Projektseminar ‘Sprachtechnologie’ (Sommersemester 1998)

An Implementation of ‘Finite-State Methods in NLP: Algorithms’

Ralph Debusmann
Universität des Saarlandes
Computerlinguistik
rade@coli.uni-sb.de

September 19, 2002

Contents

1. Introduction

This paper describes ‘Fistame’, a partial implementation of [KK], written using the Prolog programming language. Fistame provides a framework for easy encoding of and working with finite-state automata.

The paper is divided into six sections (plus an appendix). Following this brief overview, Section 2 is made up of an informal introduction to finite-state automata (henceforth ‘FSAs’) in general, followed by a description of how to encode and operate upon them within Fistame in particular. Section 3 hereafter serves to expose the various operations Fistame provides, before Section 4 goes into more detail with revealing their concrete implementation. Section 5 is then solely dedicated to a rather advanced implemented algorithm (‘minimize’). Section 6 settles the paper with some concluding remarks.

2. Finite-state automata

This section serves to briefly introduce finite-state automata in general and shows how to encode and operate upon them within the Fistame framework. Its starting point is an informal exposition of some basic knowledge about finite-state automata.

2.1. Basics

Finite-state automata are abstract devices which can be applied to recognize (i.e. accept or reject) strings of a regular language. FSAs encode regular languages using the two atomic concepts ‘state’ and ‘transition over a symbol’.

In the process of recognizing a string, it is scanned from left to right, symbol by symbol. At any point during the scan, a string symbol passed by must correspond to a transition over that symbol in the recognizing FSA if the string is to be accepted. If more than one such transition (over the same symbol) is possible from (at least) one state in the FSA, it is called ‘non-deterministic’, otherwise it is said to be ‘deterministic’.

Whereas the starting point for scanning a string is its leftmost symbol, FSAs must have a certain state explicitly marked as the ‘start state’[1]. In addition, at least one state must be labelled as ‘final state’. Given these prerequisites, a string is accepted by an FSA A if it has been scanned until its end and A is in a final state. Otherwise the string is rejected.

2.2. An example

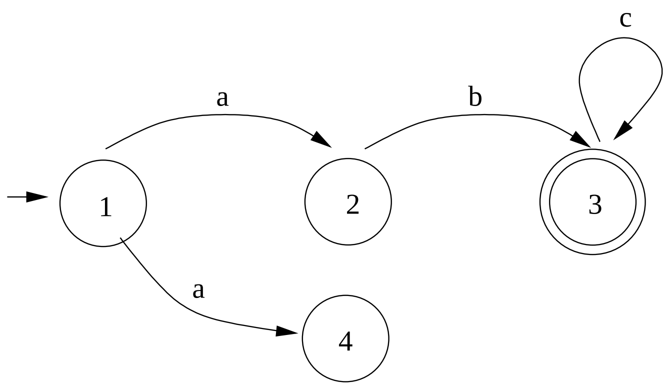

Figure 1: $A1$. Numbered circles (aka vertices, nodes) represent states, arrows (aka edges) transitions.

The FSA $A1$ accepts strings of the regular language $L1 = abc^*$, i.e. *ab*, *abc*, *abcc* etc. The number of *c*s at the end of a string in $L1$ may be unbounded (Kleene star). $A1$

[1]Fistame supports only one start state per FSA, as opposed to the widely used notion of allowing FSAs to bear more than one state of this kind. However, as a workaround, the `det`-operation allows to conflate multiple start states into one.

is composed of four states: $S1 = \{1, 2, 3, 4\}$. State 1 is the start state (represented in Figure 1 by an arrow entering the circle representing it) and state 3 the only final state (Figure 1: double concentric circle). $A1$ bears four transitions, the first from (state) 1 over (symbol) a to (state) 2, the second from 1 over a to 4, the third from 2 over b to 3 and the fourth from 3 over c back to 3. The latter 'implements' the iteration needed to account for the unbounded number of strings which abc^* denotes. The signature of $A1$ is $\Sigma = \{a, b, c\}$, made up of all the symbols over which at least one transition takes place.

Note that state 4 has only been incorporated into $A1$ for explanatory purposes in the later course of this section (Section 2.5). It could also have been dropped and still $A1$ would recognize the same language (abc^*).

2.3. Encoding finite-state automata

Encoding FSAs for use within the Fistame framework is quite straightforward. This is $A1$ in Fistame notation:

```
start(a1,1).
trans(a1,1,a,2).
trans(a1,1,a,4).
trans(a1,2,b,3).
trans(a1,3,c,3).
final(a1,3).
```

First of all, in Fistame every FSA needs to have a unique (lowercase) name, in this case `a1`. By `start(fsa,1)`, Fistame is told that the start state of `a1` is state 1 and `final(a1,3)` tells it that state 3 is a final state. Transitions are encoded via the `trans`-predicate. For instance, `trans(a1,1,a,2)` means that `a1` bears a transition from state 1 over a to state 2. Note that unmarked states (those which are neither start nor final states) do not have to be spelt out explicitly. Also note that the order of appearance of the `start`-, `trans`- and `final`-predicates is arbitrary.

2.4. Operating on encoded automata

Properly encoded automata may now be operated upon by any of Fistame's built-in operations[2]. The simplest and most common operation is to print out an FSA via `cat`:

```
1 ?- scan(cat(a1)).

automaton a1
start state 1
from 1 over a to 2
from 1 over a to 4
```

[2]Of course, Fistame should have been started in the first place. See Appendix A on how to choose a suitable startup file for your Prolog interpreter.

```
state 2
from 2 over b to 3
final state 3
from 3 over c to 3
state 4

Yes
```

Like any other operation on automata Fistame provides, `cat` must be headed by the `scan`-predicate, the backbone for all operations. A detailed explanation of `scan` is given in Section 4. Supplementing unary operations like `cat`, Fistame also provides binary operations (copy, intersection and union).

2.5. Nesting operations

Fistame operations may be combined or 'nested':

```
2 ?- scan(cat(rev(a1))).

automaton rev(a1)
start state s0
state s0
from s0 over ep to 3
state 3
from 3 over b to 2
from 3 over c to 3
state 2
from 2 over a to 1
final state 1

Yes
```

Here, `a1` is reversed by the `rev`-operation before it is printed out. For reasons explained in Section 3, `rev` adds a new state s0 and an 'epsilon transition'[3] from this state to the final state (state 3) of `a1`.

You might have noticed another peculiarity of the reversed automaton: It lacks state 4 from the 'unreversed' `a1`. This is because state 4 is a 'dead state' in `a1`—it is non-final and there are no transitions from it to any other state. In other words, since no transition 'went out' from it, after all transitions of `a1` have been reversed, none will find its way back 'into' it.

[3]The 'epsilon' (ϵ) is a special symbol denoting the empty word (encoded as `ep` in Fistame). FSAs without ϵ-transitions are called 'ϵ-free'.

2.6. Support predicates

Fistame includes a couple of ‘support predicates’, which differ in some respects from its ‘operations’. Above all, support predicates may not be nested like the latter (because they are not utilizing the `scan`-predicate as their backbone). For instance, the `accept`-predicate checks whether an FSA accepts a given string or not:

```
3 ?- accept(a1,abccc).

Yes
4 ?- accept(a1,aabccc).

No
```

`remove` removes a given FSA from the Prolog database:

```
5 ?- remove(a1).

Yes
6 ?- scan(cat(a1)).

No
```

Finally, the `reset`-predicate removes all automata from the Prolog database and then reloads the Fistame example automata from ‘examples.pl’:

```
7 ?- reset.

operations compiled, 0.14 sec, 8,012 bytes.
examples compiled, 0.10 sec, 5,424 bytes.

Yes
```

3. Operations on finite-state automata

After a rather short exposition of finite-state automata and about how the Fistame implementation deals with them in the preceding section, the upcoming section will concentrate on exhibiting the full range of operations on FSAs available. Examples will be used throughout to visualize the operations’ effects. We begin by fleshing out the unary operations provided, followed by an explanation of the binary ones.

3.1. Unary operations

Two unary operations have already been discussed in Section 2, viz. `cat` (print an automaton) and `rev` (reverse an automaton). This subsection concentrates on explaining these and the remaining unary operations (taking only one argument). Their order is alphabetical.

3.1.1. cat

The `cat`-operation prints out an automaton:

```
1 ?- scan(cat(a1)).

automaton a1
start state 1
from 1 over a to 2
from 1 over a to 4
state 2
from 2 over b to 3
final state 3
from 3 over c to 3
state 4

Yes
```

3.1.2. complement

`complement` is a nested operation made up of three operations (in this order: `complete`, `det` (determinize) and `inv` (inverse)). It generates out of an FSA the complement automaton, which accepts the complement regular language:

```
1 ?- scan(cp(complement(a1),a1c)).

Yes
2 ?- accept(a1c,abcc).

No
3 ?- accept(a1c,abcca).

Yes
```

In the above example, `a1c` becomes the complement automaton of `a1` by means of the `cp`-operation (step 1), i.e. `a1c` accepts only strings which are not in the language denoted by `a1`. This is checked for two strings (*abcc* and *abcca*) with the `accept`-support predicate in steps 2-3.

3.1.3. complete

The `complete`-operation adds to an ‘incomplete’ FSA formerly implicit transitions to a dead state, thereby ‘completing’ it: Every state of a complete automaton must be the starting point for transitions over all symbols from its signature:

```
1 ?- scan(cat(complete(a1))).

automaton complete(a1)
start state 1
from 1 over a to 2
from 1 over a to 4
from 1 over b to dead
from 1 over c to dead
state 2
from 2 over b to 3
from 2 over a to dead
from 2 over c to dead
final state 3
from 3 over c to 3
from 3 over a to dead
from 3 over b to dead
state dead
from dead over a to dead
from dead over b to dead
from dead over c to dead
state 4
from 4 over a to dead
from 4 over b to dead
from 4 over c to dead

Yes
```

3.1.4. cp and cp_rep

`cp` and `cp_rep` can be used to 'carbon copy' one FSA onto a new one. The two operations described here are unary (taking only one argument), specialized counterparts of those explained in 3.2.1.

```
1 ?- scan(cp(rev(a1))).

Yes
2 ?- scan(cat(copy)).

automaton copy
start state s0
from s0 over ep to 3
state 3
from 3 over b to 2
from 3 over c to 3
```

```
state 2
from 2 over a to 1
final state 1

Yes
```

The reverse FSA of `a1` is here copied onto a new FSA with the name `copy`. Note that if an FSA called `copy` already had existed, the `cp`-operation would have overwritten it without giving notice.

`cp_rep` performs the same operation as `cp`: It also copies an FSA onto a new one. In addition, it removes all ϵ-transitions of the former:

```
3 ?- scan(cp_rep(rev(a1))).

Yes
4 ?- scan(cat(copy)).

automaton copy
start state s0
from s0 over b to 2
from s0 over c to 3
state 2
from 2 over a to 1
final state 1
state 3
from 3 over b to 2
from 3 over c to 3

Yes
```

3.1.5. det and det_rep

The `det`- and `det_rep`-operations can be used to create a deterministic FSA out of a non-deterministic one. It also allows to conflate multiple start states into one. For explanatory reasons, we will now introduce the new FSA `a2`:

```
1 ?- scan(cat(a2)).

automaton a2
start state 1
from 1 over a to 2
from 1 over a to 4
state 2
from 2 over b to 3
final state 3
```

```
from 3 over c to 3
state 4
from 4 over ep to 2

Yes
```

a2 is nearly equal to a1, the only difference being a new transition from state 4 over ϵ to state 2. Still it accepts the same regular language abc^*, and is non-deterministic, as is a1: There are two transitions originating from state 1 over the symbol a (one to state 2 and one to state 4).

```
2 ?- scan(cat(det(a2))).

automaton det(a2)
start state [1]
from [1] over a to [2, 4]
state [2, 4]
from [2, 4] over b to [3]
from [2, 4] over ep to [2]
state [2]
from [2] over b to [3]
final state [3]
from [3] over c to [3]

Yes
```

After applying the det-operation on a2, the two transitions transitions 1 over a to 2 and 1 over a to 4 have been merged into a single transition ([1] over a to [2, 4]), resulting in a deterministic FSA.

The det_rep-operation also makes a given FSA deterministic. In addition, it removes all epsilon transitions from the result:

```
3 ?- scan(cat(det_rep(a2))).

automaton det_rep(a2)
start state [1]
from [1] over a to [2, 4]
state [2, 4]
from [2, 4] over b to [3]
final state [3]
from [3] over c to [3]

Yes
```

After applying det_rep, not only has a2 been determinized but also has the ϵ-transition from state [2, 4] to state [2] been removed (as well as state [2] itself).

3.1.6. inv

The `inv`-operation ‘inverts’ an automaton, i.e. all non-final states become final states and vice-versa:

```
1 ?- scan(cat(inv(a1))).

automaton inv(a1)
start state 1
final state 1
from 1 over a to 2
from 1 over a to 4
final state 2
from 2 over b to 3
state 3
from 3 over c to 3
final state 4

Yes
```

In this example, `a1` is inversed, resulting in all the non-final states 1, 2 and 4 becoming final states and state 3 losing this status.

3.1.7. mini

`mini` is one rather advanced Fistame-operation. It ‘minimizes’ an FSA, such that the outcome is the ‘minimal’ automaton (using a minimal number of states) recognizing exactly the same regular language as the original one. The algorithm conflates any two states which bear an equivalent ‘suffix set’[4] into one single state. The algorithm used for Fistame is a slightly abridged version of an algorithm by [Hop71], further expatiated upon in Section 5.

Let us introduce another FSA (`a3`) for explanatory purposes:

```
1 ?- scan(cat(a3)).

automaton a3
start state 1
from 1 over a to 2
from 1 over b to 3
state 2
from 2 over a to 4
final state 4
state 3
```

[4]We (quite informally) define the ‘suffix set’ of a state as the set containing all strings recognizable if you used that state as the start state.

```
from 3 over a to 4

Yes
```

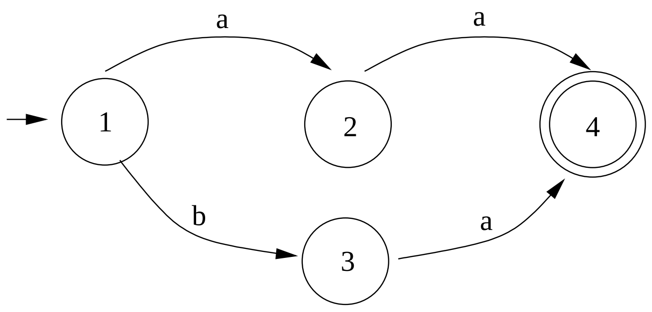

Figure 2: a3 graphically.

This finite-state automaton is now to be minimized. It includes exactly two states (2 and 3) which have the same suffix set $S = \{a\}$: Both states bear one transition over a to the final state 4 and are therefore conflated during the minimizing process. The results (graphical and textual) are:

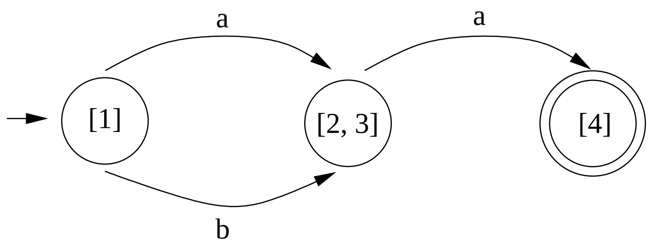

Figure 3: a3 minimized.

```
2 ?- scan(cat(mini(a3))).

automaton mini(a3)
start state [1]
from [1] over a to [2, 3]
from [1] over b to [2, 3]
state [2, 3]
from [2, 3] over a to [4]
final state [4]

Yes
```

Note that the rather verbose output of the underlying algorithm has been left out in the textual depiction above. Also, two more complex example FSAs for the `mini`-operation can be found as `a4` and `a5` in ‘examples.pl’.

3.1.8. rev

As already discussed in Section 2, `rev` reverses an FSA:

```
1 ?- scan(cat(rev(a1))).

automaton rev(a1)
start state s0
state s0
from s0 over ep to 3
state 3
from 3 over b to 2
from 3 over c to 3
state 2
from 2 over a to 1
final state 1

Yes
```

Like in the example above, the `rev`-operation always introduces a new state (here: `s0`) and ϵ-transitions from this state to all final states of the FSA to be reversed. This ‘trick’ is employed because Fistame supports only one start state per FSA (as already noted). It ensures that all final states of the original FSA act as start states in the reversed FSA.

3.1.9. sig

The `sig`-operation asserts the signature of an automaton into the Prolog database. Applied on `a1` for example, the Prolog-predicates `symbol(a1,a)`, `symbol(a1,b)` and `symbol(a1,c)` are asserted:

```
1 ?- scan(sig(a1)).

Yes
2 ?- listing(symbol).

symbol(a1, a).
symbol(a1, b).
symbol(a1, c).

Yes
```

`sig` is no clear-cut 'operation', because it is not of any use for other operations put on top of it. Nevertheless, it is useful for the formulation of several of Fistame's operations. For instance, it is employed within the `mini`-operation (Section 5).

3.1.10. string

With the `string`-operation you are able to generate an 'artificial' FSA out of a given string, which accepts only that string and no other:

```
1 ?- scan(cat(string(abc))).

automaton string(abc)
start state [a, b, c]
from [a, b, c] over a to [b, c]
state [b, c]
from [b, c] over b to [c]
state [c]
from [c] over c to []
final state []

Yes
```

3.2. Binary operations

In addition to its unary operations, Fistame also implements four binary operations, which are addressed in the following subsection.

3.2.1. cp and cp_rep

`cp` and `cp_rep` can be used to 'carbon copy' one FSA onto a new one. The two operations described here are binary, generalized counterparts of those expounded in 3.1.4.

```
1 ?- scan(cp(rev(a1),a42)).

Yes
2 ?- scan(cat(a42)).

automaton a42
start state s0
from s0 over ep to 3
state 3
from 3 over b to 2
from 3 over c to 3
state 2
from 2 over a to 1
```

```
final state 1

Yes
```

The reverse FSA of `a1` is here copied onto a new FSA with the name `a42`. Note that if an FSA called `a42` already had existed, the `cp`-operation would have overwritten it.

`cp_rep` performs the same operation as `cp`: It also copies an FSA onto a new one. In addition, it removes all ϵ-transitions of the former:

```
3 ?- scan(cp_rep(rev(a1),a4711)).

Yes
4 ?- scan(cat(a4711)).

automaton a4711
start state s1
from s1 over b to 2
from s1 over c to 3
state 2
from 2 over a to 1
final state 1
state 3
from 3 over b to 2
from 3 over c to 3

Yes
```

3.2.2. , (intersection)

The comma `,` stands for the operation of intersecting two finite-state automata. In the example below, the automata `a1` and the `a6` are to be intersected. `a6` is an FSA recognizing only two strings, that is to say *ab* and *ba* (or more formally, it recognizes the regular language $ab \cup ba$):

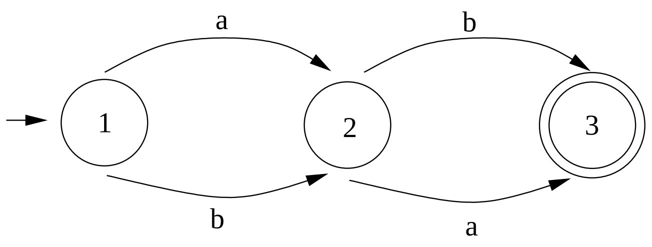

Figure 4: FSA 6 (a6) graphically.

```
1 ?- scan(cat(a6)).

automaton a6
start state 1
from 1 over a to 2
from 1 over b to 2
state 2
from 2 over b to 3
from 2 over a to 3
final state 3

Yes
```

Intersected with `a1` (recognizing abc^*), the resulting FSA `a2001` only recognizes the string ab, and no other string:

```
2 ?- scan(cp((a1,a6),a2001)).

Yes
3 ?- accept(a2001,abccc).

No
4 ?- accept(a2001,ab).

Yes
```

3.2.3. ; (union)

The `;`-operation models the counterpart of intersection: the operation of union. The union of `a1` and `a6` for instance is an FSA recognizing the regular language $abc^* \cup ba$ (i.e. both strings of the form abc^* and the strings ab and ba):

```
1 ?- scan(cp((a1;a6),a192)).

Yes
2 ?- accept(a192,abccc).

Yes
3 ?- accept(a192,ba).

Yes
```

4. Implementation

After the brief listing and explanation of the available operations which made up the previous section, this one is concerned with revealing their implementation.[5]

4.1. scan-predicate

The `scan`-predicate is the backbone for all operations Fistame offers and therefore warrants a detailed explanation. This will be done by elaborating upon the Prolog code of the `scan`-predicate ('scan.pl') in a step by step fashion.

The first four lines to be discussed make up the toplevel predicate `scan`:

```
1  scan(A):-
2    retractall(state(A,_)),
3    start(A,S),
4    scan_state(A,S).
```

Line 1 is filled by the head of the unary predicate `scan(A)` — its single argument being the FSA `A` to scan. Line 2 then provides for the removal of all predicates with the pattern `state(A,_)`, which are to be used later to take notice of already scanned states. The code in line 3 unifies the variable `S` with the start state of the FSA `A`, and line 4 initiates the scanning process by calling the `scan_state`-predicate.

`scan_state` habors the most important part of the code in 'scan.pl':

```
5  scan_state(A,S):-
6    state(A,S),!.
7  scan_state(A,S):-
8    assert(state(A,S)),
9    do_process_state(A,S),
10   (  setof(D,X^trans(A,S,X,D),Ds)
11   -> scan_states(A,Ds)
12   ;  true
13   ).
```

Let us first concentrate on the main clause of the predicate, i.e. lines 7-13. After unifying the two arguments `A` and `S` with the to-be-scanned FSA and the to-be-scanned state respectively, line 8 uses the Prolog built-in `assert`-predicate to mark state `S` as already scanned. Line 9 then calls the `do_process_state`-predicate:

```
14 do_process_state(A,S):-
15   final(A,S).
16 do_process_state(A,S):-
```

[5]Note that this and the next section assume some familiarity with the Prolog programming language. If you only want to 'consume' Fistame and have no ambition whatsoever to understand its inner workings or to enhance it, it is advisable to skip them.

```
17   process_state(A,S).
18 do_process_state(_,_).
```

`do_process_state` is only used by the `cat`-operation and thus does not deserve a very deep exposition. In short, it can be used to gain access to ‘ordinary’ states (neither ‘start states’ nor ‘final states’).

Let us return to the `scan_state`-predicate (lines 10-13). These lines are the core of the `scan`-predicate as a whole. They are built according to a special Prolog construction similar to ‘if-then-else’-constructions in imperative languages likes ‘Basic’. The first part (the ‘if’) is line 10, line 11 is the ‘then’-part and line 12 the ‘else’-part. In the ‘if’-part, the `setof`-predicate is applied, looking for all states `D` which are the destination of a transition originating from state `S` over any symbol `X`. The result of this operation, i.e. all states reachable from `S`, is a list bound to the variable `Ds` (‘destinations’). This list is then further processed by the `scan_states`-predicate[6]:

```
19 scan_states(A,[S|Ss]):-
20   scan_state(A,S),
21   scan_states(A,Ss).
22 scan_states(_,[]).
```

`scan_states` (not to confuse with the `scan_state`-predicate) traverses this list of ‘destinations’. In so doing, it calls `scan_state` on every destination state in the list until the end of the list is attained.

To finalize this description of the inner workings of the `scan`-predicate, let us look at lines 5-6. The code contained herein represents the ‘termination condition’ of the `scan_state`-predicate. It ensures that the predicate (and with it the entire ‘scan’-process) does not end up in an infinite loop (in the case of self-referential states) by checking whether the state to be scanned next has already been scanned. If this is the case, lines 5-6 prevent the execution of lines 7-13 by utilizing the Prolog-‘Cut’-operator.

4.2. A speciman operation

To understand the inner workings of Fistame entirely, you should also be given an explanation of how operations are defined on top of the `scan`-predicate. Therefore, this section deals with a simple ‘speciman operation’, namely (`det`) for ‘determinization’. Its Prolog code (taken from ‘operations.pl’) looks like this:

```
1  start(det(A),Ss):-
2    setof(X,start(A,X),Ss).

3  trans(det(A),Ss,X,Ds):-
4    setof(D,S^(member(S,Ss),trans(A,S,X,D)),Ds).
```

[6]If `setof` could not find any ‘destinations’, the ‘else’-part (line 12) ensures that the `scan_state`-predicate returns ‘true’ nevertheless, to keep the ‘scan’-process going.

```
5  final(det(A),Ss):-
6    member(S,Ss),
7    final(A,S).
```

The `det`-operations consists of redefinitions of three predicates, namely `start/2`, `trans/4` and `final/2`. Hence, let us plunge into separate explanations of these.

4.2.1. Redefinition of start/2

The redefinition of `start/2` takes place in line 1-2. An intuitive verbalization of this code would be 'the start state of the determinized version of FSA `A` is a conflation of all start states of `A` into a single one `Ss`'. This idea is expressed using the `setof`-predicate, resulting in a list of all start states of `A` in the 'state list' `Ss`. If an FSA had start states 1 and 2 for instance, `det(A)` would have the single start state `[1, 2]`.

4.2.2. Redefinition of trans/4

This redefinition (lines 3-4) bears a strong resemblance to the one above (`start/2`). Again the `setof`-predicate is employed, this time to fuse all transitions over any symbol `X` in FSA `A` starting from state `S` (which has to be in the list of states denoted by the 'state list' `Ss` of `det(A)`) and going to state `D` into the destination 'state list' `Ds`. Thence, if FSA `A` possessed two transitions, one from 1 over a to 2 and the other one from 1 over a to 3, these would be fused into a single transition [1] over a to [2, 3].

4.2.3. Redefinition of final/2

The final redefinition (lines 5-7) asserts that a state (i.e. state list) in the determinized FSA is a final state if one of the members of its state list bore the property of being a final state. Therefore, if `A` had the final state 1, all states resembling state lists like `[1, 2]` or `[1, 2, 3]` etc. would be final states in `det(A)`.

5. Minimize operation

Jutting out from the other operations is the `mini`-operation used to minimize the number of states in an finite-state automaton, the underlying algorithm being an abridged version of [Hop71]. This standing out not only results from it being the most comprehensive of all operations from 'operations.pl', but also because it is the only one to require a separate source code file ('minimize.pl'). This section is intended to clarify the algorithm and its actual implemention for Fistame.

The `mini`-operation in Fistame is split into two main parts—the first being the algorithm itself (in the file 'minimize.pl'), the second being the construction of a minimized automaton out of the upshot of the former ('operations.pl'). They are to be elaborated upon in this order.

5.1. Algorithm part

The algorithm used for minimizing an FSA in Fistame is an abridged version of an algorithm by [Hop71]. Its most salient feature is polynomial computation time, as opposed to earlier algorithms which required an exponential amount of time. To understand the principles behind it, we will expatiate upon the upper levels of its Fistame Prolog implementation, similar to my elaboration upon the `scan`-predicate above.

5.1.1. minimize-predicate

The toplevel predicate in 'minimize.pl' is `minimize`, and it also is the starting point in elaborating upon the entire algorithm:

```
1  minimize(M):-
2    scan(sig(M)),
3    setof(S,state(sig(M),S),Ss),
4    setof(X,symbol(M,X),Xs),
5    split(M,Ss,final,SC,BC),
6    cleara(M),
7    cleari(M),
8    pusha(M,SC),
9    pushi(M,BC),
10   assert(pseudo(M,BC)),
11   minimize1(M,Xs),
12   geti(M,IC),
13   format('~nInactive = ~w~n',[IC]).
```

The one argument `minimize` takes is the FSA `M` (line 1). In line 2, the `sig`-operation is used to put (assert) into the Prolog database all symbols of `M`. As a side effect, the `scan`-predicate also asserts all states of `M`: As they are marked as being visited, the predicate `state(sig(M),S)` is asserted for all states of FSA `M`. Now in lines 3-4, the set of all states of `M` is bound to the variable `Ss`, and the set of all symbols (i.e. the signature) of `M` is bound to `Xs`.

Line 5 marks the first 'split'-process. The list of all states of `M`, `Ss`, is here split using the `final`-predicate as the 'splitting criterion'. The result of `split` are two lists of states (called 'classes' or 'partitions'), bound to the variables `SC` and `BC`. `SC` is bound to the smaller resulting class, `BC` to the bigger. 'splitting' or 'partitioning' is the main ingredient in the process of minimizing an FSA—classes comprising states are split until a minimum number of these classes is attained.

Travelling along, lines 6-10 harbor a couple of further initialization steps. They concern the two 'global' lists which are to be used extensively later in the algorithm, namely the 'active list' and the 'inactive list'. The active list will carry state classes which are to be used 'actively' for further split-operations, whereas the inactive list carries classes which are not used actively for splitting purposes any longer. `cleara` erases any formerly existing active list (line 6) and `cleari` does the same for any existing inactive list (both for the automaton M). After that, the smaller class from the split above (SC) is put ('pushed') on the active, the bigger class (BC) on the inactive list.

Line 10 then copes with a small clumsiness of the algorithm: Since the first split (line 5) was different from succeeding ones in employing the final-predicate (instead of marked, see below), the bigger resulting class of this split (BC) is marked as 'pseudoactive'. This 'quirk' is not very important for seeing the whole picture, but it has to be dealt with to retain soundness.

Finishing up, line 11 calls the minimize1-predicate to initiate the recursive process of minimization after the above initialization steps (Lines 12-13 below just fetch the resulting list of inactive classes from minimize1 and print it out).

5.1.2. minimize1-predicate

The minimization algorithm implemented in Fistame contains two nested recursive loops, the first of which is minimize1 and the second minimize2. Let us focus on minimize1 first:

```
14 minimize1(M,Xs):-
15   (  popa(M,AC)
16   -> format('Minimize using ~w~n',[AC]),
17      pushi(M,AC),
18      minimize2(M,AC,Xs),
19      minimize1(M,Xs)
20   ;  true
21   ).
```

This predicate operates similar to a 'WHILE DO'-loop in some rather old-fashioned languages. Its input arguments are the FSA to be minimized (M) and its signature (Xs). Now WHILE there is at least one active class AC on the list of active classes (line 15), it uses this class to continue the minimization process. Line 17 then pushes AC on the list of inactive classes, before minimize2 is invoked (line 18). Subsequently, line 19 recursively calls minimize1 again and thus completes the 'WHILE DO' loop, enacting the role of the 'DO'.

Line 20 is reached when there are no more classes to 'pop' off the list of active classes. It returns true to the caller predicate (minimize).

5.1.3. minimize2-predicate

This is a printout of the first part of the minimize2-predicate:

```
22 minimize2(_,_,[]).
23 minimize2(M,AC,[X|Xs]):-
24   format('  Use symbol ~w~n',[X]),
25   geta(M,ACs1),
26   geti(M,ICs1),
27   format('    Active   = ~w~n    Inactive = ~w~n',[ACs1,ICs1]),
```

While line 22 is the termination condition of the predicate, line 23 shows that minimize2 takes three arguments, the first again being the FSA to be minimized, the second being an active class and the last again being the signature of FSA M. minimize2 traverses the latter in a tail-recursive fashion. Line 22 finalizes minimize2: It is triggered when the third argument—the signature list—is empty.

Line 24-27 are used to print out information about the getting on of the predicate. It prints out the symbol which is currently used for splitting (line 24), and then unifies ACs1 and ICs1 with the full active and inactive lists of classes (25-26). Line 27 prints them out.

Following now is the more important part of minimize2, viz. lines 28-41:

```
28   mark(M,AC,X,1),
29   (  setof(S,marked(M,S),Ss)
30   -> format('    Mark: ~w~n',[Ss]),
31      geta(M,ACs),
32      cleara(M),
33      minimizea(M,ACs),
34      geti(M,ICs),
35      cleari(M),
36      minimizei(M,ICs),
37      mark(M,AC,X,0),
38      minimize2(M,AC,Xs)
39   ;  format('    Mark: -~n',[]),
40      minimize2(M,AC,Xs)
41   ).
```

First of all, the mark-predicate 'marks using the active class' (line 28): Every state which is the source of a transition over the symbol currently unified with X into one of the members of the class (state list) presently unified with AC is 'marked'. Subsequently, this is used in line 29 to produce a list Ss comprising all these marked states.

If the result of line 29 was that no state had been marked and thus Ss had been unified with the empty list, the 'else'-part of the 'if-then-else'-construction residing in lines 29-41 is triggered. Here, it is printed out that no state could be marked (line 39) and minimize2 is called recursively to deal with the next symbol from the signature of M.

The 'then'-part of minimize2 (lines 30-38) begins with a printout of the list of all marked states Ss (line 30). Hereafter, the variable ACs is bound with the list of active classes (line 31), and the latter is cleared in line 32. This is to prepare the forthcoming call of the minimizea-predicate (line 33, 'minimize active list', to be explicated below). The same what happened to the active list in lines 31-33 then occurs to the inactive list in lines 34-36. After that, like 37 serves to remove all the markings done in line 29, before line 38 recursively calls minimize2 to deal with the next symbol from M's signature.

Now what remains to illustrate are the two predicates used in minimize2—minimizea for 'minimize active class list' and minimizei for 'minimize inactive class list'.

5.1.4. minimizea-predicate

This is the Prolog code making up the `minimizea`-predicate:

```
42 minimizea(_,[]).
43 minimizea(M,[C|Cs]):-
44   split(M,C,marked,SC,BC),
45   pusha(M,BC),
46   pusha(M,SC),
47   format('    Split active ~w into~n      ~w
             and~n      ~w~n',[C,SC,BC]),
48   minimizea(M,Cs).
```

The job of this predicate is to traverse the active list given to it as its second argument by `minimize2`, and to try to split every class contained in it. This is done in line 44, which forms one of the two absolute core lines in the entire file 'minimize.pl' together with line 51 below. In line 44, class `C` is split according to the `marked`-predicate criterion into the smaller resulting class `SC` and the bigger `BC`. Both split classes are then pushed on the active list (lines 45-46) and the result of the partitioning is printed out (line 47). Line 48 recursively calls `minimizea` to repeat this process until the end of the active list is reached.

5.1.5. minimizei-predicate

`minimizei` follows a very similar path as `minimizea`. This is the code listing:

```
49 minimizei(_,[]).
50 minimizei(M,[C|Cs]):-
51   split(M,C,marked,SC,BC),
52   pusha(M,SC),
53   (  pseudo(M,C)
54   -> pusha(M,BC),
55      retract(pseudo(M,C))
56   ;  pushi(M,BC)
57   ),
58   format('    Split inactive ~w into~n      ~w
             and~n      ~w~n',[C,SC,BC]),
59   minimizei(M,Cs).
```

Here, line 51 is crucial—it splits the given inactive list into `SC` and `BC` respectively. Then, in line 52, it pushes the smaller resulting class `SC` onto the active list. Lines 53-57 hereafter deal with the already mentioned clumsiness which first cropped up in the toplevel `minimize`-predicate. By means of an 'if-then-else'-construction, it is ensured that if a split class is marked as 'pseudo-active', the bigger chunk `BC` of the two resulting splits is put on the active list (line 54, 'then'-part) rather than on the inactive list (line

56, 'else'-part). As can be deduced from line 55, the 'then'-part is only executed once in the entire algorithm: In line 10 above only one class was given the 'pseudo-active'-flag, which is removed in line 55.

Line 58 and 59 conclude the `minimizei`-predicate and with it the exposition of the algorithm part of the minimization process by printing out the result of the above split and by recursively calling the predicate until the end of the inactive list is found.

5.2. Construction part

Let us now concentrate on the easier part of the operation—the construction of a minimized automaton `mini(A)` out of a FSA `A`. This work is done in 'operations.pl', and that is the code needed for it:

```
1  start(mini(A),C):-
2    ensure_minimized(A),
3    inactive(A,Cs),
4    member(C,Cs),
5    member(S,C),
6    start(A,S).

7  trans(mini(A),[S|C1s],X,C2):-
8    ensure_minimized(A),
9    inactive(A,Cs),
10   member([S|C1s],Cs),
11   member(C2,Cs),
12   member(D,C2),
13   trans(A,S,X,D).

14 final(mini(A),C):-
15   ensure_minimized(A),
16   inactive(A,Cs),
17   member(C,Cs),
18   member(S,C),
19   final(A,S).
```

The first part (lines 1-6) states what conditions have to be fulfilled for a start state in `mini(A)`. Line 2 ensures that the automaton `A` must have been minimized in the first place, using the algorithm explained above. `ensure_minimized` is a support predicate taken from 'support.pl'. It minimizes FSA `A` if it has not yet been minimized and does nothing otherwise, since repeating the minimization process would just mean a waste of time. Line 3 after that unifies the list of inactive classes (which is the result of the minimization process) with the variable `Cs`. Now if a state (class) `C` is in this list of inactive classes and if at least one single state `S` within this class is a start state of FSA `A`, then `C` is a start state in `mini(A)`.

The second part (line 7-13) of the definition of the `mini`-operation travels along similarly. First (line 8), `A` is ensured to have been minimized. Then `Cs` is unified with the resulting inactive list of classes. Now a transition over a symbol `X` in FSA `mini(A)` must be justified by a transition from `S` over this symbol to `D` in the original FSA `A`. State `S` must be a single state member of one of the resulting inactive classes, and `D` must also be member of one (the same or another) of the resulting classes. Only transitions which fulfill these conditions are transitions of the minimized automaton.

The final part of the construction of a minimized FSA `mini(A)` is depicted in lines 14-19. They express equal constraints on final states of `mini(A)` as lines 1-6 did on start states: Every final state `C` of the minimized FSA must be warranted by at least one final state `S` in the original automaton `A` (which is an element of the state class `C`).

6. Conclusion

Within the space of the previous five sections, you should have been made familiar with Fistame, a partial implementation of [KK]. After a short overview in Section 1, the paper moved on to introducing the concept of finite-state automata and their encoding and operation upon in the Fistame framework. Section 3 concentrated on exposing all of the built-in operations of Fistame, whilst Section 4 elaborated on the most important implementational issues. Section 5 followed a similar track by explicating the `mini`-operation used for minimizing the number of states in an FSA.

Some thanks. I'd like to thank Martin Kay for coming over to Saarbrücken in summer 1998 and delivering the 'Sprachtechnologie'-course in the first place and Carsten Brockmann for sorting out several nasty LaTeX-problems and much support while implementing the `minimize`-operation.

A. Portability

Fistame is programmed such as to enable it to run on virtually any decent Prolog distribution. It has been successfully tested on three of the most popular Prolog distributions, namely SWI-Prolog (SunOS, Win95, AmigaOS), SICStus Prolog 3 (SunOS) and Quintus Prolog 3.2 (SunOS). To ensure that you have no trouble running Fistame using these or other Prolog distributions on any computer running any operating system, the following subsections flesh out several portability issues. With the help of these, you should be able to get Fistame up and running on your favorite Prolog platform quite quickly.

A.1. Getting it started

Fistame consists of eight files, all of which should reside in one single directory. The files are:

```
Quintus32.pl   examples.pl   minimize.pl    scan.pl
SICStus3.pl    fistame.pl    operations.pl  support.pl
```

To start Fistame up, your Prolog interpreter must be able to locate these files in the first place. One way to ensure that is to change over to the Fistame home directory before starting the Prolog interpreter. After that, you need to locate and consult a ‘startup file’ suiting your Prolog setup. ‘fistame.pl’ is a generic file (working perfectly on SWI-Prolog), while ‘Quintus32.pl’ and ‘SICStus3.pl’ are slightly altered versions of the same file for use with Quintus Prolog 3.2 and SICStus Prolog 3 respectively. Following the generic path, this is about what will happen:

```
[rade@top] (.../rade/Prolog) $ cd fistame/
[rade@top] (.../Prolog/fistame) $ pl
Welcome to SWI-Prolog (Version 3.1.0)
Copyright (c) 1993-1998 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [fistame].
scan compiled, 0.00 sec, 2,184 bytes.
support compiled, 0.01 sec, 5,440 bytes.
minimize compiled, 0.00 sec, 7,996 bytes.
operations compiled, 0.01 sec, 9,144 bytes.
examples compiled, 0.01 sec, 4,536 bytes.
fistame compiled, 0.03 sec, 30,424 bytes.

Yes
```

The startup file uses the `ensure_loaded`-directive to consult all necessary files. If your Prolog distribution does not support this directive, you might have to change this line (file location of the line in brackets):

```
(Quintus32.pl, SICStus3.pl, fistame.pl)
:- ensure_loaded([scan,support,minimize,operations,examples]).
```

A.2. Dynamic predicates

Your Prolog interpreter must be able to declare certain predicates as ‘dynamic’. Usually, this is done by the `dynamic`-directive (SWI-Prolog, SICStus and Quintus Prolog). There are `dynamic`-directives at the beginnings of the files ‘fistame.pl’ (and also ‘Quintus32.pl’and ‘SICStus3.pl’), ‘operations.pl’ and ‘examples.pl’, so change these if your Prolog distribution employs a different syntax.

```
(Quintus32.pl, SICStus3.pl, fistame.pl)
:- dynamic state/2,symbol/2,counter/2.

(operations.pl)
:- dynamic start/2,trans/4,final/2.
```

```
(examples.pl)
:- dynamic start/2,trans/4,final/2.
```

A.3. Multifile predicates

In addition to the ability to handle 'dynamic' predicates, your Prolog interpreter should also be able to cope with 'multifile'-predicates. 'Multifile'-predicates are predicates whose definitions are scattered around different source code files but still belong together. In Fistame, there are four predicates possessing this status, viz. the ones used to encode FSAs: `start/2`, `trans/4`, `final/2`. These are employed in 'examples.pl' to encode FSAs, as well as in 'operations.pl', where there are used to define the Fistame operations (dealt with in Section 3). The directive for enabling these predicates to appear in both files is `multifile` in most modern Prologs (SWI-Prolog, Quintus 3.2, SICStus Prolog 3):

```
(operations.pl)
:- multifile start/2,trans/4,final/2.

(examples.pl)
:- multifile start/2,trans/4,final/2.
```

To run Fistame on your, possibly different Prolog setup, you might have to notify your interpreter in another way of these 'multifile'-predicates.

A.4. append/3, member/2 and name/2-predicates

Fistame makes use of the `append/3`, `member/2` and `name/2`-predicates. In distributions like SWI-Prolog, these predicates are available right from the start. Some distributions (Quintus Prolog 3.2, SICStus Prolog 3), however, do need to have some or all of them loaded into their predicate database separately. This is the line from `SICStus3.pl` which makes sure that `append/3`, `member/2` and `name/2` are available to SICStus Prolog 3:

```
(SICStus3.pl)
:- use_module(library(lists)).
```

This line can be left out if your Prolog supports these predicates without further ado (like SWI-Prolog).

Another variant is brought into play by Quintus Prolog 3.2, which does have the `append/3` and `name/2`-predicates loaded by the start of the interpreter, but lacks the `member/2`-predicate. These are the lines to mend the generic startup file 'fistame.pl' to get by this:

```
(Quintus32.pl)
member(X,[X|_]).
member(X,[_|Xs]):-
  member(X,Xs).
```

References

[Hop71] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[KK] Ronald M. Kaplan and Martin Kay. Finite-state methods in natural-language processing: Algorithms.