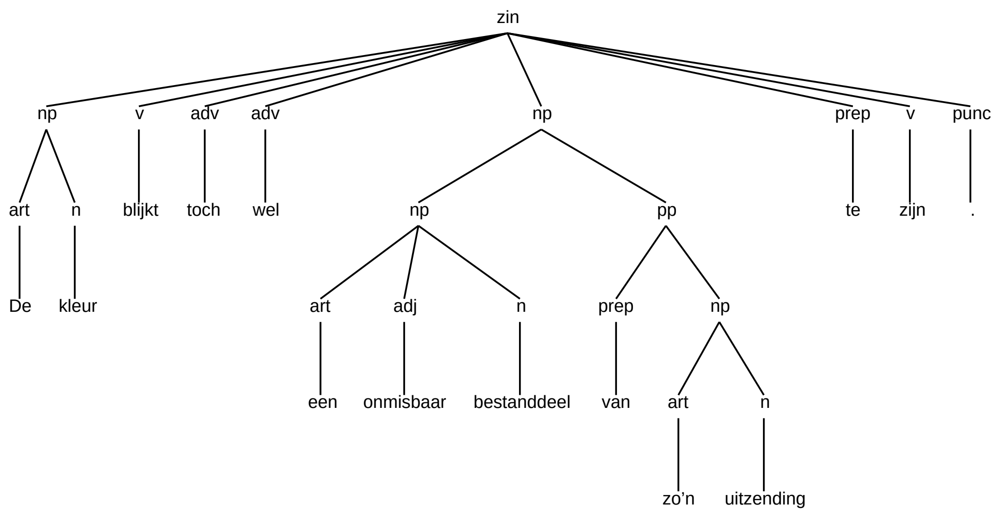


Figure 2: Application of NP chunker

5 Further Issues

5.1 Operators with Side-effects

The compiler supports a number of operators which have an effect on the underlying automata. 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 which alter the operation of the regular expression compiler. 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). The `cache(Expr)` operator can be used to cache the result of the compilation of regular expression `Expr`. For instance, in the replacement example taken from Mohri and Sproat (1996) the expressions `Left` and `Phi` occur twice: wrapping those expressions inside a `cache/1` operator implies that the expressions are compiled only once.

5.2 Alphabets

Many of the more complex examples of regular expression operators, such as the replacement operators, introduce special marker symbols into strings in order to mark off candidate regions for replacement. The assumption is that these marker symbols are outside the resulting transducer's alphabets. But the problem here is that previous versions of finite state calculus have not provided any way of specifying either the input or output alphabet, so there is no way of ensuring that the assumption holds.

This problem was at least recognized by Karttunen (1996), whose algorithm starts with a filter transducer which performs an identity transduction which filters out any string which happens to contain any of the special marker symbols. But this step is problematic for two reasons. First, when applied to a string that does happen to contain one of the marker symbols, the algorithm will simply fail. Second, the use of Karttunen's filter transducer leads to some serious inefficiency in the resulting transducer, which will be cluttered with arcs over the special marker symbols leading to a sink state.

As a step in the direction of solving this problem [3] it is possible in FSA5 to specify precisely what the input and output alphabets are. In particular, it is possible to define an identity trans-

[3] A complete solution to the problem is presented in Gerdemann & van Noord (1999).

ducer, whose sole purpose is to transduce strings over a given alphabet to strings over the same alphabet extended with some number of marker symbols. A flexible approach such as this appears to be necessary to implement any algorithm in the tradition of Kaplan & Kay. For example, the code of Kaplan & Kay contains several lines of the general form:

$$Ignore(m, Macro(Expr)) \tag{23}$$

This is interpreted as the language described by $Macro(Expr)$ with instances of the marker m freely interspersed. The $Macro$ here may be defined in terms of any operations in the finite state calculus. But if $Macro$ is defined in terms of complement, which in turn is defined as $Compl(Expr) = \Sigma^* - Expr$, then one is required to know what Σ is within the context $Macro(Expr)$. Does Σ contain m or not? In FSA5, one can be precise about such issues.

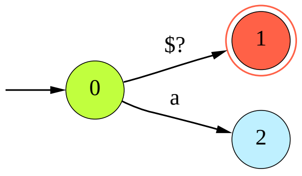
5.3 Implementation

The regular expression compiler is defined in SICStus Prolog. 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 this parser. The constructed term is then straightforwardly compiled into a corresponding finite automaton using a simple top-down recursive procedure.

Note that 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. One 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.

5.4 Questions concerning question mark

The possibility to use the question mark meta-symbol to refer to any symbol is extremely useful in order to define general regular expression operators. For instance, the `containment` operator defined above could not be defined without it. In order to be able to support this meta symbol, the automata constructed by the regular expression compiler support a related symbol, written `$?`, with a slightly different meaning: all possible symbols, except those symbols seen during the construction of the automaton. Thus, each automaton is associated with a list of symbols which are used to set the meaning of the meta-symbol `$?`. The reason for this technique, which is apparently similar to the technique used in the Xerox regular expression compiler, is that we can now construct an automaton for expressions such as `?-a`:



(24)

If automata are combined using any of the regular expression operators, then each automaton is first expanded in order to take into account the symbols used in any of the other automata. Thus, in the expression:

```
(? - a) & (? - b)
```
(25)

the automaton for `?-a` is first expanded into an automaton for {`?,b`}`-a`.

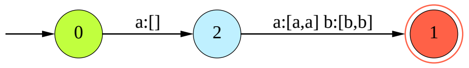
The advantage of this technique is that the actual implementation of the intersection operator can treat `$?` as an ordinary symbol. The disadvantage, however, is that in some cases this leads to automata with many (seemingly redundant) transitions.

A potential solution to this redundancy problem is to use a technique due to Watson (1996) in which a label is considered to be a predicate, taking a single argument (the input symbol). A transition is taken if the predicate returns true for the actual input symbol.

Another more fundamental problem (for which this suggestion does not seem to help) arises if the `identity` operator is applied to `?`. Obviously, the identity of `?` should be a transducer that maps any input symbol to *the same* output symbol. Note that this is different from `?:?` which is a transducer which maps any input symbol to any output symbol. For this reason, another special symbol pair is introduced in the representation for transducers: `$@:$@` is a transition which reads any symbol not in the current list of symbols and outputs the very same symbol. FSA5 supports these special symbol pairs. So far so good. But now we apply the determinization algorithm for transducers (Mohri, 1996)(Roche and Schabes, 1995). This algorithm essentially 'delays' output symbols until a deterministic choice is possible. For instance, consider the expression:

```
{[a:b,b],[a:a,a]}
```
(26)

This maps a sequence `ab` to a `bb`, and a sequence `aa` to `aa`. A 'deterministic' transducer for this example is:



(27)

Now consider the case in which two question marks are placed between the first and second symbol, i.e.:

```
{[a:b,? ,? ,b],[a:a,? ,? ,a]}
```
(28)

The resulting automaton, given in figure 3, maps sequences `aXXa` to `aXXa` and sequences `aXXb` to `bXXb`. The automaton illustrates the size increase and redundancy referred to above, but it also illustrates another problem: the output `$@` symbols are pushed into later transitions.

Such transducers are interpreted as follows (this idea is due to Lauri Karttunen, p.c.). A queue is maintained which keeps track of input symbols matched with `$@`. If an `$@` has to be output, it is dequeued. This mechanism is supported in the FSA5 interpreter and compiler. Automata containing such delayed `$@` outputs are not supported, however, by any of the regular expression operators.

Concluding Remarks

We have discussed 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

Abney, Steven. 1995. 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.

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

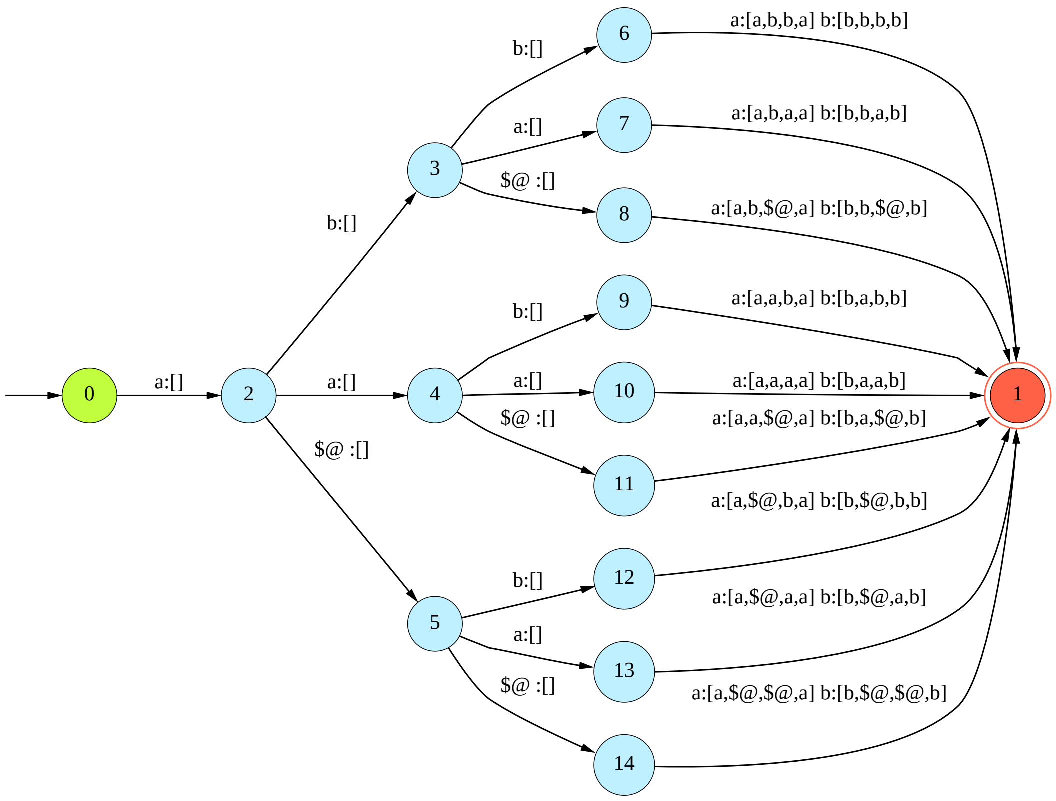


Figure 3: Delaying unknown output symbols.

Bouma, Gosse. 1999. 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.

Chanod, Jean-Pierre and Pasi Tapanainen. 1996. A robust finite-state grammar for French. In John Carroll, editor, *Workshop on Robust Parsing*, Prague.

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

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

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

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

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

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

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

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

Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2:61–80. Originally appeared in 1994 as Technical Report, institut Gaspard Monge, Paris.

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

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

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

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

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

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

Roche, Emmanuel and Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2).

Watson, Bruce. 1996. Implementing and using finite automata toolkits. In *Extended Finite State Models of Language*, Proceedings of the ECAI'96 workshop, pages 97–100, Budapest University of Economic Sciences, Hungary.

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(rbr,']').
macro(input,{cons,vowel}*).
macro(parse, dots(cons,{'O[','D[','X['},']') o dots(vowel,{'N[','X['},']')).
macro(overparse,[([] x [lbr,rbr])^,dots({cons,vowel},[],[lbr,rbr]^)]).
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 Mohri and Sproat (1996).

```
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)).
```