Grail: Engineering Automata in C++,
Version 2.5\textsuperscript{†}

Darrell R. Raymond\textsuperscript{*} and Derick Wood\textsuperscript{†}

Technical Report HKUST-CS96-24
June 1996

\textsuperscript{*}Department of Computer Science
University of Western Ontario
London, Ontario N2L 5B7
Canada

\textsuperscript{†}Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon
Hong Kong

Abstract

This report documents the current version of \textit{Grail} with three documents: an introduction to \textit{Grail}, a user’s guide, and a programmer’s guide. \textit{Grail} is a symbolic system for manipulating formal-language theory objects. The current version allows the manipulation of (extended) finite-state machines, regular expressions, and finite languages. The system is written in C++ with a systematic use of templates to provide parameterized classes and hence make the production of user-defined extensions easier.

\textsuperscript{†}This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada.
Grail: Engineering Automata in C++

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

Introduction · 3
Features of Grail · 4
Grail’s design · 7
A short history of Grail · 10
Related software systems · 15
Some empirical lessons · 17
How do I obtain Grail? · 18
Acknowledgements · 18
References · 19
INTRODUCTION

I saw the Holy Grail, All pall’d in crimson samite.

Tennyson, *Holy Grail*

They seemed to seek some Hofbrauhaus of the spirit like a grail, hold a krug of Munich beer like a chalice.

T. Pynchon, *V*

This equipment can be used to counter heat-seeking missiles such as the Soviet SA-7 Grail shoulder-fired weapon, now extensively deployed in Third World countries.

Daily Telegraph, Nov. 22, 1985

We can’t go doddering across Malaya behind an inspired crackpot following the Holy Grail, can we?

H.M. Tomlinson, *Gallions Reach*

The Edge was Fox’s grail, that essential fraction of sheer human talent, nontransferable, locked in the skulls of the world’s hottest research scientists.

W. Gibson, *New Rose Hotel*

*Grail* is a symbolic computation environment for finite-state machines, regular expressions, and finite languages. Using *Grail*, one can input machines or expressions, convert them from one form to the other, minimize, determinize, complement, and perform many other operations. *Grail* is intended for use in teaching, for research into the properties of machines, and for efficient computation with machines.

This paper provides a basic introduction to *Grail* and describes
some of its history and development. If you want to use Grail, you should also consult the User's Guide to Grail and the man pages for the individual filters. If you are installing Grail, or if you want to write C++ programs that use Grail, consult the Programmer's Guide to Grail and the Release Notes.

Grail is written in C++. It can be accessed either through a process library or through a C++ class library. The process library is used much like other filters; from a command shell, a user can execute processes on files or input streams, generating output that can be filtered by other processes. The C++ class library can be compiled into applications that need direct access to Grail, or that wish to minimize the costs of stream I/O.

The name ‘grail’ isn’t necessarily an acronym, though it could be. In the past, we have sometimes suggested that Grail stands for something like ‘grammars, regular expressions, automata, languages’ (we’ve never come up with something convincing for the i!). It’s probably just as reasonable to think of our Grail experience as a search for the hofbrauhaus of formal language theory.

FEATURES OF GRAIL

Version 2.5 of Grail enables you to manipulate parameterizable finite-state machines, regular expressions, and finite languages. By ‘parameterizable’, we mean that the alphabet is not restricted to the usual twenty-six letters and ten digits. Instead, all algorithms are written in a type-independent manner, so that any valid C++ base type and any user-defined type or class can be the alphabet of a finite-state machine, regular expression, or finite language.

Regular expressions in Grail use the conventional notation of the theoretical community. Grail supports catenation, union, and Kleene star for regular expressions, along with parentheses to specify precedence (complement is not supported). The following are examples of regular expressions acceptable to Grail:

\[ a+b \]
\[ ((a+bcde*)+c)* \]
\[ {} \]
\[ """+a \]
The expression \{\} denotes the empty set, and the expression "" denotes the empty string.

The traditional representation for automata is the 5-tuple:

\[< Q, \Sigma, \delta, s, F >\]

where \(Q\) is the set of states, \(\Sigma\) is the input alphabet, \(\delta\) is a partial relation \(\delta : Q \times \Sigma \rightarrow \{Q\}\), \(s \in Q\) is the start state, and \(F \subseteq Q\) is a set of final states. In *Grail*, we represent machines as sets of instructions. A machine that accepts the language \(ab\), for example, is specified by:

\[
\text{(START)} |\rightarrow 0 \\
0 \ a \ 1 \\
1 \ b \ 2 \\
2 \ \rightarrow \text{(FINAL)}
\]

Each instruction is a triple consisting of a source state, an instruction symbol, and the corresponding target state. The start and final states of the machine are indicated by means of special *pseudo-instructions*, whose labels are special symbols that can be thought of as endmarkers on the input tape. The states (START) and (FINAL) are *pseudo-states*; they simply indicate that the other state in the instruction is a start or final state. The set of (non-pseudo) instructions is an enumeration of the instruction relation. The alphabet of the machine is given implicitly; it is the set of symbols that appear in (non-pseudo) instructions. *Grail*’s machines differ from conventional machines in that we permit multiple start states as well as multiple final states. *Grail*’s machines are also parameterizable.

A finite language is simply a set of words (each of finite length) constructed from the characters of the alphabet.

To the user, *Grail* is a set of individual filter programs that operate on streams containing descriptions of finite-state machines, regular expressions, or finite languages. Most filters take a *Grail* object (that is, a finite-state machine, regular expression, or finite language) as input, and produce a *Grail* object as output. Objects 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

whose output would be

(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 an input regular expression into a non-deterministic finite-state machine, which it prints on its standard output. This output can be the input for another filter; for example, a filter that converts the machine back into a regular expression (folded here to fit onto the page):

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

For those who want to avoid the cost of I/O implicit in the use of the filter approach, Graal can also be accessed directly as a C++ library. The above filter command

% echo "(a+b)*(abc)" | retofm | fmtore

can also be written directly in C++:


```
#include "grail.h"

main()
{
    re<char> r;
    char* example = "(a+b)*(abc)\n";

    istrstream(example, strlen(example)) >> r;
    r.fmtore(r.retofm());

    cout << r << endl;
}
```

In the above program, the `istrstream` function is used to convert an internal string into input to be read as a regular expression; the `retofm` function converts the expression into a machine, and the `fmtore` function converts it back to an expression.

*Grail*'s algorithms are independent of the type of the alphabet defined. We can have, for example, machines whose alphabet is the (infinite) set of ordered pairs of integers:

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

Each of *Grail*'s filters can be compiled to work with this symbol set; thus, we can convert such a machine to a regular expression (of ordered pairs), enumerate its language (which is a set of strings of ordered pairs), and so on.

*Grail*'s Design

Most tools for working with machines and expressions are designed for a specific application, such as program parsing. *Grail*, on the other hand, is designed to be a general-purpose package for symbolic computation with machines and expressions. We intend for *Grail* to (eventually) fill all of the following needs:
• research

*Grail* should facilitate the theoretical and practical investigation of machines and expressions, and the development of new algorithms for processing them. *Grail* has already been useful in investigating the properties of subset construction (Leslie 92).

• education

*Grail* should facilitate teaching about machines. In part, it should do this by making it easier to experiment with machines, but we also hope that *Grail* will add a leavening of engineering to a subject that is mostly taught as theoretical mathematics. *Grail* has already been used for undergraduate teaching.

• application

*Grail* should facilitate the use of machines in solving applied problems, such as protocol testing, embedded state machines, executing concurrent processes, parsing, string searching, and any other application that can be described by machines or expressions. Users of *Grail* have employed it for grammar transduction and natural language processing.

The key theme of *Grail*'s design is modularity. We seek modularity not just because it is the generally accepted route to a good software design, but because we expect that adding new facilities to *Grail* and developing new uses for *Grail* will be the most common activity of both its users and its designers. Modularity in *Grail* arises in four important areas:

• philosophy

Other approaches to software for machines assume that minimal, deterministic machines are the desired end result of all processing. In *Grail* we do not make this assumption; we treat machines, languages, and expressions as equal first-class objects. Programmers will find in *Grail* a collection of useful tools and a number of ways to connect the tools to address new and interesting problems in formal language theory. Moreover, we
intend to make many algorithms and implementations of algorithms accessible within Grail, both the (apparently) inefficient as well as the efficient, in order to facilitate experimentation and study, as well as to generate test cases.

- process-based software
  Instead of developing yet another language for writing machine programs, Grail is based on a set of individual processes that can be accessed by any command shell or any program that is capable of launching processes. Processes are modules whose encapsulation is enforced by the operating system; a process-based approach encourages programmers to develop simple, generally-applicable tools. A second advantage of this approach is that it is easy to distribute computation; by using the capabilities of rsh to set up internet pipes, we can run processes on different machines. A third advantage is that a process-based approach separates language issues from machines processing. It also leverages users' knowledge of shell programming; rather than requiring users to learn a new language, users can exploit sh, csh, ksh, bash, perl, and many other languages.

- textual interchange
  A multiple-process design requires some form of interprocess communication, since processes cannot access each others' data. We use a textual description of machines and regular expressions as the intermediary for Grail. Each process reads a textual description of the input machine, converts it into an internal form, processes it, and writes a textual description of an output machine. The advantage of this approach is that the input and output can be read, edited, and manipulated by standard utilities such as vi, sort and wc. The disadvantage is the extra cost of encoding and decoding between the language and internal forms, and the cost of process invocation and switching. In our experience, this cost is small compared to the cost of (for example) converting between objects or producing minimized machines.

- C++ class library
C++ encourages encapsulation and the definition of interfaces, and hence encourages modularity in low-level code. In addition, we make extensive use of template classes, which in effect define operations on 'black boxes' that are ready to be instantiated with the user's choice of modules.

*Grail*'s 41 filters are listed in Table 1.

**A SHORT HISTORY OF *Grail***

*Grail* was preceded by two packages written at the University of Waterloo. The earlier effort was Leiss’s *REGPACK* (Leiss 79), a package written in 1977 to support experimentation and research with finite-state machines. *REGPACK*, written in SPITBOL, supported the conversion of nondeterministic machines to deterministic machines, minimization of deterministic machines, and construction of syntactic monoids. While *REGPACK* did not directly influence the current effort, it is interesting to note that Leiss’s goal of an environment for experimentation with machines is still one of our primary goals.

A program with more direct influence on *Grail* was Howard Johnson’s *INR* (Johnson 86). *INR* was developed because of Johnson’s interest in rational relations and their use in defining string similarity (Johnson 83). *INR* takes rational relations (including regular expressions) as input and converts them into finite-state machines, which can then be manipulated in various ways. *INR* can produce single- or multiple-tape machines; the latter are useful for describing transducers, since one tape can be considered an output tape for the other (input) tapes.

Johnson made special efforts to ensure that *INR* was a highly efficient and powerful tool for managing machines. His goal was the effective processing of machines with thousands of states and instructions. As a result, *INR* is written very compactly in C, and is especially efficient in handling potentially costly tasks such as memory allocation, subset construction, and minimization of machines. The basic algorithms for handling such tasks are well known, but there has been relatively little attention paid to efficient implementation of these algorithms. Johnson made the effort to develop efficient implementations, with the result that *INR* was the only software system
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmcomplement</td>
<td>complement a machine</td>
</tr>
<tr>
<td>fmclosure</td>
<td>complete a machine</td>
</tr>
<tr>
<td>fmcatenate</td>
<td>catenate two machines</td>
</tr>
<tr>
<td>fmcross</td>
<td>cross product of two machines</td>
</tr>
<tr>
<td>fmdecompos</td>
<td>make a machine deterministic</td>
</tr>
<tr>
<td>fmenumerate</td>
<td>enumerate strings in the language of a machine</td>
</tr>
<tr>
<td>fmsexecute</td>
<td>execute a machine on a given string</td>
</tr>
<tr>
<td>fminimize</td>
<td>minimize a machine by Hopcroft’s method</td>
</tr>
<tr>
<td>fmminimize</td>
<td>minimize a machine by reversal</td>
</tr>
<tr>
<td>fmplus</td>
<td>plus of a machine</td>
</tr>
<tr>
<td>fmreachable</td>
<td>reduce a machine to reachable states and instructions</td>
</tr>
<tr>
<td>fmmenum</td>
<td>canonical renumbering of a machine</td>
</tr>
<tr>
<td>fsereverse</td>
<td>reverse a machine</td>
</tr>
<tr>
<td>fmstar</td>
<td>star of a machine</td>
</tr>
<tr>
<td>fmsstats</td>
<td>print information about a machine</td>
</tr>
<tr>
<td>fmtofl</td>
<td>convert a machine to a finite language</td>
</tr>
<tr>
<td>fmtofm</td>
<td>convert a machine to regular expression</td>
</tr>
<tr>
<td>fmunion</td>
<td>union of two machines</td>
</tr>
<tr>
<td>fappend</td>
<td>append a given string to every word</td>
</tr>
<tr>
<td>fexec</td>
<td>execute a finite language on a given string</td>
</tr>
<tr>
<td>ffilter</td>
<td>find intersection of finite language and finite-state machine</td>
</tr>
<tr>
<td>fleftq</td>
<td>left quotient</td>
</tr>
<tr>
<td>fleftprep</td>
<td>prepend a given string to every word</td>
</tr>
<tr>
<td>fleftprod</td>
<td>cross product of two finite languages</td>
</tr>
<tr>
<td>freverse</td>
<td>reverse words in a finite language</td>
</tr>
<tr>
<td>frightq</td>
<td>right quotient</td>
</tr>
<tr>
<td>fltofm</td>
<td>convert a finite language to a finite-state machine</td>
</tr>
<tr>
<td>fltore</td>
<td>convert a finite language to a regular expression</td>
</tr>
<tr>
<td>flunion</td>
<td>union of two finite languages</td>
</tr>
<tr>
<td>iscomp</td>
<td>test a machine for completeness</td>
</tr>
<tr>
<td>isdeterministic</td>
<td>test a machine for determinism</td>
</tr>
<tr>
<td>isomorphic</td>
<td>test two machines for isomorphism</td>
</tr>
<tr>
<td>isuniversal</td>
<td>test a machine for universality</td>
</tr>
<tr>
<td>isempty</td>
<td>test for equivalence to empty set</td>
</tr>
<tr>
<td>isnull</td>
<td>test for equivalence to empty string</td>
</tr>
<tr>
<td>frecat</td>
<td>catenate two regular expressions</td>
</tr>
<tr>
<td>fremin</td>
<td>minimal bracketing of a regular expression</td>
</tr>
<tr>
<td>frestar</td>
<td>Kleene star of a regular expression</td>
</tr>
<tr>
<td>retofm</td>
<td>convert a regular expression to a machine</td>
</tr>
<tr>
<td>retofl</td>
<td>convert a regular expression to a finite language</td>
</tr>
<tr>
<td>reunion</td>
<td>union of two regular expressions</td>
</tr>
</tbody>
</table>

Table 1.1: Grail filters
capable of handling the transduction of the *Oxford English Dictionary* (Kazman 86). Even today, many of INR’s capabilities are more advanced than those of other software (though we like to think that *Grail* is catching up). The present effort has borrowed INR’s philosophy of combining powerful capabilities with efficient design, as well as its notation for machines.

The first project to actually use the name ‘Grail’ was a joint effort between Howard Johnson, Carl-Johan Seger, and Derick Wood. This project extended INR to handle context-free grammars and machines with regular expressions as instruction labels. Software developed for this project consisted of a layer of code that used INR as an underlying computational engine. After some work, this effort was discontinued.

The *Grail* project was resuscitated by the present authors in 1990. We began with the observation that some issues were not satisfactorily handled either by INR or ‘old Grail.’ The first issue was obscurity. In pursuit of efficiency, INR had become a somewhat complex and monolithic piece of code. The layer of software added by ‘old Grail’ merely increased the complexity, because it was not easily maintainable or modifiable. The lack of documentation for INR and ‘old Grail’ made this software difficult to understand for anyone other than its programmers. Thus, the first order of business was to develop software that was more approachable and better documented, to improve maintainability and robustness, and to ensure that many programmers could work on the software.

The second issue was modularity. Much of the difficulty of building upon INR was a result of its tightly connected structure. Adding a new algorithm for subset construction, for example, required knowing much about the internals of INR, including its data structures, memory allocation, parser, and so on. We wanted a software environment in which programmers could work on algorithms without having to learn too much about the details of the existing code. This meant that we would have to build the software in a modular fashion, devising interfaces at several levels.

The third issue was generality. Like most systems that have appeared since, INR assumed that the user wanted to input regular expressions and receive deterministic, minimized machines as output. INR did not support the user who wanted to input machines
and produce regular expressions as output. We wanted *Grail* to be a general purpose manipulation language, in which one could convert machines and expressions freely, with user control over minimization and determinism.

*Grail* version 0.5 was written in C, and consisted of the following filters:

- **cross**: compute the cross product of two machines
- **lreverse**: reverse the input using empty-string instructions
- **min**: minimize the input by Hopcroft’s partition algorithm
- **mini**: minimize the input by reversal and subset construction
- **percent**: compute the alternation (i.e. $(ab)^+$) of two machines
- **plus**: compute $\text{star} - \epsilon$ of the machine
- **quest**: compute the machine $+ \epsilon$
- **reverse**: reverse the input machine
- **star**: compute the Kleene star of the input machine
- **subset**: subset construction of the input machine
- **union**: compute the union of two machines

These filters accessed a library of functions that did most of the actual work (the filters themselves were essentially simple I/O routines). The library contained procedures for handling I/O and for processing machines. The idea behind this decomposition was that the filters should be efficient enough for most problems involving machines; for very large or complex problems, a competent C programmer could access the library directly and thereby avoid any inefficiency introduced by process communication.

While the filters were reasonably successful, the library was not. Our C code was not particularly reliable, readable, or reusable. This latter problem was irritating both aesthetically and as a pure engineering problem. Operations on machines and regular expressions involve frequent manipulation of container structures such as sets and relations; it would be both elegant and efficient to use a single implementation of these structures for many different contents. Using C, however, one can provide this generality only by giving up strict type checking. In spite of these problems, version 0.5 did support a significant research project on subset construction (Leslie 92).
We decided to switch to C++ to re-implement Grail. We made this choice of language under the impression that we would develop an elegant class hierarchy that would greatly increase code reuse and the overall robustness of the system. While we have made some use of classes, C++ has been much more important for its better clarity and robustness, which are a result of its strict type checking and encapsulation. C++’s template facility is indispensable to Grail, and recent versions of the software have made more extensive use of inheritance and virtual functions.

Versions 0.8 through 1.2 of Grail saw the development of our C++ class library, which included the classes set, list, string, regexp, trans (transition), state, fa, tset (sets of transitions), and xfa (extended finite machine). This latter class defines machines that have regular expressions as transition labels. The set and list classes are template classes; they and xfa were our first attempt to rely on C++’s ability to support code reuse. In addition to rewriting our existing code in C++, we also added more functionality—the number of filters jumped from 11 to 34. Version 1.0 introduced an automatic testing facility that was used to check that changes to code still resulted in working filters. Version 1.1 introduced an automatic profiling facility that was used to test that purported improvements actually did lead to more efficient code. Version 1.2 was subjected to quality checks, both through the use of Purify and through correcting the bugs and inconsistencies that were discovered by compiling the code with two C++ compilers that are more strict than cfront.

Version 2 of Grail added support for parameterizable machines and expressions. Parameterizable finite-state machines can take any type as instruction label, and parameterizable regular expressions can take any type as a symbol class. Version 2 thus dispenses with the distinction between xfa and fa (each is an instance of the new parameterizable machine class fm), and has extended the reach of the regexp class (now called re) beyond strings of ASCII alphabetic characters. Version 2 also dispenses with the class tset and makes string a parameterized class.

The most recent version of Grail is Version 2.5. This version introduces a class for finite languages (fl) and for memory allocation (pool), and exhibits improved performance for large machines.
RELATED SOFTWARE SYSTEMS

Recently, several systems for computing with machines have appeared in the literature or have been made available over the Internet.

Bruce Watson has written a C++ toolkit for finite-state machines and regular expressions called the FIRE Engine (Watson 94a, 94b). This package has the goals of efficiency and modularity, and implements more algorithms than does Grail. The FIRE Engine does not come with a non-programmer interface, such as Grail's filters.

Champarnaud's AUTOMATE system, written in C, supports finite-state machines and finite semigroups (Champarnaud and Hansel 91). It can compute deterministic minimal machines, syntactic monoids, and transition monoids of regular languages.

The AMORE system, written in C, supports finite-state machines, regular expressions, and syntactic monoids (Jansen et al. 90). It can produce minimal DFAs, handle e-NFAs, and perform various tests on syntactic monoids (for example, star-freeness, finiteness, and cofiniteness). AMORE can also display its machines graphically.

Both AMORE and AUTOMATE have goals similar to those of Grail—to serve as a research environment, to facilitate the study of machine implementations, and to provide a package for executing machines for other purposes (such as validating concurrent programs). Where Grail differs is in its emphasis on providing a full symbolic computing environment; in its provision of both filters and a class library; and in the fact that Grail does not attempt to provide its own graphical user interface or programming language. AMORE and AUTOMATE appear to be monolithic programs that attempt to provide a single interface to the user.

One use of machines is for hardware verification and protocol checking. FANCY, the Finite AutomatoN Checker of nancY, is Stefan Krischer's tool for formal hardware verification. It provides equivalence and inclusion checking for finite-state machines and is accessible through a graphical user interface.

FADELA, the Finite Automaton DEbugging LAnguage, is a project directed by Gjalt de Jong (van der Zanden 90). FADELA is designed to investigate \( \omega \)-regular languages (that is, regular languages whose words are of infinite length). FADELA supports the production of
deterministic Müller machines, and can convert these machines into regular expressions. FADELA also supports other operations on machines including minimization and complement.

An interesting experience is the development of machine tools in Nuprl, a proof language based on the lambda calculus (Kreitz 86). Definitions were constructed in Nuprl for finite sets, strings, tuples, and deterministic machines. Nuprl was then able to construct a proof of the pumping lemma. The main point of this work was not the development of an environment for manipulating machines, but an illustration of the utility of the Nuprl proof development system.

We know of several other systems whose motivation is primarily pedagogical. An early effort was GRAMPA, which was only partially implemented (Barnes 72). More recently, Hannay has built a Hypercard-based system for simulating machines (Hannay 92). This program appears to be useful for introductory teaching purposes, and for simulating small machines. FLAP, the Formal Languages and Automata Package, comes from Rensselaer Polytechnic Institute. FLAP supports the drawing and execution of finite-state machines, pushdown machines and Turing machines. FLAP can handle non-deterministic machines, provides the ability to step through the execution of a machine, and supports paper output (LoSacco and Rodger 93). The COLOS project at the University of Milan has a system called AUTOMATA, which permits students to work with finite-state automata, push-down automata, Mealy and Moore machines, regular expressions, and context-free grammars. It includes an X windows interface. Finally, Turing's World is a program for teaching the basics of finite-state machines and Turing machines (Barwise and Etchemendy 93). This program's strength is a nice graphical interface to the machines.

In addition to these systems, there is a vast amount of work on using grammars and machines in applications. Many operating system utilities understand a limited form of regular expression, for example, and almost every text editor provides general-purpose search-and-replace capabilities. The machines used in such tools are generally custom built, or perhaps adapted from custom code; operating systems have yet to offer a standard machine package for handling parameterizable machines and expressions in the same way that they offer parameterizable sorting and searching routines.
Some Empirical Lessons

Developing Grail has taught us much about implementing algorithms for finite-state machines. C++ is an important contributor to the robustness of the code, mainly because of strict type checking. The C++ compiler has resisted many questionable constructs that were unquestioningly accepted by C. Consequently, programming bugs and errors less frequently show up in low-level operations. When bugs do appear, they are now almost always incorrect specifications of algorithms.

Grail has also taught us some lessons that apply to the construction of mathematical libraries in general. One lesson is that a library of routines is only half the battle; the other half is in developing a library of test data, and in the provision of a mechanism for automatic testing and performance evaluation. In the early stages of development, Grail’s filters were tested with simple machines and the results were checked by hand. As the pace of development increased, however, this was no longer sufficient; one cannot very well test tens of programs on each of several test cases by hand, and one cannot test very large machines or expressions by hand at all, since the probability of a manual error in checking soon becomes higher than the probability of an error in the code. Thus, it becomes necessary to automate testing. Automation is also essential in performance evaluation, which relies on large inputs in order to thoroughly exercise the code. One approach to generating large test cases is to apply filters that generate non-isomorphic machines that are language equivalent. Repeatedly converting between machine and regular expression, for example, will result in a large machine that accepts a known language. Hence, the result of processing such a machine can be tested by minimizing and comparing it to the known minimal machine. Another related tactic is to repeatedly take the cross product of a nondeterministic machine with itself; there will be an exponential blowup in the size of the result, which is still language equivalent with the original.

A second important lesson is that a sound theoretical understanding of an algorithm is not the same as a sound implementation. To paraphrase a popular saying, a little knowledge of worst-case performance is a dangerous thing. Algorithms that have bad worst case
performance may be quite acceptable for most practical uses. Subset construction, in particular, is exponential in the worst case, but empirical study shows that the number of machines that exhibit this behaviour is small (Leslie 92). Moreover, it appears to be predictable from the input whether an exponential result is likely to occur. Since most users do not want to store or further use exponential output, predicting this result may be sufficient. Another instance of this behavior is reported by Bruce Watson, who notes that Brzozowski’s algorithm for minimization (applying reversal and subset construction twice) performs better than Hopcroft’s algorithm in practice, even though worst-case analysis of the two algorithms suggests the opposite (Watson 95).

On the other hand, a sloppy implementation of a well-known algorithm with reasonable average case performance may be unacceptable for every large input. Linear-time algorithms can easily become quadratic-time if careful attention is not paid to problems such as the proper management of sets.

**HOW DO I OBTAIN Grail?**

*Grail* is available without charge to researchers and students, or anyone who wishes to use the software for their own private education. Version 2.5 of *Grail* can be obtained from the Grail project’s World Wide Web site, at

\[
\text{http://www.csd.uwo.ca/research/grail}
\]

This site also contains links to many other automata research projects.

*Grail* is not in the public domain. It cannot be sold, used for commercial purposes, or included as part of a commercial product without our permission.

**ACKNOWLEDGEMENTS**

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


Kreitz 86 C. Kreitz, “Constructive Automata Theory Implemented with the Nuprl Proof Development System”, TR-86-779, De-
department of Computer Science, Cornell University, Ithaca, New York (September 1986).

Leiss 77 E. Leiss, “REGPACK: An Interactive Package for Regular Languages and Finite Automata”, CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada (October 1977).


Release Notes for \textit{Grail}

Version 2.5

Darrell Raymond$^1$
Roger Robson$^2$

March 1996

$^1$ Department of Computer Science, University of Western Ontario, London, Canada
$^2$ Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
TABLE OF CONTENTS

INTRODUCTION - 83
PERFORMANCE - 83
NEW CLASSES - 84
COMPILERS - 90
MISCELLANEOUS - 98
LIST OF CHANGES - 99
ACKNOWLEDGEMENTS - 101
INTRODUCTION

This document describes the changes and improvements in Grail Version 2.5. Version 2.5 introduces finite languages, custom memory management schemes, and improved performance.

This is not a complete description of Grail; for that, consult other parts of The Grail Papers. This document, and Grail itself, can be found at our Web site:

http://www.csd.uwo.ca/research/grail.

The main changes in Version 2.5 are as follows:

1. Grail now includes support for finite languages through the class \texttt{fl}.

2. There is improved memory usage (partly through the class \texttt{pool}), and hence improved efficiency in a variety of areas.

3. Grail can now be compiled with Symantec 7.0, IBM CSet++ 2.0, and Microsoft Visual C++ 2.0.

4. The \texttt{null.exp} class is no longer present.

5. cfront 3.0.2 is still supported, but only just.

6. Memory leaks in the regular expression classes have been fixed.

These changes are discussed in more detail in the remainder of this report.

PERFORMANCE

We have spent some time improving the performance of Grail for large machines. Our motivation for working on this aspect of Grail is due to requests from a variety of computational linguists who wish to convert large dictionaries to finite-state machines and then massage them.\footnote{We would particularly like to thank Franz Guenther and Bourbaker Meddeb-Hamrouni for their interest in Grail for these purposes.}

Version 2.4 of Grail was effectively limited to machines of less than 10,000 states, or in other words, dictionaries of approximately
1000 words. Version 2.5 of *Grail* is better by an order of magnitude; it can handle machines in the range of 100,000 states and dictionaries of 20,000 words. This is still an order of magnitude less than what is needed for large-scale natural language processing, so look for further improvements in the future.

Through profiling we learned that much of *Grail*’s time had been spent in creating and destroying temporary arrays; many hundreds or thousands or arrays might be created and destroyed, even though only a small number were ever in use at one time. Clearly, what was needed was a small pool of arrays that could be reused, and so we added a mechanism to `array` that keeps a small buffer of arrays available. This greatly reduces the need to allocate and free memory, leading to a substantial time savings.

Version 2.5 also employs a scheme for custom memory management, based on the new class `pool`, which is described in greater detail in the next section. `pool` is used in Version 2.5 to manage regular expressions, but it is a general-purpose memory management class that will probably see greater use in future versions of *Grail*.

We eliminated several subtle problems that were resulting in memory mismanagement. One interesting problem occurred because of the definition of `array::operator=(const array&)`. In this routine, the target array was reallocated to the maximum size of the argument array, on the assumption that the target array should have as much room to expand as does the argument array. This action proved to be particularly costly in situations where a single temporary variable is used repeatedly to add elements to the array; if the temporary variable had needed to be very large at some point in the past, then its maximum size may be much larger than its current size, and this overhead is passed on to the target array. Removing this overhead improved several routines.

**NEW CLASSES**

`pool`

The class `pool` provides general-purpose dynamic memory management for classes that have large numbers of small objects. It is well known that C++ programs can be improved by an order of mag-
nitude simply by using custom memory allocation rather than the
default provided by new and delete. pool is our first attempt to
provide this kind of efficiency in a general way in Grail.

pool is a template class that manages a set of fixed arrays of its
argument type. The arrays are allocated according to powers of two.
A new array is allocated only when all elements of smaller arrays
are already in use. pool uses a bitmap to keep track of the elements
that are in use. As elements are used, the bits in the bitmap are set;
as elements are returned to the pool, the bits are cleared.

In order to use pool with a given class, you must define an in-
stance of a pool for the class. Suppose you want cat_exp<char> to
use pool memory management. Then you would create a pool this
way:

    pool<cat_exp<char> > cat_pool;

and overload new and delete for the class cat_exp this way:

void*
cat_exp<char>::operator new(size_t)
{
    return cat_pool.get_member();
}

void
cat_exp<char>::operator delete(void* p)
{
    cat_pool.return_member(p);
}

pool will now take care of allocating blocks of cat_exp<char>s.

A memory management scheme for small objects should exhibit:

1. fast new and delete
2. bulk allocation of memory
3. the ability to retrieve unused memory
4. low fragmentation
5. low overhead
pool provides us with most of these features. **new** and **delete** are much faster than the default, because pool simply manages pointers to existing memory; it does not allocate a new piece of memory for every call to **new**, nor free it on every call to **delete**. Bulk allocation of memory is done: each new block that pool allocates is twice the size of the previous block, and is allocated in one call. pool uses its bitmap to register any members that are returned to the pool, and will use any returned members before allocating new blocks of memory.

pool can suffer from fragmentation, if for example every other object is returned to the pool. Fragmentation occurs because pool does not rearrange objects—once they are allocated, they stay put. The advantage of this is low overhead for using pools. If objects were rearranged, then fragmentation could be avoided, but it would probably necessitate additional overhead in determining the new addresses for objects.

One thing that pool does not do, which might be considered desirable, is return whole blocks of memory to the heap if they are unused. Doing so is problematic. It is more costly, because we must test for an unused blocks that may need to be returned. It’s quite possible that we would not actually free any memory, since if any single member of a block is in use, then that whole block cannot be returned.

Given that there are almost always fewer than 10 blocks in any pool (they are in increasing size, in powers of 2, starting at 128), the odds are that most blocks would have some member in use. Thus, testing for an unused block probably simply adds nothing but overhead to the system.

It is sometimes possible to solve the problem of unused blocks in another way. pool is carefully designed so that one can have more than one pool for a given class (by making **new** and **delete** more complex). Thus, if one knows that memory will be used heavily in one part of a program, and then can be freed, one can arrange for the memory to be returned simply by using different pools at different points of the execution of a program.

---

4 Unless we permitted rearranging blocks.
The class \texttt{fl} describes a finite language: that is, a language composed of a finite number of finite-length words. The internal storage mechanism for \texttt{fl} is a set\langle\texttt{string<char>}\rangle that contains the enumeration of the language. The input and output functions for \texttt{fl} employ a hardwired syntax that assumes newlines are used to delimit words. This (or any other) fixed syntax is unacceptable in general, but it was easy to implement for this release of \textit{Grail}. Another alternative is to permit user-defined delimiters, perhaps using the current \textit{Grail} approach of user-defined delimiter variables. We are generally unhappy with the current strategy for delimiter handling, partly because of the number of global variables and partly because there is no assistance given to ensure that conflicting delimiters have not been chosen. We decided to use a simple solution for the current implementation of \texttt{fl}, and develop a more general technique for user-defined representations of all objects in future releases of \textit{Grail}.

Several new filters accompany the introduction of \texttt{fl}.

\texttt{fntofl} converts a finite-state machine to a finite language. Since not all finite-state machines correspond to a finite language, there is a check to ensure that an input machine \texttt{is\_finite()}. 

The check for finiteness is accomplished by passing through the machine collecting reachable states and looking for repetitions. The conversion itself uses a similar algorithm, recording a word whenever a path reaches a final state.

\texttt{floftofm} converts a finite language to a finite-state machine. This conversion is always possible. The generated machine has the form of a trie, and hence is deterministic, but usually non-minimal.

\texttt{fltoexp} converts a finite language to a regular expression. This conversion is always possible. The expression is not ‘minimal’. Given the following finite language:

- \texttt{adder}
- \texttt{addend}
the resulting expression is \texttt{adder+addend+sum+subtract} and not \texttt{add(\texttt{er+end})+su(\texttt{m+tract})}.

\texttt{retofl} converts a regular expression to a finite language. Only star-free regular expressions are finite, and the filter checks for star-freeness. One exception is permitted: any starred subexpression that evaluates to the empty string is allowed.

The conversion algorithm used is similar to the one used by \texttt{retofm}. For \texttt{retofl}, however, each subexpression is converted to a finite language instead of a submachine.

\texttt{flexec} replicates the behaviour of \texttt{fmexec}, except that it does not accept the \texttt{-d switch}, and it 'executes' finite languages instead of finite-state machines.

\texttt{ffilter} accepts a finite language and a finite-state machine. The filter outputs a language consisting of all words belonging to the finite language which are accepted by the finite-state machine.

\texttt{ffprod} returns the cross product of two finite languages. The product of any finite language with an empty language yields an empty language. The cross product of any string with the empty string yields the original string.

\texttt{freverse} reverses a finite language. The filter has no effect on empty languages or empty strings. A new member function was added to the string class to simplify the reversal code, and to make the string reversal functionality publicly available.

\texttt{funion} returns the union of two finite languages. Since a finite language is a set of words, the filter is implemented by performing a set union.

\texttt{flq} returns the left quotient of a finite language and a string. The left quotient of a language \(L\) and a string \(x\) is defined as the language of all words \(y\) such that \(xy\) is in \(L\). The left quotient of any language \(L\) with the empty string yields the language \(L\).
The left quotient of the empty language and any string yields the empty language.

\( \text{flrq} \) returns the right quotient of a finite language and a string. This is similar to the \( \text{fl}\text{lq} \) filter. The right quotient of a language \( L \) and a string \( x \) is defined as the language of all words \( y \) such that \( yx \) is in \( L \).

\( \text{flappend} \) appends a given string to every word in a finite language. It is the equivalent of the \( fl \leftarrow fl*string \) operation. It is also, in a sense, the inverse of the left quotient operation. Appending a string to the empty language yields the empty language.

\( \text{flprepen} \) prepends a given string to every word in a finite language. It is the equivalent of the \( fl \leftarrow string*fl \) operation. It is also, in a sense, the inverse of the right quotient operation.

The automatic testing facility has been updated to include tests for all applicable filters. The new tests entailed the creation of six finite language test objects, named \( l1 \) through \( l6 \). The following filters have no automatic tests:

- \( \text{flexec} \)
- \( \text{flfilter} \)
- \( \text{flappend} \)
- \( \text{flprepen} \)
- \( \text{fllq} \)
- \( \text{flrq} \)

Little attempt was made to optimize the time efficiency of the filters.

No attempt has been made to extend the finite language filters for use with the \( \text{mlychar, mlyint or re} \) languages, due to the problem with the stream operators. The functionality of \( \text{flexec} \) and \( \text{flfilter} \) should probably be modified for the Mealy types, to allow output to be true Mealy output rather than simply the input strings.

The following improvements and modifications to \( fl \) are recommended:

1. The feasibility of storing the finite languages internally as a trie or sorted list should be examined.
2. The stream operators should be improved once the delimiter problem has been solved. This will also allow extension to other languages as indicated above.

3. Derick Wood recommends a shuffle operation for string and languages. Shuffling two strings means interleaving their characters. Shuffling two languages means a product of the two languages, in which words are shuffled together instead of concatenated.

COMPILERS

This section describes some of the peculiarities of particular compilers, and the techniques we have used to overcome them.

cfront

It is still possible to use cfront to compile Grail. We use version 3.0.2, dated 12/01/92, on a Sparcstation 20 running SunOS Release 4.1.3_U1.

As noted in the Release notes for 2.4, cfront 3.0.2 confuses the class set and the member function ‘set’ in class bits, presumably because they both appear in the same (single) file that constitutes the Grail source code. We have left the #ifdefs that were put in place in Version 2.4, but we will probably remove them in the next release of Grail.

A new problem introduced in Version 2.5 is due to the pool class. Because we want a single pool per class for cat_exp, plus_exp, star_exp, and symbol_exp, we normally have a static variable in each class definition as follows:

static pool<cat_exp<S> > cat_pool;

C++ does not normally permit classes to contain members of their own type, but it makes an exception for static members. In this case, the static member is actually a different class parameterized by the class type. It is perhaps not surprising that cfront can’t recognize that this is a legal construct.

In order to use the pool class under cfront, we do not use the static definitions of the pools, and instead manually instantiate a
pool for each parameterization of *Grail*. When used by cfront, the file `classes/re/memory.src` contains the following:

- `pool<cat_exp<char> > cat_pool;`  
- `pool<plus_exp<char> > plus_pool;`  
- `pool<star_exp<char> > star_pool;`  
- `pool<symbol_exp<char> > symbol_exp;`

This solution is ugly but workable; it requires the programmer to manually instantiate pools for regular expressions of each alphabet that are to be used. Note that it does not have the same level of encapsulation or robustness as the static solution.

**DCC**

We compiled the SGI binaries with DCC under IRIX Release 5.3. This compiler needs no `#ifdefs`. Some points:

1. DCC found several declared-but-unused variables that were not mentioned by other compilers.

2. DCC mistakenly reported that `q` was used before it was set in the following fragment of code:

   ```c
   int q;
   for (k=-1;;k=q)
   {
     if ((q = inter.next(i)) == -1)
       break;
   }
   ```

We do not ship a statically bound version of the SGI binaries, as the machine we used to compile them did not have the appropriate library.

**xlC**

We compiled the RS6000 binaries with version 1.0 of xlC, on an RS/6000. There is one `#ifdef` for xlC in our code, in `array/array.src`: 
#ifndef XLC
template <class Item>
int array<Item>::max_pool = 32;

template <class Item>
array<Item>* array<Item>::pool = (array<Item>*)
  new char[array<Item>::max_pool * sizeof(array<Item>)];
#endif

#ifdef XLC
int max_pool = 32;

template <class Item>
array<Item>* array<Item>::pool = (array<Item>*)
  new char[max_pool * sizeof(array<Item>)];
#endif

xC has a bit of a problem with recognizing the static class variable
array<Item>::max_pool, so we have to make it an external variable.

It would be desirable to have statically linked binaries for the
RS/6000. Mike Whitney of the University of Victoria suggested using
the following flags to produce a static executable:

LDLFLGS = -bnoo -bI:/lib/syscalls.exp -liconv -bmodelcsect

When we have tried this in the past, it was reported that the results
were not executable under some versions of AIX. The distributed
RS/6000 binaries are, consequently, not statically compiled.

Visual C++

Version 1.52 of Visual C++ does not support templates, so it cannot
compile Grail. Version 2.0, which runs only under Windows NT, will
compile Grail. We found the following problems when compiling
Grail under Version 2.0:

1. Visual C++ requires explicit declarations of templated friend
   member functions before the class definition is seen. This re-
   quired four declarations:
// in re\re.h

#ifdef MSVC
    template <class S> class re;
    template <class S>
    ostream&
    operator<<(ostream&, const re<S>&);
    template <class S>
    istream&
    operator>>(istream&, re<S>&);
#endif

// in fm\fm.h

#ifdef MSVC
    template <class Label> class fm;
    template <class Label>
    istream&
    operator>>(istream&, fm<Label>&);
#endif

// in inst\inst.h

#ifdef MSVC
    template <class Label> class inst;
    template <class Label>
    istream&
    operator>>(istream &, inst<Label>&);
#endif

Note that the re class required two declarations, and that in each case a declaration of the class is necessary before the declaration of the friend function. MSVC was helpful when it first
flagged this error—it said explicitly what was required.

2. MSVC did not like an explicit pointer/class member function expression in set pluseq.src. This was corrected by breaking up the expression and using a temporary variable:

```c
#ifdef MSVC
   array<Item> & tmp = *this;
   tmp += q;
   // note: *this is changed because tmp is a
   // reference variable
#else
tmp = *this->array<Item>::operator+=(q);
#endif
```

3. MSVC does not equate stringstream.h with the DOS filename stringstream.h, similarly to CSet. This was corrected by using an #ifdef.

4. MSVC uses a different signature for set new handler. In their version, the PF argument is an:

```c
int function(size_t)
```

and not a:

```c
void function()
```

This was corrected by including new.h, and modifying the new_error function (all protected by #ifdefs).

5. fmreverse was not recognized. MSVC passes the filter name without .EXE, and hence the nine-character name did not match the eight-character name passed from DOS.

Makoto Murata of Fuji Xerox found that VC++ 2.0 required the same #ifdefs as does cfront for the use of pools with regular expressions; that is, VC++ 2.0 doesn't seem to understand the combination of static data members and templates.
Murata also notes that VC++ 2.0 doesn’t seem to recognize \texttt{set\_new\_handler}, even if \texttt{new.h} is included. His solution is to do the following:

\begin{verbatim}
#ifndef MSVC
set\_new\_handler(&new\_error); // error handler for new
#endif
\end{verbatim}

\textit{Grail}, as shipped, does not include changes or \texttt{ifdefs} for Visual C++.

**Symantec 7.0**

Although it is possible to compile \textit{Grail} 2.5 with Symantec 7.0, the changes are substantial enough that we do not include them in the delivered source code. For those who are using this compiler, here is a list of what needs to be done:

1. Symantec will not compile properly unless all formal template parameter names are identical. This does not apply to template parameter names for the \texttt{mealy} and \texttt{pair} classes. This is most easily accomplished by searching for \texttt{Label} and \texttt{Item}, and replacing with \texttt{S}. Also, \texttt{inst}
   \begin{verbatim}
std.h uses T as a parameter name.
\end{verbatim}

2. An explicit declaration of
   \begin{verbatim}
template <class S>
class inst<S>;

template <class S>
ostream&
operator<<(ostream&, inst<S>&);
\end{verbatim}

is required in \texttt{inst}
\begin{verbatim}
instr.h before the inst class definition. Also, the following must be added to inst
ostream.src:
\end{verbatim}

\begin{verbatim}
#include "../re/re.h"
\end{verbatim}
Although this is not required for the compilation process, lack of a definition for re thwarts the instantiation process.

3. Explicit manual instantiations of

```cpp
fm<char>::member
fm<re<char> >::member
```

are required to circumvent faulty function signature matching in Symantec’s compiler. Also, manual instantiations of

```cpp
operator>>(istream &, inst<char>)
operator<<(ostream &, fm<re<char>>)
```

are required to circumvent faulty instance-generation in the compiler.

4. Trailing tab characters must be removed from ‘cp’ commands in the Makefiles.

5. The arithmetic-if statements in bits/pluseq.src (line 13) and string.h (line 51) must be edited to include superfluous bracketing of the if-then and if-else arguments, because they contain assignments.

6. An explicit declaration of

```cpp
template <class S>
ostream&
operator<<(ostream&, const fm<S>&);
```

must be made in fm.h just prior to the fm class definition.

### CSet++ 2.0

CSet had the following problems with *Grail*:

1. CSet does not recognize `strstream.h` as an alias for the DOS-shortened `strstream.h`. This occurs in grail.h and inst/inst.h.
2. CSet suffers from the same unusual include-path semantics as Borland; that is, the need to know all include directories as absolute paths, rather than relative to the file from which they are included.

3. CSet has a macro called \texttt{max}. During the initialization of array class (in the constructor), the \texttt{max} data member is initialized on an initialization list. This syntax is misunderstood as a call to the \texttt{max} macro. Moving the \texttt{max} initialization to an assignment in the constructor body.

4. An error was generated in \texttt{array/sort}. Apparently, CSet requires that the linkage type of functions passed by pointer explicitly match that of its formal argument. In this case, the default CSet linkage specifier, \texttt{Optlink} must be added.

5. CSet generates an "informational" warning regarding the use of static members in template classes. This warning generally involves the exporting of such members from a library or compilation unit. As such, they do not apply to the current implementation of \texttt{Grail}. A compiler switch can be used to suppress "informational" warnings.

6. CSet supports the unix convention of not adding \texttt{.EXE} to \texttt{argv[0]}, and so uses \texttt{names.h} rather than \texttt{dosnames.h}.

CSet++ can compile \texttt{Grail} 2.5 in under a minute on a Pentium/90 with 16 Mbytes of EDO memory.

\textbf{Watcom}

\texttt{Grail} compiles successfully with Watcom 9.5, 10.0a, and 10.5. Watcom has a number of problems with constructs that the other compilers passed without complaint. In particular, it grumbled about using a \texttt{cast} in situations like this in \texttt{set.h} and \texttt{list.h}:

\begin{verbatim}
#ifndef WATCOM
    { (array<Item> &) *this = 1; return *this; }
#endif
#endif WATCOM
#include WATCOM
    { array<Item>::operator=(1); return *this; }
\end{verbatim}
Watcom also needed a special instantiation of `string::operator>>` in order to handle `istringstream` (which should just be a derivation from the operator for `iostream`), and an explicit declaration and definition of `mealy::operator<<` (which should just be a derivation from the operator for `fm`).

MISCELLANEOUS

We’ve used both Purify and Quantify fairly extensively on this version of `Grail`.

We have removed all errors that we found having to do with memory leaks, array bounds that were exceeded, and uninitialized memory references. More precisely, we removed all UMRS that were in our code; there are some UMRS left over, but these are in the `iostream` library that is supplied with `cfront 3.0.2`, so there’s not much we can do about those. A sample of these errors follows;

**UMR: Uninitialized memory read:**

* This is occurring while in:
  
  ```
  ios::flags(long) [libC.a]
  fstreambase::fstreambase() [libC.a]
  fstream::fstream() [libC.a]
  get_one(fm<char>*,int,char**,char*) [grail.o]
  main [grail.o]
  start [crt0.o]
  *
  ```

  * Reading 4 bytes from 0xeffff7e4 on the stack.
  * Address 0xeffff7e4 is 20 bytes below frame pointer
    in function get_one(fm<char>*,int,char**,char*)

Similar uninitialized memory reads also occur in the following `iostream` functions:

```
  ios::init(streambuf*) [libC.a]
  ios::precision(int) [libC.a]
  ios::fill(char) [libC.a]
  ios::tie(ostream*) [libC.a]
  ios::flags(long) [libC.a]
```
LIST OF CHANGES

1. Added test for self-assignment to bits::bits(const bits&).

2. Plugged memory leak in copy constructors for cat_exp, plus_exp, star_exp.

3. Removed copy() function member from re classes.

4. Custom memory allocation for array class.

5. pool class written, and re's allocation done by pool.

6. Batch copying in array::operator+=(const array&).

7. 'Destructive' copying in array::operator+=(array*).

8. Used a bitmap to manage sets in fm::states().

9. Fixed "make clean" in tests/Makefile.

10. bits::next() written.

11. bzero, bcmp, bcopy used instead of loops in various functions.

12. array::swap() written.

13. pool class written.

14. Memory leaks in re fixed.

15. re's parsing is now linear instead of quadratic (only one call to istrstream).

16. Overgenerous memory allocation in array::operator=() fixed (only allocate sz, not max).
17. Variables in grail.C allocated at time of use, not at start of procedure.

18. fmminrev need not take deterministic input.


20. null_exp removed from re.

21. fl class added.

22. install_unix/install_dos dichotomy removed.

23. makefile now uses wmake and DOS commands instead of MKS make and Korn shell.

24. fm::min_by_partition() and fm::enumerate now remove unreachable states. Thanks to Makoto Murata for bug reports.

25. fm::member now properly handles empty strings.

26. re_lambda and re/std.h removed as unused. Thanks to Wolfgang Frech for bug reports.

27. retofm restored to mlychar and mlyint. Thanks to Wolfgang Frech for bug reports.

28. mealy::dmember fixed to actually transduce instead of just copying the input stream. Thanks to Wolfgang Frech for bug reports.

29. re::print should start with priority 0 (otherwise some Kleene star expressions are done incorrectly). Thanks to Wolfgang Frech for bug reports.

30. array::merge function written.

31. fm::reachable_states greatly improved. Thanks to Jonathan
Buss for complaints.

ACKNOWLEDGEMENTS

This research was financially supported by the Natural Sciences and Engineering Research Council of Canada. The production of the SGI and RS/6000 binaries was done at the University of Waterloo. Our thanks to Frank Wm. Tompa and the Computer Graphics Laboratory at the University of Waterloo for allowing us to use their machines.
User’s Guide to *Grail*

Version 2.5

Darrell Raymond¹
Deric Wood²

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

Introduction - 25
Objects - 27
Filters - 28
Minimizing machines - 30
Executing machines - 31
Language equivalence is not identity - 33
Using other alphabets - 33
Generating large machines - 37
An extended example - 38
Implementation - 43
Acknowledgements - 43
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*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*a)(aa*b+bb*aa*a)*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*a(b*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
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, 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:

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

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
```

27
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 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 Grail. More details on individual filters can be found by consulting the appropriate 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
fmderem make a machine deterministic
fmenum enumerate strings in the language of a machine
fmexec execute a machine on a given string
fmmin  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
fmenum renumber a machine
fmreverse reverse a machine
fmstar  star of a machine
fmstats print information about a machine
fmtol convert a machine to a finite language
fmtore  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
- **restar**: 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
- **flrq**: right quotient
- **fltofm**: convert a finite language to a finite-state machine
- **fltofe**: 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 \textit{Grail} filter \texttt{fmmin}. The second method, introduced by Brzozowski, is to reverse the machine, make it deterministic, and repeat these two steps. Using \textit{Grail}, we can show that this procedure results in an isomorphic result:

```bash
% cat dfm
(START) 1- 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 | fmreverse | fmreverse | fmreverse | fmreverse >out2

% isomorph out out2
isomorphic

Brzozowski's minimization technique is implemented by the \textit{Grail} filter \texttt{fmminrev}.

EXECUTING MACHINES

The filter \texttt{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 \texttt{fmexec} to the machine of the last section:

```bash
% fmexec dfm "acd"
accepted

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

\begin{verbatim}
% 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
\end{verbatim}
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 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 | fmdestem | fmmin | >out1
% echo "r(rs+r)*r" | retofm | fmdestem | 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’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 -1 (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]
```

We can complement the machine:

```
% fmcment 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 -| (FINAL)
3 -| (FINAL)
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) 1 - 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:

```
% 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)>*
<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

---

\(^3\) 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:

% fmcomp re
(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:

% fmcomp re | fmtore
% bin/fmcomp remach | bin/fmto re
<bac>(<c(d+e)*><x><ba*>)*(x)<c(d+e)>+(<ab*>+<ba*)((<c(d+e)><x><ba*>)*<c(d+e)>+(<ab*>)+<ba*)((<c(d+e)>)
<x><ba*>))*<c(d+e)>+(<ab*>)+<ba*)((<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:

```
$ cat nfm
(START) 1- 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+private+protected+public+register+return+short+signed+sizeof+static+struct+switch+template+this+throw+
try+typedef+union+unsigned+virtual+void+volatile+while

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

% isdeterm key.fm
nondeterministic

% iscomp key.fm
```

38
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, and 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:

% tr '\012' '+' <keywd | retofm | wc
   353   1059   3876

% tr '\012' '+' <keywd | retofm | fmdeterm | wc
   263    789   2579

% tr '\012' '+' <keywd | retofm | 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.

39
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 accessible 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), 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.
Programmer’s Guide to *Grail*

Version 2.5

Darrell Raymond\(^1\)

March 1996

\(^1\) Department of Computer Science, University of Western Ontario, London, Canada
# Table of Contents

**Introduction** - 47  
**Working with Grail** - 48  
Organization of the files - 48  
Compiling - 49  
Testing - 50  
Filters - 52  
Classes - 53  

**Changing and extending Grail** - 58  
Adding a new Grail filter - 58  
Adding a new alphabet to Grail - 61  
Modifying Grail’s classes - 65  
Miscellaneous - 67  
Changes in version 2.5 - 68  
Changes in version 2.4 - 70  
Changes in version 2.3 - 72  
Changes in version 2.2 - 73  
Changes in version 2.1 - 74  
Changes in version 2.0 - 76  
Changes in version 1.2 - 77
Introduction

This document is about programming with the Grail class library. It describes how to compile, test, and profile Grail, how to write C++ programs using Grail, and how to modify and extend Grail.

If you plan only to install Grail with its standard filters, then you need to read only the first few sections of the document, which describe the organization of the file system and how to go about compiling and testing Grail. It isn’t necessary to know much about C++ in order to use Grail as shipped. If you intend to parameterize Grail’s finite-state machines and expressions, or to write your own filters, then you should read most of the document. In addition, you should ensure that you have a good understanding of templates, since most of Grail’s classes are template classes.

This research was supported by the Natural Sciences and Engineering Research Council of Canada. The author can be reached at drraymon@csd.uwo.ca.
Working with *Grail*

This section is about compiling and testing the distributed version of *Grail*.

**ORGANIZATION OF THE FILES**

*Grail* is a self-contained package organized in the following directories:

- **bin**
  This directory contains the *Grail* filters for a given architecture. Generally, these programs are symbolic links to one of the binaries found in **binaries**.

- **binaries**
  This directory contains subdirectories for specific machine architectures, and compiled binaries for filters for four types of alphabets.

- **classes**
  This directory contains subdirectories for each of *Grail*’s classes. These classes define the objects that *Grail* can manipulate. Most of the source code belongs to classes.

- **doc**

- **man**
  This directory contains *man* pages for *Grail*, suitable for online documentation.

- **tests**
  This directory contains test scripts, test machines, and the expected results for each test.
There are also directories present for each type or class that serves as an alphabet. The distribution provides four different alphabets, and programmers are able to add their own alphabets. The following alphabet directories are in the distribution:

- **char**
  ASCII characters
- **int**
  integers
- **mlychar**
  Mealy machines (with ASCII character alphabet)
- **mlyint**
  Mealy machines (with integer alphabet)

The binaries in **binaries** are labelled **char.out, mlychar.out, int.out,** and **mlyint.out,** corresponding to the filters for a given input alphabet.

### Compiling

Before compiling **Grail,** you need to specify which system and C++ compiler you’re using. In the **Makefile,** you choose between the following:

```
# set SYS to:
# XLC - if you're using IBM's xLC under AIX
# DEC - if you're using USL's Cfront on DEC Ultrix
# SUN - if you're using USL's Cfront under Sun OS
# WAT - if you're using Watcom under DOS
# SGI - if you're using Delta/C++ compiler under IRIX
#SYS=WAT
SYS=XLC
#SYS=DEC
#SYS=SUN
#SYS=SGI
```

49
Uncomment the appropriate SYS variable for the type of system you're using. This will automatically result in choosing the appropriate compiler, compilation flags, and other operating system utilities needed to prepare Grail.

Assuming you have both the source code and the distributed binaries, there are two ways to install Grail. The first method simply installs the binaries that are appropriate for your architecture. Execute one of the following:

```
make sparc
make rs6000
make dec
make sgi
wmake /h /c 486 "MAKE=wmake /h /c" 486
```

No compilation occurs with this technique; it simply constructs symbolic links for each filter to the appropriate existing binary. ¹

If you want different compilation options, or the distributed binaries simply don't work in your environment, then you must compile the code first before installing binaries. You can do this simply by invoking

```
make
```

or

```
wmake /h /c "MAKE=wmake /h /c"
```

if you are using Watcom.

Compilation first constructs a single file from each of Grail's classes, compiles this file (using the compiler designated by the SYS variable), copies the binary to the appropriate binaries subdirectory, and then makes all filters. This process is repeated for each alphabet.

**TESTING**

Grail has its own test system. The test system is useful as a check that Grail has compiled correctly. It's also useful as a preliminary

¹ In the case of 486, separate copies of the binary are made for each filter, since DOS doesn't have symbolic links.
check that modifications you make to *Grail* don’t affect the correctness of its algorithms. *Grail* is tested by doing

```
make checkout
```

or

```
wmake -h /c "MAKE=wmake /h /c" checkout
```

from the root of the *Grail* filesystem. The testing procedure is designed to check the filters designed for an ASCII alphabet against the test objects. Testing scripts execute each filter with each test object as input, and compare the result with a previously obtained result stored in a subdirectory named for the filter; for example, *fmtore* is run against *d1* and the result compared with *tests/fmto*re/*d1*. If the result is identical, the script proceeds to the next test; otherwise, the differences are printed and the whole test result is placed in the directory *errors*. If tests are successfully completed, the following output will be generated:

```
Testing fmcomp on d1
Testing fmcomp on d2
Testing fmcomp on d3
Testing fmcomp on d4
Testing fmcomp on d5
Testing fmcomp on d6
Testing fmcomp on n1
Testing fmcomp on n2
Testing fmcomp on d1
Testing fmcomp on d2
Testing fmcomp on d3
Testing fmcomp on d4
```

(No news is good news.) Some of the tests may put diagnostic messages on the standard error stream (for example, *can’t minimize nfm*) but this is normal output. If a filter fails a test, the difference between the stored result and the computed result is displayed and is saved in the *errors* directory. An error is saved in a file
with the name filter.object; for example, an error when running fmtore on n2 would result in the file errors/fmtores.n2. Comparing errors/fmtores.n2 with fmtores/n2 will help you debug fmtore.

The output of test runs and the stored results are both sorted before comparison. This avoids differences that result only from the order of the output. What it does not avoid is differences that result from language-equivalent but non-identical objects. The testing procedure can detect only non-identical output; it isn't satisfied by language-equivalent results, or even isomorphic results. Thus, if you write a completely new conversion for finite-state machines to regular expressions, for example, you should not expect that your conversion will generate identical results for the test machines (though they should be language equivalent).

The set of test cases includes some boundary cases and a few small examples. We hope to expand the set of test cases in future versions of Grail.2

FILTERS

Grail provides 41 filters that can be used like any other command available at shell level. In previous versions of Grail, each of these filters was represented by a separate source code file and a separate executable. Structuring the filters in this way led to very long compile times, since some compilers re-instantiate the templates for each filter. Another problem with this approach is that the filter code itself was duplicated many times.

In Version 2, we've taken a different approach. All filters for a given alphabet are implemented by a single executable. This executable determines which function to apply by checking the name by which it was invoked. If the char.out executable was invoked with the name fmdeperm, for example, then it would execute the conversion to deterministic machines. The advantage of this technique is that it's easier and faster to copy or rename a file than to recompile it. This is particularly true for the current version of Grail, which makes extensive use of templates.

2 Note that we don't yet have fmexec in the test suite; this may explain why we've shipped buggy versions of fmexec in the past!
array
  list
  set
  string
bits
fl
fm
inst
pool
re
state
subexp
cat_exp
empty_set
empty_string
plus_exp
star_exp
symbolExp

Table 2.1: *Grail*’s class hierarchy.

Under Unix, each of the individual filters in Version 2 is actually a symbolic link from *bin* to the appropriate executable in *binaries*. Using symbolic links eliminates the cost of storing multiple copies of the files. Under DOS, multiple copies of the executable are used.

**CLASSES**

Version 2.5 of *Grail* employs 18 classes, organized in a relatively flat hierarchy shown in Figure 2.1. The main classes are *fl* (finite languages), *fm* (finite-state machines), and *re* (regular expressions). These classes define the capabilities that make *Grail* useful for symbolic computation with machines and expressions.

There are two types of support classes. The first type implements the basic container classes *set*, *list*, *array*, *string*, *bits*, and *pool*. In *Grail*, lists, sets, and strings are all forms of *array*. *bits* manages bitmaps, and *pool* manages a memory pool. The
second type of support class implements substructures of the main classes; state implements the states of a finite-state machine, inst implements the instructions of a finite-state machine, and subexp implements the subexpressions of a regular expression.

subexp is an abstract base class for the set of possible subexpressions. These include the empty set (empty_set), empty string (empty_string) single-symbol expression (symbol_exp), catenation expression (cat_exp), union expression (plus_exp), and Kleene closure expression (star_exp).

With the exception of state, all of Grail’s classes are templates that are instantiated for a chosen type or class. Grail thus provides wide flexibility in designing and executing machines.

Here are some general comments about the design of the classes:

- All assignment and copying is deep; that is, the whole substructure of an object is duplicated. None of Grail’s structures point to shared data. There is no reference counting.

- There are no iterator classes. Utilities that want to iterate through a set or a list simply use a loop over the selection operator.

- No implicit casts have been defined, and the number of copy constructors (which act like implicit casts) is severely limited. This has been done to ensure the strictest possible type checking.

Here are some comments about technical points of the design of the classes.

fm Internally, fms are stored as three sets: a set of start states, a set of final states, and a set of insts.

fm contains operations for ‘disjoint union’. These can be used for fast union of machines that are known to be disjoint. The standard union operator (operator+=) tests for membership before adding, while the disjoint union does not. It is the programmer’s responsibility to check for disjointness.

fm contains operations for ‘selecting’ instructions based on their states or labels. These operations will in future be moved to
a class relation that will support general-purpose project, select, and join operators.

**re** Why isn’t fmtore a member of fm, rather than of re? fmtore operates on an \texttt{fm<\text{S}>} and generates an \texttt{fm<\text{re}<\text{S}>}; if it was made a member of fm, it would result in an infinite template instantiation (the generated \texttt{fm<\text{re}<\text{S}>} would itself be a target of fmtore, generating an \texttt{fm<\text{re}<\text{re}<\text{S}>}}, that would itself be a target of fmtore...).

**state** States in a finite-state machine are non-negative integers. The class state shifts all integers by 2, to ensure that 0 and 1 are available to represent the start and final pseudo-state, respectively.

**inst** looks for the pseudo-labels \texttt{|-} and \texttt{-|} on its input, and generates them on output, but does not represent them internally.

**array** is the basic data structure. lists, sets, and strings are all derived from array, with small differences that are due to the different update constraints required by each structured. Generally speaking, sets are unordered and do not have duplicates; lists preserve their order and may have duplicates; strings preserve their order, may have duplicates, and can be compared with a \texttt{strcmp}-like function. There are efficient conversion operations from list and from set that simply adjust the array pointers (and in the case of conversion from list, removes duplicates); these conversion routines do not preserve the original list or set.

array includes a merge function that can be used to quickly merge two sorted arrays, and produce a sorted result. This function relies on the programmer to ensure that the original arrays are sorted.

**list** defines a static comparison function that can be passed to qsort.

**set** contains operations for ‘disjoint union’. These can be used for fast union of sets that are known to be disjoint. The standard union operator (\texttt{operator+=}) tests for membership before
adding, while the disjoint union does not. It is the programmer’s responsibility to check for disjointness.

**string** in **Graal** is not a **char**. Even a **string<char>** is not a **char**, since it isn’t null-terminated. It’s necessary to append a null character to a **string<char>**’s content if you intend to handle it with functions such as **strcmp** or **printf**.

**string** defines a function **ptr()** which returns a **char** pointer. This is a trap door for potential problems, since the array can be arbitrarily modified without the **string** object adjusting its size and maximum value. Use this capability only for operations that do not perform update to the array.

The **string** comparison operators are defined so that strings will be ordered first by size, then lexicographically within equal sizes. This differs from the usual ordering, but is more appropriate for dealing with languages, where we typically want to see the shortest words first.

**subexp** A **subexp** is the virtual base class for the recursive definition of regular expressions. A regular expression contains one subexpression, which may be one of **empty_set**, **empty_string**, **symbol_exp**, **cat_exp**, **plus_exp**, or **star_exp**. The latter three subexpressions are themselves made up of subexpressions.

One interesting problem for subexpressions is defining their comparison operators. Individual subexpressions are ordered according to the following precedence:

- **empty_set** < **empty_string** < **symbol_exp** < **plus_exp** < **cat_exp** < **star_exp**

Hence, **empty_string::operator>(const empty_set<S>&)** should return 1, since empty string expressions are always greater than empty set expressions. We cannot simply compare the content of subexpression pointers, however, since function arguments are interpreted according to their apparent type, not their actual type. Each subexpression therefore defines a set of functions of the form **compare_xyz_exp**. This function determines how a given subexpression compares to an **xyz** expression. In effect, we are using two function calls (the opera-
tor and the \texttt{compare} function to determine the actual types of both arguments to the comparison operation. This technique is called \textit{double dispatching}.

Most subexpressions define a \texttt{new\_subexp()} function, which is the actual constructor. This function is defined because it is not possible to have virtual constructors. Similarly, the functions \texttt{copy} and \texttt{clone} are defined to provide the effect of a virtual constructor. See p. 217 of Stroustrup’s \textit{The C++ Programming Language, 2nd Edition} for more information.

\texttt{star\_exp} overloads the \texttt{star} operator of \texttt{subexp} and defines it as a no-op. This has the effect of ensuring that a ‘starred’ expression is only starred once.
Changing and extending *Grail*

There are two basic ways to modify *Grail*: you can add a new alphabet, or you can add some new functionality that’s alphabet-independent. The latter method typically results in a new filter.

**ADDING A NEW *Grail* FILTER**

A new filter for *Grail* may simply combine existing *Grail* functions, or it may include new functionality that you add to one or more of *Grail*’s classes. As an example, let us suppose you have discovered a new operation on machines that you call ‘squeezing’, and you want to add a filter that ‘squeezes’ a machine.

The first task is to write up the algorithm as a member function of the class *fm*. You might put this in a file `classes/fm/squeeze.src`. Note that we use the `.src` suffix, rather than `.C` or `.cpp`, because we don’t compile routines separately; instead, all the `.src` files will be catenated together to make up one file describing `fm`. `squeeze.src` will make use of existing functions in *fm*, and it will probably also use other data structures in *Grail*, such as *sets* and *lists*. You needn’t worry about including any header files if you only use other *Grail* classes, since they are all (eventually) provided for you.

The second task is to ensure that `squeeze.src` will be included in the compilation of *Grail*. You do this by making sure that `squeeze.src` is listed in the file `classes/fm/include.h`.

The third task is to arrange for a ‘squeeze’ filter to be produced when *Grail* is compiled. This involves several steps:

1. Add the necessary code to invoke `fm::squeeze` to `char/grail.C`.

   `grail.C` is essentially a large case statement that selects the action to be executed based on the value of its name that was used to invoke the program; that is, based on the value of `argv[0]`. Simplified, `grail.C` looks like this:

   ```c
   # Even this file will not be separately compiled; since this file describes a template, the compiler can’t do much without a type parameter.
   ```
main(argc, argv)
{
    .
    if (strcmp(my_name, fmcment) == 0)
    { // do complement operation  }
     .
    if (strcmp(my_name, fmcat) == 0)
    { // do catenation operation  }
     .
    if (strcmp(my_name, fmenum) == 0)
    { // do enumeration operation  }
     .
}

The variable my_name is initialized to argv[0]. To make a ‘squeeze’ filter, you would add something like:

    if (strcmp(my_name, fmsqueeze) == 0)
    {  
        get_one(a, argc, argv)
        a.squeeze();
        cout << a;
        return 0;
    }

Here the programmer arranges for fmsqueeze to be the name of the filter. If the executable is called with this name, then it will enter the body of the if statement. The function get_one is a utility function that obtains the input machine; it will get input either from a file or from standard input (if ‘squeezing’ was a binary operation, you would use the utility function get_two to get two finite-state machines as arguments. The input machine is stored in a; the function squeeze is called, the squeezed machine is printed on standard output, and the filter returns.

2. Define the strings that will used to name the filter’s file.
**fmsqueeze**, the second parameter to the `strcmp` in `char/grail.c`, is not a string but a variable pointing to a string. This variable is initialized to different strings for different operating systems. Under DOS, it points to an uppercase name with a `.EXE` extension, and limited to 8 characters. Under UNIX, it points to a lowercase name with no extension and not limited to 8 characters. In `char`, you will find files `names.h` and `dosnames.h` that define the names to be used for each filter. You must add a definition for `fmsqueeze` to each of these files.

3. Repeat the previous two steps for `int`, `mlychar`, `mlyint`, and any other alphabets that your version of `Grail` supports.

4. Add a line to the main `Makefile` to create a symbolic link from `bin/fmsqueeze` to the executable `binaries/*/char.out`.

   This step must be performed for every machine architecture you want to support.

To fully integrate your filter with `Grail`, you should also add it to the test directory. To add the filter to the test directory, you need to do the following:

- Make a directory `tests/fmsqueeze`. This is where pre-computed results of testing are kept.

- Modify `tests/Makefile` to run `fmtest` (or `fm2test`, if your filter takes two arguments) on your filter.

- Run your filter on each of the test cases and carefully check the output. If you're certain that the results are correct, then store the output for each test case in `tests/fmsqueeze`. (If you're not certain that the output is correct, then by storing the output all you're doing is giving future testers a false sense of confidence.) The result of ‘squeezing’ `dfm1` should be in `tests/fmsqueeze/dfm1`, the result of ‘squeezing’ `dfm2` should be in `tests/fmsqueeze/dfm2`, and so on.

- If you need to add some new test machines to test special conditions (for example, an ‘unsqueezable’ machine) for your filter, it would be useful if you also run all the other filters in
Grail on this test case, check their results, and add the output to the respective directories. This practice will increase the value of the test system for the whole of Grail.

- Write a man page for your new filter.

Adding functionality may seem too complicated. The only excuse we can offer is that when you have an environment that attempts to support multiple architectures, operating systems, and alphabets, there is going to be a lot to worry about.

ADDING A NEW ALPHABET TO Grail

Adding a new alphabet can be simpler than adding new functionality (we emphasize ‘can’—it may not be!). If your type or class is well specified, and you have a modern compiler, then almost all of the work will be done for you, and all of the functionality of Grail will be carried over to your parameterized class.

Parameterizing over a base type

Suppose you want to create finite-state machines whose instruction labels are instances of int. The following steps are necessary:

1. Do a recursive copy of the directory char (or some other directory for an existing alphabet type) to a new directory int.

2. Edit int/grail.C.
   Change all variables of type fm<char> to fm<int>.
   Change all variables of type re<char> to re<int>.
   Change all variables of type fl<char> to fl<int>.
   Change all variables of type string<char> to string<int>.

3. Edit int/lexical.h.
   You need to define lexical delimiters that will be used to input and output machines and expressions of type int. The following delimiter variables need to be defined:
There is one instance of each of these variables per parameterized class; so, there is one `re<char>::re_star`, one `re<int>::re_star`, and so on. These variables are provided to permit you to define your own symbols, either because you prefer some other delimiters or because one or more of the defaults is a valid symbol in the input alphabet you want to use.

Note that the default symbol for concatenation and the left and right delimiter are both 0. If these values are specified for these variables (only), then no output is generated for those symbols.

4. Edit `int/names.h` and `int/dosnames.h`.

You need to create names for all the executables that satisfy the constraints that the operating systems impose on filenames. You may also wish to create names that are distinguishable from all other *Grail* executables or other programs you use (alternatively, you can have several directories for filters, and change your search path to use only the ones appropriate for a given project).

5. Edit `int/Makefile`. Change all the executable names to be the same as those you used in `int/names.h` and `int/dosnames.h`. The single binary file should also be changed to the name of your type (`char.out` should become `int.out`).

6. Edit the root `Makefile`. Add a compilation statement with `TYPE=int`. Add `install` statements for each architecture for `int`. Add `int` to the `make clean` command.

62
7. Compile *Grail* (which, if you've done the previous steps correctly, will compile all types and install all filters).

Remember that using a template inside a template is permitted, but you must leave a space between end-brackets. That is,

```
fm<re<char> >
```

is valid, but

```
fm<re<char>>
```

is not (the C++ parser thinks that >> is the ostream operator, not the end of the template specification).

**Parameterizing over your own classes**

Parameterizing over your own classes or types is much the same as parameterizing over base types or *Grail* types. The main difference is that the *grail.C* file must be able to find the class definition and its member files. Typically this is done by copying them to the directory for that alphabet, and putting an `#include` statement in *grail.C*

There are two problems that may arise with parameterization of your own classes.

The first problem is the provision of minimally required functions and operators. *Grail*'s templates (like those of any other C++ class library) operate on the assumption that certain functions are defined by the type used for parameterization. There is no way for us to arrange that you define these functions, but if they aren't defined (or if you define them ambiguously), then your compilation will fail at template instantiation time. We require that you define a small number of operators:

---

2 For classes that you only need to link, you are only required to make the class header accessible; the compilation command should be altered to include the necessary linking directive to locate your class binary.
If you have defined these operators for your type, it should instantiate without trouble.

Even if all necessary operators are defined, you may misinterpret the results of Grail's operations. To understand this problem, let's look at `fm<re< char>>` in some detail.

There are at least two possible ways to define the `==` operator for `re<char>`. One way, based on identity, treats two `re<char>`s as equivalent if they are identical. The second way, based on language equivalence, treats two `re<char>`s as equivalent if they denote the same language. In general, the only feasible way to determine language equivalence for regular expressions is to convert them to finite-state machines, minimize the finite-state machines, and test the minimal finite-state machines for identity. This test is an expensive proposition, so there is some motivation for choosing to base equivalence on identity.

Grail, of course, has no way of knowing which choice you have made; indeed, the whole point of parameterization is that it should not need to know which choice you have made. Grail simply takes it for granted that the operator `==` will return positively if the two regular expressions are equivalent, and negatively otherwise. But your choice of semantics for `==` will affect the outcome of Grail's operations. `==` is used in subset construction, for example, to cluster all states which are reachable on the same instruction label. If you've defined language equivalence as your semantics, then Grail will treat the regular expressions `a` and `a*a(a+a)` as equivalent; if you've chosen identity as your semantics, then Grail will treat these two expressions as distinct. Thus, the two semantics lead to different output.

Parameterization allows Grail to implement a collection of functions that are performed on 'black boxes', which you can instantiate with a type. Grail will provide correct results, but only within the
semantics you defined for the operators of that type. If you choose to define identity semantics, don’t expect to get language equivalence semantics in the result.

The same is true of the semantics of the other comparison operators <, >, and !\text{!}.

MODIFYING Grail’s CLASSES

Modifying Grail’s classes can be straightforward, but it requires a good understanding of three complicated areas: C++ templates, Grail’s existing structure, and the theoretical properties of finite-state machines and regular expressions. Here are some points to remember:

1. Maintain the separation between a class’s interface and its implementation. The class \text{fm}, for example, is implemented as two sets of \text{states} and one set of \text{insts}, but this should not be visible outside the class. As much as possible, ensure that the interface is restricted to logical functionality.

2. Remember that your new function must work regardless of the type of the instruction label (or, for regular expressions, of the symbols of the alphabet). Do not make assumptions that are true only of fixed types. Is your function general enough to apply to a \text{fm<re<fm<set<string> >>} ? If not, should you rethink the function?

3. Remember to run the tests on all Grail filters after you have made your modifications.

4. If you create important new functionality, consider making it available through a separate filter. Follow the procedure that we described in the section on making filters.

It would be convenient if your additions to Grail are consistent with the set of conventions Grail uses for filenames. We use two-letter prefixes for filters. Regular expression filters use the prefix \text{re}. Finite-state machine filters use the prefix \text{fm}. Finite language filters use the
prefix \$. We also use these prefixes as suffixes for commands that convert from one type of object to another; for example, `retom`. \(^3\)

Each class directory has a file `classname.h` that contains the class declaration. The `string` class, for instance, is declared in the file `string.h`. This is the first place to look for information about the class, since it contains declarations of all the methods.

Each of the functions defined for a class is contained in a separate `function.src` file. When the function is a function call with an alphanumeric name, its filename is the same name (for compatibility with non-flexname file systems, long function names are shortened to fit an 8-character limit). Hence, the function `parse` in the class `re` is located in the file `parse.src`. Since operator functions don’t have alphabetic names, we’ve chosen to use the following standard alphabetic names for operators:

```plaintext
<< ostrstream.src
>> istream.src
< lt.src
> gt.src
== eq.src
!= neq.src
+= pluseq.src
*= timeseq.src
-= minuseq.src
&= concat.src
+ plus.src
- minus.src
[] index.src
```

We use `classname.src` for constructors and `~classname.src` for destructors. Constants, macros, and types that are specific to a class are kept in `defs.h`. The set of system and local files that are necessary for compilation of functions are specified in `include.h`.

\(^3\) All predicates begin with the prefix `is`. This is likely to be changed in the future, because it does not distinguish between predicates for machines and predicates for expressions, and because ‘is’ is not the only type of predicate we want to support.
MISCELLANEOUS

Some odds and ends:

1. Why do we use the suffix `.src` for our class code files? Because too many compilers make invalid inferences from suffixes like `.c` or `.C`. In some cases the compiler decides that the code is C rather than C++; in other cases, the compiler's template instantiation mechanism thinks that a `.c` file with the same prefix as the template's `.h` file must be the template definition file. Many C++ compilers allow you to specify your own suffix with a command line option, but their template instantiation mechanisms do not always use this information. Consequently, we use a suffix that no one expects, `#include all the files in to a single class module`, and use that as (part of) the compilable object.

2. Why do we include all files in Grail in one single, monolithic module for compilation? In our experience, this is the fastest approach to compiling Grail. Multiple modules mean multiple invocations of the compiler, with redundant processing of many common header files. Another reason is that some C++ compilers use the source filename to construct an external entry point for the destructor function for each class; this has led to linking problems if the same filename is used for some other class. The third reason is Grail's heavy use of templates. With some compilers, separate compilation of templates involves a costly process in which each failure of the linker to locate an instantiation of a needed template function causes the compiler to be invoked to generate that function. Separate compilation of Grail in such an environment can take over an hour. By producing a single module, we completely avoid the interaction between linker and compiler, and we have seen our compile times drop to about five minutes.

3. The class headers include an `#ifdef` to ensure that every class is defined only once. This hack should be avoidable by proper use of the `#include` facility, but it doesn't seem possible (the problem may be due to how template instantiation works).
4. The classes derived from `subexp (empty_set, empty_string, cat_exp, plus_exp, symbol_exp, and star_exp)` are accessed only within `re`, and indeed should not even be visible outside `subexp`. Why then are these derived classes not nested within `subexp`? The reason is that some compilers don’t implement nested classes within templates.

5. Why haven’t we made `Grail` work with GNU C++? The main reason in the past was GNU’s non-standard behavior and poor template support. The commercial compilers are better and more reliable than GNU, at least for the moment.

6. Some notes on compilation: On most UNIX machines `Grail` compiles in two to three minutes, depending on load and compilation options. IBM’s `xlc` on the RS/6000 550 and SGI’s Delta compiler on the Onyx/2 are the fastest environments we’ve used. `xlc` and Watcom 10.5 are tied for most robust compiler; each is able to find errors that the other one won’t, and both are much more strict than cfront-based compilers.

   If you’re using Watcom 10.5, we strongly recommend that you do most of your compiling without the optimization flags `-oneatx`. Without these flags, `Grail` compiles in about two minutes on a 90 MHz Pentium with 16 Mbyte of EDO RAM; with optimization enabled, compiling takes as much as an hour.

**CHANGES IN VERSION 2.5**

This section describes the changes and improvements made since Version 2.4.

1. Added test for self-assignment to `bits::bits(const bits&)`.

2. Plugged memory leak in copy constructors for `cat_exp, plus_exp, star_exp`.

3. Removed `copy()` function member from `re` classes.
4. Custom memory allocation for array class.

5. pool class written, and re's allocation done by pool.

6. Batch copying in array::operator+=(const array&).

7. 'Destructive' copying in array::operator+=(array*).

8. Used a bitmap to manage sets in fm:states().

9. Fixed "make clean" in tests/Makefile.

10. bits::next() written.

11. bzero, bcmp, bcopy used instead of loops in various functions.

12. array:swap() written

13. pool class written.

14. Memory leaks in re fixed.

15. re's parsing is now linear instead of quadratic (only one call to istrstream).

16. Overgenerous memory allocation in array::operator=() fixed. (only allocate sz, not max)

17. Variables in grail.C allocated at time of use, not at start of procedure.

18. fmminrev need not take deterministic input.

19. Old subset construction restored for machines with more than 1000 states.
20. array assignment allocates only for sz, not for max, of the argument (much space was wasted if local array variables were large).

21. null_exp removed from re.

22. fl class added.

23. install_unix/install_dos dichotomy removed.

24. makefile now uses wmake and DOS commands instead of MKS make and Korn shell.

25. fm::min_by_partition() and fm::enumerate now remove unreachable states. Thanks to Makoto Murata for bug reports.

26. fm::member now properly handles empty strings.

27. re_lambda and re/std.h removed as unused. Thanks to Wolfgang Frech for bug reports.

28. retofm restored to mlychar and mlyint. Thanks to Wolfgang Frech for bug reports.

CHANGES IN VERSION 2.4

This section describes the changes and improvements made since Version 2.3.

1. string, list, and set classes now derived from array class.

2. Protected assignment operator in subexp.

3. More extensive use of initialization lists in various classes.
4. Added array::unsorted. This is needed if array members are changed by an external object, as in fm::reverse.

5. No more need for LIST_SIZE and SET_SIZE defaults.

6. Bitmap class written and extensively profiled.

7. Bitmaps used in fm::subset

8. Added test for argc before grail.C's call to fmenum

9. Template pair class, mealy class added. Old pair class (non-template, fixed elements) removed.

10. mlyint, mlychar directories created.

11. String class given an explicit separator (i.e., a catenation operator symbol). */grail.h, */lexical.h modified to define default and explicit separators.

12. Many needlessly friendly iostream functions made external to the classes they support.

13. Redundant constructors for set, list, and string eliminated.

14. Erroneous calls to destructors removed from re classes; deletes used instead.

15. Memory-leaking constructions for cat_exp, plus_exp, star_exp, and null_exp plugged.

16. Static variables removed from class re.

17. Fixed up the file headers (we're not so
university-specific now).

18. Makefiles now move *.out into binaries directory on invocation of compile, not install.


20. #ifdef for bits/set.src under cfront.

21. Ran Purify to test for memory access errors.

22. Ran Quantify to test for gratuitous inefficiencies.

CHANGES IN VERSION 2.3

This section describes the changes and improvements made since Version 2.2.

1. Fixed string/istream.src (again).

2. Added xfmenu, xfexec, xfmcross, xfmmin, xfmminrev, xfmcmment, xfmcomp.

3. Fixed bug in fm/catenate; was removing self-loops on start states of argument machine.

4. Fixed bug in retofm; was not incrementing state number high enough (and thus generating self loops on null expression).

5. Fixed bug in fm/catenate; was not making final state of invoking machine final if argument machine included empty string.

6. Added specialized set deletion that substitutes last element.
7. Split xfm stuff into separate directory; now using directory 'type' instead of 'grail' (e.g., 'char', 're', 'pair', etc.)

8. Added sorting and tests for sortedness to set.

9. Added state::operator=(const state&).

10. Substituted initialization for assignment in constructors (Myers #12).

11. Return value of empty_set<S>::operator() was subexp<S>&; changed to empty_set<S>& (Myers #15).

12. fm data members made private.

13. minor improvements made to re::fmtore

14. -n flag added to fmenu.

15. Makefiles improved for multiple architectures, multiple compilers, multiple alphabet types.

Changes in Version 2.2

This section describes the changes and improvements made since Version 2.1.

1. New array class; set, list, and string are derived from array.

2. Removed classes/Makefile and classes/*/Makefile; instead, we use #include and compile everything in one shot (thus avoiding long template instantiation and makefile differences across systems).

3. fm altered to save start and final states explicitly.
4. Redundant class members removed; small functions inlined; classes generally cleaned up.

5. Removed grail/template.2 (not necessary with new #include style).


7. Fixed bugs: fmexec did not handle 4-argument case correctly; string/istream.src read last character twice. Thanks to Jochen Seemann of the University of Wurzburg

8. Added flags for static binding.

9. Fixed profiler to use proper filter names (null profiles were being generated because filters had .pixie suffix).

10. Added .EX, .EE macro definitions to top of each man page.

11. Included Rational DOS extender.

CHANGES IN VERSION 2.1

This section describes the changes and improvements made since Version 2.0.

1. Fixed bug in handling of istrstream for fmexec arguments in fm.C. Thanks to Tillman Kolks of IMEC, Belgium

2. Change loop index variable to "j" where "i" was being used twice in nested loops in fm::enumerate. Thanks to Tillman Kolks of IMEC, Belgium

3. Fixed bug in min_by_partition; machines consisting of only
final states should not be reduced to single-state machine.
Thanks to Tillman Kolks of IMEC, Belgium

4. Made sure fmrenum does not include unreachable states.
Thanks to Tillman Kolks of IMEC, Belgium

5. All classes/*.cc files moved to classes/*.src files, and
   Makefiles converted correspondingly. This change made
   to support Sun CC template instantiation.
Thanks to Scot Dyer, University of Nebraska-Lincoln

6. -c argument to fmenum in grail/fm.c fixed.
   Thanks to Tillman Kolks of IMEC, Belgium

7. Missing return statements added to grail/fm.c, grail/re.c, and
   grail/fmre.c.

8. inst::operator== changed to eliminate label test for start and
   final transitions. This necessitated changes to re::fmtore to
   handle regular expressions on start and final transitions.

9. grail/names.h and grail/dosnames.h added to permit compilation
   under DOS.

10. Makefile.wat added to various directories, for compilation under
    Watcom C++ 9.5.

11. Changed argv[0] usages to my_name in grail/fm.c, grail/re.c,
    grail/fmre.c. Made executable name extraction work with both
    Unix and DOS-style path delimiters.

12. Fixed bug in fmstar (added too many final/start instructions
    to clone state).

13. Added test cases d7, d8. Renamed all test cases to work within
    DOS-style file suffix limitations.
CHANGES IN VERSION 2.0

This section describes the changes and improvements made since Version 1.2.

1. Converted fa and trans to template classes.

2. Removed tset and xfa.

3. Cleaned up directories and files.

4. #ifdefs used to avoid duplicate definitions of classes (seems to be required by template instantiation mechanism)

5. fa filters are all now symbolic links to one executable that checks argv[0] to determine which operation to perform.

6. state::number made private.

7. Fixed trans comparison operators to avoid checking labels for pseudo-transitions.

9. Removed fa::operator+(trans&) (it had different semantics from fa::operator+(fa&), which could be confusing).

10. Filters renamed to use "fm" prefix; fixed test cases.

11. isomorph does its own renumbering and sorting now.

12. Renamed "fa" class to "fm"; renamed "trans" class to "inst", "regexp" class to "re".

13. re class rewritten; new classes: empty_set, empty_string, cat_exp, plus_exp, star_exp, symbol_exp, subexp.

14. re filters are all now symbolic links to one executable that checks argv[0] to determine which operation to perform.
15. xfm filters are all now symbolic links to one executable that checks argv[0] to determine which operation to perform.

16. Made string parameterized; altered usage of string where necessary to string<char>.

17. Rewrote retofm and fmtore.

18. Added various hacks to enable proper template instantiation (grail/template.1, grail/template.2, note changes in re.h)

19. re now does not automatically "minimize" expressions; remin has the "minimization" functionality.

CHANGES IN VERSION 1.2

This section describes the changes and improvements made since Version 1.0.

1. Compiles under x1C 1.00, AT&T 3.0, Watcom C++ 9.5.

2. Added set>gt.cc and set<lt.cc.

3. string::operator+= reallocation changed so that blocks are always a power of 2. This seemed to fix a bug when running fatore on RS/6000.

4. In string.h, fa.h, state.h, grail.h, use <iostream.h> instead of <stream.h>.

5. Removed "form" from regexp\concat.cc, regexp\term.cc, regexp\token.c.

7. Removed duplicate xfaplus from grail/Makefile.

8. Improved grail/Makefile to use default rules, removed unnecessary operations.

9. Added "tempinc" to clean targets so that x1C recompilation proceeds correctly.

10. set/include.h and list/include.h designed to handle the default requirements of x1C/Cfront template mechanisms (for x1C, you include the template header file, for Cfront, you don't).

11. Added "XLC" and "ATT" defines to Makefile, tset.h.

12. "delete [] p" removed from "tset(). It incorrectly duplicates the functionality of "set(), causes a crash under Watcom 9.5 (discovered by Mark DeLaFranier of Watcom).

13. mksys scripts written for list, set (to provide correct suffixes for x1C and Cfront).

14. Removed <libc.h>, substituted <stdlib.h>.

15. All grail filters given "return 0" at end of main; all return values checked (and modified) for correctness.

16. from_set and from_list made members of list and set respectively.

17. find_part removed from xfa.h.

18. list::compare() only; removed compare from all other classes; compared contents of pointers instead of pointers.

19. list::< and list::>.

20. Removed print functions from set, tset, list; redefined
ostream operators.

21. converted Item::compare to list<Item>::compare in list::sort

22. note that tset::operator<< second argument must be const.

23. famin fixed; can't treat min_by_partition result as boolean.


25. For nfa's, faenum computes deterministic density and converts to deterministic automata if appropriate.

26. Purify'd. Fixed bugs in string::operator+=(const char*) and ostream::<<(ostream&, regexp&).