User's Guide to *Grail*

Version 2.5

Darrell Raymond
Derick Wood

March 1996

1 Department of Computer Science, University of Western Ontario, London, Canada
2 Department of Computer Science, Hong Kong University of Science and Technology, Kowloon, Hong Kong
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>25</td>
</tr>
<tr>
<td>Objects</td>
<td>27</td>
</tr>
<tr>
<td>Filters</td>
<td>28</td>
</tr>
<tr>
<td>Minimizing machines</td>
<td>30</td>
</tr>
<tr>
<td>Executing machines</td>
<td>31</td>
</tr>
<tr>
<td>Language equivalence is not identity</td>
<td>33</td>
</tr>
<tr>
<td>Using other alphabets</td>
<td>33</td>
</tr>
<tr>
<td>Generating large machines</td>
<td>37</td>
</tr>
<tr>
<td>An extended example</td>
<td>38</td>
</tr>
<tr>
<td>Implementation</td>
<td>43</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>43</td>
</tr>
</tbody>
</table>
Grail is a collection of programs for manipulating finite-state machines, finite languages, and regular expressions. Using Grail you can convert finite-state machines to regular expressions or vice-versa, you can convert finite languages to machines or expressions, and you can convert expressions and machines to finite languages (if the language of the expression or machine is finite). You can minimize machines, make them deterministic, execute them on input strings, enumerate their languages, and perform many other useful activities.

Each of Grail's facilities is provided as a filter that can be used as a standalone program, or in combination with other filters. Most filters take a machine, language, or regular expression as input and produce a new one as output. Input can be entered directly from the keyboard or (more usually) redirected from files. To convert a regular expression into a finite-state machine, for example, one might issue the following command:

```
% echo "(a+b)*(abc)" | retofm
(START) |- 4
  0 a 1
  2 b 3
  0 a 0
  0 a 2
  2 b 0
  2 b 2
  4 a 1
  4 a 0
  4 a 2
  4 b 3
  4 b 0
  4 b 2
  1 a 6
  3 a 6
  4 a 6
  8 c 10
  6 b 8
10 -| (FINAL)
```
The filter retofm converts its input regular expression to a nondeterministic finite-state machine, which it prints on its standard output. The machine is specified as a list of instructions, with some special pseudo-instructions to indicate the states that are start and final.

The output of one filter can be the input for another; for example, we can convert the machine back to a regular expression (the result is folded here to fit onto the page):

```bash
% echo "(a+b)*(abc)" | retofm | fmtore
((aa*a+ba*a+a+b)(b+ba*a)*ba*a+aab+aa+aab+ab+ba*aab+
 ((aa*a+ba*a+a+b)(b+ba*a)*b+b)*ab)c
```

The filter fmtore converts a machine to a regular expression. We can make the machine deterministic, using the filter fmdeterm, before converting it to a regular expression:

```bash
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmtore
(aa*b+bb*aa*b)(aa*b+bb*aa*b)*c
```

We can minimize the deterministic machine, using the filter fmmin, before converting it to a regular expression:

```bash
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmtore
b*aa*b(bb*aa*b+aa*b)*c
```

We can test the membership of a string in the given language by executing it on the machine:

```bash
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmexec "ababababc"
accepted
```

The filter fmexec executes its input machine on an argument string and prints accepted if the string is a member of the language of the machine. Finally, we can enumerate some of the strings in the language of the machine:

```bash
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmenum -n 10
abc
aabc
babc
```

26
aaabc
ababc
baabc
bbabc
aaaabc
aababc
abaabc
abaabc

The filter `fmenu` enumerates the language of a machine, shortest first and then in lexicographical order; the argument `-n 10` specifies the number of strings to be printed.

OBJECTS

*Grail* manages regular expressions, finite languages, and finite-state machines. *Grail*'s regular expressions follow the conventional theoretical notation (not the UNIX notation). Each of the following is a regular expression:

\[
\begin{align*}
\emptyset & \quad \text{empty set} \\
\epsilon & \quad \text{empty string} \\
a-b, A-Z & \quad \text{any single letter} \\
x + y & \quad \text{union of two expressions} \\
x^* & \quad \text{Kleene star}
\end{align*}
\]

*Grail* follows the normal rules of precedence for regular expressions; Kleene star is highest, next is catenation, and lowest is union. Parentheses can be used to override precedence. Internally, *Grail* stores regular expressions with the minimum number of parentheses (even if you input it with redundant parentheses).

The conventional method for describing a finite-state machine is as a 5-tuple of states, labels, instruction relation, start state, and final states. In *Grail*, however, machines are represented completely by lists of instructions. The machine accepting the language `ab`, for example, is given as:

```
(START) |- 0
0 a 1
```

Each instruction is a triple that specifies a source state, a label, and a sink state. States are numbered with nonnegative integers, and labels are single letters. In addition, the machine contains one or more pseudo-instructions to indicate the start and final states. Pseudo-instructions use the special labels \( |- \) and \( |- \), which can be thought of as end-markers on the input stream. The label \( |- \) can appear only with the \((\text{START})\) state, and the label \( |- \) can appear only with the \((\text{FINAL})\) state. \((\text{START})\) can appear only as a source state of a pseudo-instruction, and \((\text{FINAL})\) can appear only as a target state of a pseudo-instruction.

Unlike the conventional model for machines, \textit{Grail} machines can have more than one start state, and (as with conventional machines) more than one final state. Machines with more than one start state are nondeterministic.

Transitions need not be ordered on submission to \textit{Grail}; they’ll be ordered internally in the process of being input. The output of \textit{Grail}’s filters is generally unsorted.

Finite languages are specified as a set of words, one per line. The words need not be sorted. If duplicate words appear in the input, they’re discarded.

FILTERS

The following list provides a brief description of the filters provided by \textit{Grail}. More details on individual filters can be found by consulting the appropriate \textit{man} pages.

**Predicates for finite-state machines**

The following filters return 1 if the argument machine possesses the desired property, and 0 otherwise. A diagnostic message is also written on standard error.
iscomp test a machine for completeness
isdeterm test a machine for determinism
isomorph test two machines for isomorphism
isuniv test a machine for universality

Filters for finite-state machines
Among other functionality, there are filters for constructing finite-state machines, complementing them, completing them, minimizing them, executing them, and enumerating their languages.

    fmcomplement complement a machine
    fmcomplete complete a machine
    fmconcatenate catenate two machines
    fmcross product of two machines
    fmmake deterministic make a machine deterministic
    fmenumerate enumerate strings in the language of a machine
    fmmexecute execute a machine on a given string
    fmminimize minimize a machine by Hopcroft’s method
    fmminrev minimize a machine by reversal
    fmplus plus of a machine
    fmreach reduce a machine to reachable states and instructions
    fmrenumber renumber a machine
    fmreverse reverse a machine
    fmstar star of a machine
    fmprint print information about a machine
    fmtofl convert a machine to a finite language
    fmtoregex convert a machine to a regular expression
    fmunion union of two machines

Predicates for regular expressions
Currently, there are only two predicates provided for regular expressions.

    isempty test for equivalence to empty set
    isnull test for equivalence to empty string
Filters for regular expressions

In addition to the basic construction operations for regular expressions (union, catenation, and star), *Grail* also supports conversion of regular expressions to finite-state machines.

- **recat**: catenate two regular expressions
- **remin**: minimal bracketing of a regular expression
- **restart**: Kleene star of a regular expression
- **retofm**: convert a regular expression to a machine
- **retofl**: convert a regular expression to a finite language
- **reunion**: union of two regular expressions

Filters for finite languages

*Grail* supports the conversion of finite languages to finite-state machines and regular expressions. It also provides left and right ‘quotient’ operators. The left quotient of a finite language and a string \( x \) is the set of words \( y \) such that \( xy \) is in the finite language; right quotient is defined similarly for \( yx \).

- **flappend**: append a given string to every word
- **flexec**: execute a finite language on a given string
- **flfilter**: find intersection of finite language and finite-state machine
- **fllq**: left quotient
- **flprepen**: prepend a given string to every word
- **flprod**: cross product of two finite languages
- **flreverse**: reverse words in a finite language
- **flsq**: right quotient
- **fltovm**: convert a finite language to a finite-state machine
- **fltore**: convert a finite language to a regular expression
- **flunion**: union of two finite languages

MINIMIZING MACHINES

In *Grail* there are two ways to minimize machines. The standard method is to minimize by repeatedly partitioning the set of states
according to differences in instruction labels. This method is implemented in the *Grail* filter `fmmin`. The second method, introduced by Brzozowski, is to reverse the machine, make it deterministic, and repeat these two steps. Using *Grail*, we can show that this procedure results in an isomorphic result:

```
% cat dfm
(START) | - 0
  0 a 1
  0 b 4
  1 c 2
  2 d 3
  3 - | (FINAL)
  4 e 5
  5 f 6
  6 - | (FINAL)

% fmmin <dfm | >out

% fmreverse <dfm | fmdecrypt | fmdecrypt | fmdecrypt >out2

% isomorph out out2
isomorphic
```

Brzozowski's minimization technique is implemented by the *Grail* filter `fmminrev`.

EXECUTING MACHINES

The filter `fmexec` is used to execute a machine, given an input string. By default, this filter simply says whether a string is a member of the language of the machine. For example, we can apply `fmexec` to the machine of the last section:

```
% fmexec dfm "acd"
accepted

% fmexec -d dfm "abc"
not accepted
```
If supplied with the -d option (for ‘diagnostic’), fmexec checks for acceptance and also indicates at each stage of execution which instruction is being taken. Consider fmexec applied to the following machine:

```
% cat nfm
(START) | 1
  1 a 2
  1 a 3
  2 b 2
  3 b 3
  2 c 4
  3 c 5
  4 d 4
  5 d 5
  4 -| (FINAL)
  5 -| (FINAL)
```

```
% fmexec -d nfm "abcd"
on a take instructions
  1 a 2
  1 a 3
on b take instructions
  2 b 2
  3 b 3
on c take instructions
  2 c 4
  3 c 5
on d take instructions
  4 d 4
  5 d 5
terminate on final states 4 5
accepted
```
One of the standard problems in textbooks on automata theory is to determine whether two regular expressions denote the same language. This is difficult because, unlike machines, minimal regular expressions are not unique. One procedure for checking language equivalence involves several steps: (i) convert the expressions to nondeterministic machines (ii) convert the nondeterministic machines to deterministic machines (iii) minimize the deterministic machines (iv) test the machines for isomorphism. If done manually, this is a tedious process; however, it can be done easily with Grail simply by combining the appropriate filters. For example:

```
% echo "(rs+r)*r" | retofm | fmterm | fminmin | >out1
% echo "r(sr+r)*r" | retofm | fmterm | fminmin | >out2
% isomorph out1 out2
isomorphic
```

The two expressions are of the same size, are minimal (we determine this by inspection), and they denote the same language, but they’re not identical.

Non-identical but language-equivalent regular expressions are often produced by application of Grail filters.

**USING OTHER ALPHABETS**

As distributed, Grail is provided with source code for two types of alphabets: characters (used in the other examples in this paper), and regular expressions. It’s possible to recompile Grail to manage alphabets of your own choice. Consider for example an alphabet that consists of ordered pairs of integers. A finite-state machine over this alphabet looks like this:

```
(START) |- 0
0 [1,2] 1
1 [2,2] 1
1 [3,4] 2
2 -| (FINAL)
```

We can convert this machine to a regular expression of ordered pairs:
We can enumerate the language of the machine, generating a set of strings of ordered pairs:

% fmenu -n 10 op
[1, 2] [3, 4]
[1, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]
[1, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [2, 2] [3, 4]

We can complement the machine:

% fcment op
(START) 1 - 0
0 [1, 2] 1
1 [2, 2] 1
1 [3, 4] 2
0 [2, 2] 3
0 [3, 4] 3
2 [1, 2] 3
2 [2, 2] 3
2 [3, 4] 3
1 [1, 2] 3
3 [1, 2] 3
3 [2, 2] 3
3 [3, 4] 3
0 - 1 (FINAL)
3 - 1 (FINAL)
1 - 1 (FINAL)

*Grail* doesn’t read an explicit specification of the alphabet of its machines, and so must infer the alphabet over which complementation
is to be performed. *Grail*'s complement operator assumes that the
set of labels on the instructions defines the whole alphabet, and so
complementation is done with respect to that set. This makes it
possible to do complementation when the alphabet is chosen from a
potentially infinite set, like that of ordered pairs.3

We can also manipulate machines whose instruction labels are
regular expressions:

(START) l- 0
0 <ab*> 1
0 <ba*> 2
1 <a+b+c >3
2 <c(d+e)> 3
3 <x> 0
3 -l (FINAL)

Note that we use the angle brackets to delimit each regular expres-
sion. We can enumerate the language of this machine, producing a
set of strings of regular expressions:

% fmenum -n 10 re
<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*
<ba*><c(d+e)>*<x><ba*><c(d+e)>*
<ab*><a+b+c<ab*><a+b+c<ba*><c(d+e)>*
<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*
<ab*><a+b+c<ba*><c(d+e)>*<x><ab*><a+b+c<ba*><c(d+e)>*

We can also complete the machine (that is, produce an equivalent
machine in which every state has an instruction on every symbol).
Completion, like complement, is done with respect to the limited

---

3 If the alphabet defined by a given machine's instructions does not represent
the set over which you want complementation to be performed, it is relatively
simple to generate a language-equivalent machine that is appropriate—you
simply add a single non-final sink state, and add as many instructions as are
necessary to include the desired symbols from your alphabet.
alphabet of only those labels that appear on the instructions of the input machine:

% fmcompre
(START) |- 0
0 <ba*> 2
0 <ab*> 1
1 <a+b+c 0
2 <c(d+e)> 3
3 <x> 0
0 <a+b+c 4
0 <c(d+e)> 4
0 <x> 4
3 <ba*> 4
3 <ab*> 4
3 <a+b+c 4
3 <c(d+e)> 4
2 <ba*> 4
2 <ab*> 4
2 <a+b+c 4
2 <x> 4
1 <ba*> 4
1 <ab*> 4
1 <c(d+e)> 4
1 <x> 4
4 <ba*> 4
4 <ab*> 4
4 <a+b+c 4
4 <c(d+e)> 4
4 <x> 4
3 |- (FINAL)

Finally, we can generate a regular expression corresponding to the complete machine:

% fmcompre | fmtore
% bin/fmcomp remach | bin/fmto re
<ba*><c(d+e)*><x><ba*>>+(<ab*>+(<ba*><c(d+e)>x><ba*>)x+(<a+b+c<ab*>+(<a+b+c<ba*>(<c(d+e)*><x><ba*>))x+(<a+b+c<ab*>+(<a+b+c<ba*>(<c(d+e)*><x><ba*>))x+(<a+b+c<ab*>+(<a+b+c<ba*>(<c(d+e)*><x><ba*>))x+(<a+b+c<ab*>+(<a+b+c<ba*>(<c(
Notice that while the names of the filters for these special alphabets are the same as the names of the filters for the standard alphabet, we cannot use the same filters. Each alphabet requires a new set of filters. You can either use different names for these filters, or you may put them in different directories and modify your `PATH` as necessary.

**Generating Large Machines**

Our previous examples showed *Grail* filters being used in pipelines. *Grail* filters can also be used in general purpose shell scripts. Since machines and expressions are stored as text files, they can also be processed with standard filters. In the following session, we output a machine (to display its content), then apply cross product recursively to the machine, using `wc` to compute the size of the resulting machines:

```bash
$ cat nfm
(START) | -- 0
  0 a 1
  0 a 2
  1 -- (FINAL)
  2 -- (FINAL)

$ for i in 1 2 3 4
  do
    bin/fmcross nfm nfm >tmp
    mv tmp nfm
    wc nfm
  done
  9  27  89 nfm
  33  99  349 nfm
  513  1539  6413 nfm
131073 393219 2162701 nfm
$
As we recursively apply cross product, the resulting machines grow in size very rapidly.

The preceding script was written in the Bourne shell (sh) rather than the C-shell (csh). We could just as easily have called Grail filters from ksh, bash, tcsh, vi, or any other program that can launch processes as part of its activity.

The machines generated by cross product of a machine with itself have the same language (as before, we can determine this by making the result of the cross product deterministic, minimizing, and checking for isomorphism). Generating large machines for a given language is useful for evaluating the performance of other Grail filters.

AN EXTENDED EXAMPLE

In this section we show how Grail can be used to do some simple lexical processing.

We start with a file containing a list of C++ keywords, one word per line. We'll convert this to a regular expression with the Unix program tr. Next, we convert the regular expression to a finite-state machine; the conversion is nondeterministic, incomplete, and nonuniversal.

```
% tr '\012' '+' < keywd
asm+auto+break+case+catch+char+class+const+continue+
+default+delete+do+double+else+enum+extern+float+
+for+friend+goto+if+inline+int+long+new+operator+pr
ivate+protected+public+register+return+short+signe
d+sizeof+static+struct+switch+template+this+throw+
try+typedef+union+unsigned+virtual+void+volatile+w
hile

% tr '\012' '+' <keywd | retofm >key.fm

% isdeterm key.fm
nondeterministic

% iscomp key.fm

38
We can make the machine deterministic and then minimize it, using either Hopcroft’s algorithm or reversal and subset construction. The results of the two algorithms are isomorphic, and language-equivalent with the original machine.

Using `wc` shows us the sizes of the machines that are produced:

\[
\begin{array}{ccc}
353 & 1059 & 3876 \\
263 & 789 & 2579 \\
175 & 525 & 1429 \\
\end{array}
\]

We can enumerate the language of the result. Note that the keywords are produced in order of their length, and then sorted lexicographically.
default
private
typedef
virtual
continue
operator
register
template
unsigned
volatile
protected

We can execute the machines with various strings and, using the \(-d\) option, show the instructions that are executed at each point.

\%
\texttt{fmexec key.det "protected"}
\texttt{accepted}

\%
\texttt{fmexec -d key.fm "private"}
\texttt{on p take instructions}
244 p 245
258 p 259
276 p 277
\texttt{on r take instructions}
245 r 247
259 r 261
\texttt{on i take instructions}
247 i 249
\texttt{no states accessible on V}
\texttt{not accepted}

Next we produce the complementary machine, which will accept any string other than the C++ keywords. This is useful for determining a subset of valid identifiers. We enumerate the first 15 of these (note that the empty string is not a keyword, though of course it is not an identifier either). We can test potential identifiers by executing them on the complement machine.

\%
\texttt{fmcment key.mv >key.cment}
% fmenum -n 15 key.cment

a
b
c
d
e
f
g
h
i
k
l
m
n
o

% fmexec -d key.cment "protectx"
on p take instructions
  0 p 16
on r take instructions
  16 r 49
on o take instructions
  49 o 82
on t take instructions
  82 t 107
on e take instructions
  107 e 120
on c take instructions
  120 c 125
on t take instructions
  125 t 93
on x take instructions
  93 x 127
terminate on final states 127
accepted
IMPLEMENTATION

*Grail* is written in C++. It includes classes for regular expressions (re), finite languages (fl), and finite-state machines (fm). It includes its own array, string, list, set, and bit vector classes, which are also useful for programming that does not involve machines or expressions. The class library provides all the capabilities of the filters and more, accessible directly from a C++ program. For more information on programming with the *Grail* class library, consult the *Programmer’s Guide to Grail*.

ACKNOWLEDGEMENTS

This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada. Darrell Raymond can be reached at drraymon@csd.uwo.ca. Derick Wood can be reached at dwood@cs.ust.hk.