User's Guide to *Grail*
Version 2.0

Darrell Raymond [*] Derick Wood [†]

January 1994

[*]Department of Computer Science, University of Waterloo, Waterloo, Canada
[†]Department of Computer Science, University of Western Ontario, London, Canada

Contents

1 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
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
(START) |- 4
1 a 6
3 a 6
4 a 6
6 b 8
8 c 10
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
abc+aa*abc+b(b+aa*b)*abc+b(b+aa*b)*aa*abc+aa*b(b+aa*b)*abc+
aa*b(b+aa*b)*aa*abc
```

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(aa*b+bb*aa*b)*c+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(aa*b+bb*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 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 `10` specifies the number of strings to be printed.

2 Objects.

Grail manages three types of objects: regular expressions, conventional finite-state machines, and extended finite-state machines.

Regular expressions. *Grail*'s regular expressions are very similar to standard regular expressions. 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).

Finite-state machines. 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 of course, can have more than one final state). Machines with more than one start state are nondeterministic.

Transitions need not be ordered on input to *Grail*, and are generally not ordered when output by the various filters.

Extended finite-state machines. In addition to conventional finite-state machines, *Grail* supports extended machines. An extended finite-state machine has regular expressions as instruction labels. For example:

```
(START) |- 0
0 ab* 1
1 (c+d+e)abc 2
2 -| (FINAL)
```

The pseudo-instructions are the same as for conventional finite-state machines.

Any conventional machine is also an extended machine, because a single-symbol instruction label is also a regular expression. Thus, any filter for extended machines can be applied to conventional machines. Filters for conventional machines can be applied only to conventional machines.

For some problems, it is easier to specify an extended machine than it is to specify a conventional machine. However, at present *Grail* does not support all types of operation on extended machines; in particular, subset construction (for producing deterministic machines) and minimization are not supported.

3 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.

3.1 Filters for standard machines.

`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

3.2 Predicates for standard 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

3.3 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

3.4 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

3.5 Filters for extended machines.

Any of the filters for extended machines can also be used for standard machines as well.

`xfmcat`	catenate two extended machines
`xfmplus`	plus of an extended machine
`xfmreach`	reduce an extended machine to reachable states and instructions
`xfmreverse`	reverse an extended machine
`xfmstar`	star of an extended machine
`xfmtore`	convert an extended machine to a regular expression
`xfmunion`	union of two extended machines

4 More examples.

4.1 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`.

4.2 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
```

4.3 Language equivalence is not identity.

One of the standard problems in textbooks on machines 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. The order of instructions sometimes affects the output of a particular filter. Consider the following orderings of the same set of instructions:

```
    dfm              dfm2             dfm3

    4 -| (FINAL)     4 -| (FINAL)     1 b 0
    1 b 0            (START) |- 0     2 d 0
    2 d 0            3 g 2            0 a 1
    2 e 3            1 c 2            3 g 2
    (START) |- 0     2 e 3            1 c 2
    3 f 4            3 f 4            2 e 3
    3 g 2            0 a 1            3 f 4
    1 c 2            1 b 0            4 -| (FINAL)
    0 a 1            2 d 0            (START) |- 0
```

The instructions define the same machine; they are merely ordered differently. This reordering affects *Grail*'s conversion of machines to regular expressions:

```
% fmtore <dfm
a(ba+c(eg)*da)*c(eg)*ef

% fmtore <dfm2
a(ba+cda+ce(ge)*gda)*ce(ge)*f

% fmtore <dfm3
a(ba)*c(da(ba)*c)*e(g(da(ba)*c)*e)*f
```

The reason for the difference is the different input orderings of the instructions. `fmtore` generates regular expressions by means of state elimination; it successively replaces states in the machine with a regular expression that captures that state's instructions. The input order of instructions determines the order in which states are eliminated, and hence the resulting form of the regular expression.

As with the previous example, it is possible to test for language equivalence by converting to machines, minimizing, and testing for isomorphism:

```
% fmtore <dfm1 | retofm | fmdeterm | fmmin >test1
% fmtore <dfm2 | retofm | fmdeterm | fmmin >test2
% fmtore <dfm3 | retofm | fmdeterm | fmmin >test3
% isomorph test1 test2
isomorphic
% isomorph test2 test3
isomorphic
```

4.4 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. So 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 of known language is useful for evaluating the performance of other *Grail* filters.

5 Implementation.

Grail is written in C++. It includes classes for regular expressions (`re`) and standard finite-state machines (`fm`). It also includes its own string, list, and set classes, which are useful even for programming that does not involve machines (the list and set classes are templates). 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.

6 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@csd.uwo.ca.