User's Guide to *Grail*

Version 2.3

Darrell Raymond[1]
Derick Wood[2]

January 1995

1 Department of Computer Science, University of Waterloo, Waterloo, Canada
2 Department of Computer Science, Hong Kong University of Science and Technology, Kowloon, Hong Kong

TABLE OF CONTENTS

Introduction	·	23
Objects	·	25
Filters	·	26
Minimizing machines	·	28
Executing machines	·	28
Language equivalence is not identity	·	30
Using other alphabets	·	30
Generating large machines	·	34
An extended example	·	35
Implementation	·	39
Acknowledgements	·	40

INTRODUCTION

Grail is a collection of programs for manipulating finite-state machines and regular expressions. Using *Grail* you can convert regular expressions to finite-state machines and vice versa, 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 or regular expression as input and produce a new machine or expression as output. Expressions and machines 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):

```
% echo "(a+b)*(abc)" | retofm | fmtore
((aa*a+ba*a+a+b)(b+ba*a)*ba*aab+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 may choose to make the machine deterministic, using the filter `fmdeterm`, before converting it to a regular expression:

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

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

```
% 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:

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmexec "abababab c"
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:

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmenum -n 10
abc
aabc
babc
aaabc
ababc
baabc
bbabc
aaaabc
```

```
aababc
abaabc
```

The filter `fmenum` 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 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:

{}	empty set
""	empty string
a-b,A-Z	any single letter
xy	catenation of two expressions
x + y	union of two expressions
x*	Kleene star

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
1 b 2
2 -| (FINAL)
```

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 `(START)` state, and the label `-|` can appear only with the `(FINAL)` state. `(START)` can appear only as a source state of a pseudo-instruction, and `(FINAL)` can appear only as a target state of a pseudo-instruction.

Unlike the conventional model for machines, *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 *Grail*; they'll be ordered internally in the process of being input. The output of *Grail*'s filters is not generally sorted.

FILTERS

The following list provides a brief description of the filters provided by *Grail*. More details on individual filters can be found by consulting their *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.

`fmcment`	complement a machine
`fmcomp`	complete a machine
`fmcat`	catenate two machines
`fmcross`	cross product of two machines
`fmdeterm`	make a machine deterministic
`fmenum`	enumerate strings in the language of a machine
`fmexec`	execute of a machine on a given string
`fmmin`	minimize a machine by Hopcroft's method
`fmminrev`	minimize of a machine by reversal
`fmplus`	plus of a machine
`fmreach`	reduce of a machine to reachable states and instructions
`fmrenum`	renumber a machine
`fmreverse`	reverse a machine
`fmstar`	star of a machine
`fmtore`	convert of a machine to 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
`restar`	Kleene star of a regular expression
`retofm`	convert a regular expression to a machine
`reunion`	union of two regular expressions

Minimizing machines

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 | fmdeterm | fmreverse | fmdeterm >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` will not only check for acceptance, but it will also indicate 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
```

LANGUAGE EQUIVALENCE IS NOT IDENTITY

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 nfms (ii) convert the nfms to dfms (iii) minimize the dfms (iv) test 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 | fmdeterm | fmmin | >out1
% echo "r(sr+r)*" | retofm | fmdeterm | fmmin | >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 are not identical.

Non-identical but language-equivalent regular expressions can also be produced by *Grail*, without the user being aware of it.

USING OTHER ALPHABETS

As distributed, *Grail* can be compiled with filters for three types of alphabets: characters (used in the other examples in this paper), ordered pairs of integers, and regular expressions. A machine that has ordered pairs of integers as its 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:

```
% fPtorP op
[1,2][2,2]*[3,4]
```

We can enumerate the language of the machine, generating a set of strings of ordered pairs:

```
% fPenum -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:

```
% fPcment op
(START) |- 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 -| (FINAL)
3 -| (FINAL)
1 -| (FINAL)
```

Grail does not have a separate specification of the alphabet of its machines. Thus, its complement operator assumes that the set of labels on the instructions defines the whole alphabet to be used for the purpose of complement. This is particularly useful 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) |- 0
0 <ab*> 1
0 <ba*> 2
1 <a+b+c >3
2 <c(d+e)*> 3
3 <x> 0
3 -| (FINAL)
```

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

```
% fXenum -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><ba*><c(d+e)*>
<ba*><c(d+e)*><x><ba*><c(d+e)*><x><ba*><c(d+e)*>
<ab*><a+b+c<ab*><a+b+c<ab*><a+b+c<ba*><c(d+e)*>
<ba*><c(d+e)*><x><ab*><a+b+c<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 alphabet of only those labels that appear on the instructions of the input machine:

```
% fXcomp re
```

3 If the alphabet defined by a given machine's instructions is incomplete, it is always possible to generate a language-equivalent machine with additional labels, simply by adding instructions with those labels to a non-final sink state.

```
(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:

```
% fXcomp re | fXtorX
% bin/fXcomp remach | bin/fXtorX
<ba*>(<c(d+e)*><x><ba*>)*<c(d+e)*>+(<ab*>+<ba*>(<c(d+e)*>
<x><ba*>)*<c(d+e)*><x><ab*>)(<a+b+c<ab*>+<a+b+c<ba*>(<c(
d+e)*><x><ba*>)* <c(d+e)*><x><ab*>)*<a+b+c<ba*>(<c(d+e)*>
<x><ba*>)*<c(d+e)*>
```

Notice that the names of the filters for these special alphabets are simple modifications of the names of the filters for the standard alphabet. We use `fP` and `rP` in filters for machines and expressions of ordered pairs, and `fX` and `rX` for machines and expressions of regular expressions.

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:

```
$ 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      97 nfm
      33      99     381 nfm
     513    1539    6925 nfm
  131073  393219 2293773 nfm
$
```

As we recursively apply cross product, the resulting machines grow in size very rapidly, as does *Grail*'s use of memory; it requires almost 20 Mbytes to compute the last iteration of cross product.

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 regular expression that defines the language consisting of C++ keywords. This can be converted to a finite-state machine. The conversion is nondeterministic, incomplete, and nonuniversal.

```
% cat keywd
asm+auto+break+case+catch+char+class+const+continu
e+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

% retofm keywd >key.fm

% isdeterm key.fm
nondeterministic

% iscomp key.fm
not complete

% isuniv key.fm
nonuniversal
```

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, but they are only language-equivalent with the original machine.

```
% fmdeterm key.fm >key.det

% isdeterm key.det
deterministic

% fmminrev key.det >key.mv

% fmmin key.det >key.min

% isomorph key.mv key.min
isomorphic

% isomorph key.mv key.fm
nonisomorphic
```

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

```
% retofm keywd | wc
     353    1059    3876

% retofm keywd | fmdeterm | wc
     263     789    2579

% retofm keywd | fmdeterm | fmmin | wc
     175     525    1429
```

We can enumerate the language of the result. Note that the keywords are produced in order of their length, and then sorted lexicographically.

```
% fmenum key.det
do
if
asm
for
int
```

```
new
try
auto
case
char
else
enum
goto
long
this
void
break
catch
class
const
float
short
throw
union
while
delete
double
extern
friend
inline
public
return
signed
sizeof
static
struct
switch
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.

```
% fmexec key.det "protected"
accepted

% fmexec -d key.fm "priVate"
on p take instructions
244 p 245
258 p 259
276 p 277
on r take instructions
245 r 247
259 r 261
on i take instructions
247 i 249
no states acccessible on V
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.

```
% 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`) and standard finite-state machines (`fm`). It includes its own array, string, list, and set 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 grants from the Natural Sciences and Engineering Research Council of Canada, and the Information Technology Research Centre of Ontario. The first author was also supported by an IBM Canada Research Fellowship.
Darrell Raymond can be reached at `drraymon@daisy.uwaterloo.ca`.
Derick Wood can be reached at `dwood@cs.ust.hk`.