Grail: Engineering Automata in C++
Version 1.0 *

Darrell R. Raymond † Derick Wood ‡

January 1, 1993

*This research supported by grants from the Natural Science and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.
†Department of Computer Science, University of Waterloo, Waterloo, Canada
drraymond@daisy.uwaterloo.ca
‡Department of Computer Science, University of Western Ontario, London, Canada
dwood@csd.uwo.ca
Contents

1 Introduction 5

2 A short history of Grail 6

3 Design 11
   3.1 Abstract Objects: Automata and regular expressions 12
   3.2 User level 13
   3.3 Programmer interface 14

4 Software organization 17
   4.1 System directories 17
   4.2 Classes 18
   4.3 A class directory 18
   4.4 Making the code 20
   4.5 Test directory 20

5 Miscellaneous information 20
   5.1 How do I obtain Grail? 20
   5.2 Related software systems 21
   5.3 Future work 22
   5.4 C++ references 22
   5.5 Organizational quirks 23

6 References 24

A Finite automata: \texttt{fa} 26
   A.1 Definition 26
   A.2 Public functions 26
   A.3 Friend functions 26

B Regular expressions: \texttt{regexp} 27
   B.1 Definition 27
   B.2 Public functions 27
   B.3 Friend functions 29
   B.4 Private functions 30

C Sets: \texttt{set} 31
   C.1 Definition 31
   C.2 Public functions 31
C.3 Friend functions .......................................................... 32

D States: state ................................................................. 33
   D.1 Definition ............................................................. 33
   D.2 Public functions ..................................................... 33
   D.3 Friend functions .................................................... 35

E Strings: string .............................................................. 36
   E.1 Definition ............................................................. 36
   E.2 Public functions ..................................................... 36
   E.3 Friend functions .................................................... 37

F Transitions: trans .......................................................... 38
   F.1 Definition ............................................................. 38
   F.2 Public functions ..................................................... 38
   F.3 Friend functions .................................................... 40

G Sets of transitions: tset .................................................... 41
   G.1 Definition ............................................................. 41
   G.2 Public functions ..................................................... 41
   G.3 Friend functions .................................................... 43

H Extended finite automata: xfa .......................................... 44
   H.1 Definition ............................................................. 44
   H.2 Public functions ..................................................... 44
   H.3 Friend functions .................................................... 45
1 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, 32/6

We can’t go doddering across Malaya behind an inspired crackpot following the Holy Grail, can we?
H.M. Tomlinson, *Gallions Reach*

Grail is an ongoing project devoted to the production of efficient, modular software for managing finite automata, regular expressions, and other formal language theory objects. Our goal is to develop software that is both more powerful and more extensible than systems like *lex*, *yacc*, and *grep*. Simultaneously, we hope to make Grail much more accessible than these systems; we intend to make it highly modular and easy to use for teaching purposes. This report presents the design and organization of Grail, and serves as an introduction to the modules. We describe the modules in sufficient detail to enable programmers to use them, but we do not discuss the details of the data structures and algorithms that underlie the modules.

Grail is written in C++. 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. C++ has led to much better reuse and robustness, but not because of the class hierarchy. Instead, we have found that C++’s strict type checking and encapsulation have been the most important contributors to better code.

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.

2 A short history of Grail

The study of formal language theory has a long history at the University of Waterloo. Fundamental contributions have been made over a long period, especially Brzozowski’s work on regular expressions and Wood’s work on grammars and L-forms. The implementation of grammars and automata has also been essential, both in influencing theory and in supporting practical applications, such as the tag enhancement of the electronic text of the Oxford English Dictionary. Probably the earliest software at the University of Waterloo was Leiss’s REGPACK[12], a package written in 1977 to support experimentation and research with finite automata. REGPACK, written in SPITBOL, supported the conversion of nondeterministic automata to deterministic automata, minimization of deterministic automata, and construction of syntactic monoids. While REGPACK did not directly influence the current effort, it is interesting to note that we are still pursuing REGPACK’s goal of the production of an environment for experimentation with automata.

A more recent package with direct influence on Grail was Howard Johnson’s INR[9]. INR was developed because of Johnson’s interest in rational relations and their use in defining string similarity[8]. INR takes rational relations (including regular expressions) as input and converts them into finite automata, which can then be manipulated in various ways. INR can produce single- or multiple-tape automata; 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 automata. His goal was the effective processing of automata with thousands of states and transitions. As a result, INR is written very compactly in C, uses its own memory allocation scheme, and is especially efficient in handling potentially costly tasks such as subset construction and minimization. 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 took the trouble to develop efficient implementations, with the result that INR was the only software system that was capable of handling the transduction of the *Oxford English Dictionary*[10]. Even today many of INR’s capabilities are more advanced than those of other automata software (though we like to think that Grail is catching up). The present effort has borrowed heavily from INR, adopting its philosophy of combining powerful capabilities with efficient design, as well as its notation for automata.

The first project to actually use the name ‘Grail’ was a joint effort between Howard Johnson, Carl-Johan Seger, and Derick Wood. This was an attempt to extend INR to handle context-free grammars and automata with regular expressions as transition 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.

Several years later, the Grail project was resuscitated by the present authors. 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 written in a nonstandard C that was neither easily maintainable nor easily 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 routine 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 improving 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 end use. Tools like yacc and lex were built for compiler writers, while INR was built as part of a research project. We wanted to add a third type of use, namely, pedagogical. The algorithms for automata and grammars are
routinely taught to computer science students, but the implementations are less often considered a fit subject for study. One reason is that there is no existing software environment that supports short programming assignments in formal language theory (whereas there are such environments for databases, operating systems, and numerical analysis). We intended Grail to be such a software environment. The second reason is that language theory is usually taught by theoreticians whose primary interest is not the development of efficient software. Building Grail has given us a healthy respect for the difficulty of implementing some ‘well-known’ solutions, and an interest in teaching formal language theory by combining its theoretical concepts with sound engineering principles. We hope that Grail will serve as a proof of concept of this approach to formal language theory, and indeed, encourage others to take a similar approach in other areas.

Grail’s modularity can be beneficial to teaching in two ways. The most obvious way in which it is beneficial is that it permits students to explore small parts of the code without needing to understand the whole; this is just the flip side of the software maintenance advantage. The second beneficial aspect, however, is that the modularity encourages the development of many solutions to a given problem, not just the most optimal one. This aspect is important because the teaching of algorithms typically involves the presentation of many alternatives; sorting, for example, is learned by the study of different characteristics of several sorting algorithms, not just the study of quicksort. Hence, unlike INR, Grail is designed to support different implementations of a given function.

Multiple implementations also turn out to have other unexpected uses. One of the most important is testing. A manual check of the correctness of subset construction (as an example) for an automaton of fewer than five states is quite easy to do, but a manual check of an automaton with tens or hundreds of states is quite another matter. Multiple implementations allow us to compare output as a first check on correctness.

Pedagogical use should extend beyond the simple problems addressed at the undergraduate level; we wanted Grail to be capable of addressing problems of research interest. Thus, we want Grail to be a symbolic computing environment. By ‘environment’, we do not mean a monolithic, integrated enterprise with its own editors and debuggers, but rather a philosophy of software construction similar to that of Unix: Programmers should 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.

Accordingly, the first attempt at the new Grail was based on several shell-executable
filters which could be pipelined to perform operations on an input automaton. The minimization of an automaton, for example, could be achieved by executing the following:

\[
\text{reverse } < \text{input-fa } | \text{ subset } | \text{ reverse } > \text{output-fa}
\]

This pipeline uses two invocations of \textit{reverse} and one of \textit{subset} to implement minimization. There are several points to note about this approach:

- **Modularity by multiple processes**
  Each of the components of the pipeline is a separate process that communicates by means of standard input and output. Encapsulation is automatically enforced by the operating system, which does not allow processes to access each others’ memory or other resources. Thus, the use of a multiple-process design encourages programmers to solve simple, general problems. Another advantage of this approach is that it is easy to distribute; by using the capabilities of \texttt{rsh} to set up internet pipes, we can run processes on different machines.

- **Text-based interchange language**
  A multiple-process design requires some form of interprocess communication, since processes cannot access each others’ routines. We chose to use a text-based description of automata and regular expressions as an intermediary; each process reads a text-based description of the input automaton, converts it to an internal form, processes it, and writes a text-based description of an output automaton. One advantage of this approach is that the input and output can be read, edited, and manipulated by standard Unix utilities. One disadvantage is the extra cost of encoding and decoding between the language and internal forms.

- **Ready-made programming language**
  While the multiple-process approach requires an interchange language, it saves us the effort of building a language for interacting with the software: we rely on the same language that the user employs in day-to-day activities with the operating system, namely the shell. This approach has the advantage that the user need not learn a new language, and can use whatever shell is most comfortable (e.g., \texttt{sh}, \texttt{csh}, \texttt{ksh}, \texttt{bash}). It also facilitates the insertion of other operating system utilities in the stream (e.g., \texttt{sort}, \texttt{wc}).
In our first attempt at the construction of Grail, we developed the following filters:

- **cross**: compute the cross product of two automata
- **lreverse**: reverse the input automaton using lambda transitions
- **min**: minimize the input automaton by Hopcroft’s partition algorithm
- **min1**: minimize the input automaton by reversal and subset construction
- **percent**: compute the alternation (i.e. $(ab)^*$) of two automata
- **plus**: compute $star-\varepsilon$ of the automaton
- **quest**: compute the automaton+$\varepsilon$
- **reverse**: reverse the input automaton
- **star**: compute the Kleene star of the input automaton
- **subset**: subset construction of the input automaton
- **union**: compute the union of two automata

These filters were written in C and linked with a library of functions that did most of the work. The library contained procedures for handling input/output, and for processing automata. The idea behind this decomposition was that the filters should be efficient enough for most problems involving automata; for the largest problems, a competent C programmer could access the library directly and thereby avoid any inefficiencies due to process communication.

While the filters were reasonably successful, the library was not. The first problem was that we experienced some difficulty with software reliability. C (at least as we wrote it) does not eliminate the temptation to access data structures directly, with the result that encapsulation or abstraction is violated. As an example, we had tried to plan for future changes to states in automata by using a type definition:

```c
typedef STATE int
```

This tactic proved inadequate. When we decided to represent states as `longs`, we could change the `typedef`, but we still had to change all code where we had used variables of type `int` to hold states. We needed a stronger form of information hiding.

The second problem, lack of reusability, was irritating both as an aesthetic and as an engineering problem. Operations on automata and regular expressions involve frequent manipulation of container classes such as sets and relations; it would be both elegant and efficient to use a single implementation of these classes for many different contents. C, however, does not help with this problem. If one wants type
checking, then one must provide each combination of container class and contained element, which is tedious; alternatively, one can forego type checking, use void pointers, and trust to luck. We were not willing to settle for either choice.

The final and probably most important problem was lack of clarity; despite our efforts to write software that expressed the algorithms in a fluent and elegant form, the result was unacceptable. Too much low-level detail kept cropping up in the algorithms, because C did not help us to deal with automata and regular expressions as first-class objects. In spite of these problems, the library was developed to the point that it supported a significant research project on subset construction[13].

The remainder of this report describes the current version of Grail, which is written in C++. As with the first version, written in C, we provide both a library of routines (in C++, this time) and a set of shell-executable filters, so that Grail can be used with relative ease at the shell level (with a slight decrease in efficiency), or at the C++ programming level (with a slight increase in programmer effort). The current version of Grail is much better in terms of reliability, clarity, and resusability.

3 Design

We shall present the design of Grail in a top-down fashion, first discussing the abstract objects, then the user level utilities, and finally the programming utilities.

3.1 Abstract Objects: Automata and regular expressions

Currently, Grail handles finite automata (deterministic and nondeterministic) and regular expressions. It cannot handle context-free grammars, pushdown automata, or other more powerful objects.

3.1.1 Finite Automata

In Grail, finite automata depart from the classical model in two ways: they may have multiple start states, and transitions between states may depend on regular expressions rather than simply on letters. We call these extended finite automata. The other properties of extended finite automata are the same as classical FAs (they have multiple final states, they can be deterministic or nondeterministic, and they
have a finite number of states, transitions, and labels). Currently Grail does not support \( \varepsilon \)-transitions.

For simplicity, finite automata are specified in Grail as a set of transitions. Start and final states are denoted by special pseudo-transitions. For example, a simple automaton that accepts the language \( ab \) can be specified as follows:

\[
\begin{align*}
(\text{START}) & \rightarrow 0 \\
0 & \text{a} 1 \\
1 & \text{b} 2 \\
2 & \rightarrow (\text{FINAL})
\end{align*}
\]

An automaton specification is a set of transitions, usually one per line. Each transition consists of a source state, a transition label, and a target state. States are given as integers and labels are given as regular expressions (though typically they are single letters).

The start and final states of the example automaton are specified by the two pseudo-transitions that have a fixed state and transition label. Note that we use the reserved labels \( \rightarrow \) and \( \rightarrow \) for the pseudo-transitions; they are the only labels permitted for these transitions. Conceptually, these special labels are ‘end-markers’ that bound the input string. Also note that the states (START) and (FINAL) are not ‘real’ states, they simply indicate that the other state in the transition is either a start or final state.

Although Grail permits transition labels to be regular expressions, not all operations are applicable to such automata. Subset construction, for example, is defined only for automata with single-letter transitions. Grail supports both automata with general regular expressions as transition labels, and the more common form that have only a single letter for each transition.

### 3.1.2 Regular expressions

Grail supports the classical form of regular expressions over the letters of the (ASCII) alphabet, using the operators Kleene star (\( \ast \)), catenation (\( \wedge \)), and union (\( + \)). The precedence of the operators follows this order, with Kleene star having the highest precedence. As usual, we can use parentheses to override the default precedence.
Grail stores its regular expressions in reverse-polish or postfix form; that is, without parentheses. Thus, if one inputs an expression with redundant parentheses and then outputs it, only the minimal parentheses are provided:

```
echo "(((ab)))" | remin
ab
```

The function `remin` computes the minimal parentheses for a regular expression (merely converting it to postfix form and then calling the standard output function).

Grail provides functions to convert regular expressions to finite automata and vice versa.

### 3.2 User level

The outer layer of Grail is a set of Unix processes, each of which performs some useful transformation on its input and passes the result to its output. The input is an automaton or regular expression which is filtered to produce the output automaton or regular expression. We follow the convention that filters that are to be applied to automata are prefixed with `fa`, while filters that are to be applied to regular expressions are prefixed with `re`. The filters that are currently implemented include:

- `fareverse` reverse the input automaton
- `faquest` compute `?` of the input automaton
- `fastar` compute `*` of the input automaton
- `fatore` convert the input automaton into a regular expression
- `remin` generate the minimal parentheses for the input regular expression
- `restar` compute `*` of the input regular expression
- `retofa` convert the input regular expression into an automaton

All process communication is solely by means of the automata or regular expressions that are generated as output; no state or other intermediate information is transmitted. This choice ensures that the input and output of any filter can be examined or further processed with text editors or other text processing tools. For example, the number of transitions in an output automaton can be counted easily with `wc`:

```
cat really-big-regexp | retofa | wc
```
3.3 Programmer interface

Each of the user level filters is built using the Grail library libgrail.a; in most cases the filter is simply an I/O program that makes one or more calls to libgrail.a. Thus, competent programmers have easy access to Grail at the function call level. Function call access is useful when the needed functionality cannot be provided by combining existing filters (in this case, we encourage programmers to develop filters based on libgrail.a and make them available to other users of Grail), or where the size of an automaton or the complexity of its processing makes the use of multiple processes infeasible.

The internal construction of Grail reuses as much of the existing code as possible. We chose C++ as the implementation language to encourage reuse and modularity, and to discourage hidden dependencies and connections.

Grail uses a combination of inheritance, containment, and multiple representations with transformations.

3.3.1 Inheritance

The inheritance hierarchy employed by Grail is flat. The major example of inheritance is the template-based use of sets. set is the generic template class that maintains a collection of elements, each unique in the collection. Grail uses sets of regular expressions, states, and transitions. Sets of regular expressions and states are straightforward instantiations of the set template for those types; sets of transitions have been captured in a special class definition (tset) because of the many functions particular to sets of transitions.

There is currently one instance of inheritance from xfa:

fa: xfa

xfa is the class for extended finite automata (whose labels can be general regular expressions). fa is a subclass of xfa that restricts the labels to single letters. It inherits all the functions of xfa, and (is planned to include) others, such as subset construction and minimization, which are defined for automata with only single-letter labels.
### 3.3.2 Containment

Containment is a relationship in which a class includes an instance of another class, rather than inheriting from it. Containment supports information hiding, but does not require the container class to be a superclass of the contained class. In Grail we make heavy use of containment to hide the details of the data structures used to implement certain classes, while still permitting other objects to make use of those classes. Containment is sometimes called composition[16] layering[15], or delegation[4]. Grail uses containment for the following relationships:

- \( \text{fa} \to \text{tset} \) An \( \text{fa} \) is represented as a \( \text{tset} \), a set of transitions.
- \( \text{tset} \to \text{trans} \) A \( \text{tset} \) is a set of \( \text{trans} \).
- \( \text{trans} \to \text{regexp}, \text{state} \) A \( \text{trans} \) is represented as a pair of \( \text{states} \) and a \( \text{regexp} \).
- \( \text{regexp} \to \text{string} \) A \( \text{regexp} \) is represented as a string, in reverse polish notation.

Containment provides the advantages of encapsulation and code reuse without the knotty ontological problems posed by inheritance. By using containment, we avoid the question of whether a \( \text{regexp} \) ‘is-a’ type of \( \text{string} \)–we merely decide that a string is a reasonable representation for a \( \text{regexp} \). By using containment, we ensure not only that improvements in the efficiency of \( \text{string} \) will be immediately passed on to \( \text{regexp} \) (and indirectly, to any object that delegates to \( \text{regexp} \)), but also that \( \text{regexp} \) does not make any unwarranted assumptions about \( \text{string} \)’s implementation. \( \text{regexp} \), like any other class, has no access to \( \text{string} \) beyond its public members and methods. If \( \text{regexp} \) inherited from \( \text{string} \), however, we would have no such guarantee of encapsulation.

### 3.3.3 Multiple representations and transformation

The third type of relationship in Grail is that among what we might call peer classes. The best current example is the relationship between \( \text{fa} \) and \( \text{regexp} \).

---

1 It might seem reasonable to implement \( \text{regexp} \) as a subclass of \( \text{string} \), since then we can reuse the representation and operations of \( \text{string} \). But formally speaking, any fixed string is trivially a regular expression, which suggests that \( \text{regexp} \) should be a superclass, not a subclass, of \( \text{string} \)!

The conflict between the implementation and specification aspects of a class hierarchy is a common problem in object-oriented design[17], one we try to avoid with containment.
these are peer classes because it is possible to convert from one to the other. Instead of trying to arrange such classes in an inheritance hierarchy, we treat them as roots of their own hierarchies, and use transformation routines to convert regular expressions into finite automata and vice versa. Finite automata and regular expressions are two different representations for the same class of languages, with well known transformation algorithms.

In the future, we plan to implement classes such as digraph, relation, and monoid. Transformation routines will be written to convert from xfa to these classes, so that we can exploit algorithms from these new domains to address problems in finite automata. For example, automata can be converted to graphs and then tested for connectedness using graph-based algorithms, and automata can be converted to relations and processed with relational algebra.

With some peer classes, the existence of an inverse transformation is not guaranteed; we cannot map arbitrary graphs or relations to automata, as we can with regular expressions. This observation might suggest that graph and relation are more abstract objects and, perhaps, superclasses of fa. But which is the more appropriate superclass? Should relation be a superclass of graph or vice versa? The problem with a static inheritance hierarchy is that it sometimes forces us to answer in advance problems that seem to be open research questions. We see little advantage in trying to enforce one view, and so we treat the objects as ‘incomparable’ peers.

4 Software organization

There are many ways to organize C++ applications; we have chosen one that seems workable.

4.1 System directories

Grail is organized as four directories:

    bin  classes  grail  tests

bin contains the Grail executables and the library, libgrail.a.

classes contains the definitions of each of the Grail classes. Each individual class is kept in a separate directory whose name is the class name. Thus,
classes/regexp and classes/tset are subdirectories of classes that contain the regular-expression class and the set-of-transitions class, respectively.

grail contains the definitions of the individual Grail filters. These filters are programs that use the Grail classes to perform some useful activity. For example, retofa is a program that reads a regular expression on its standard input and generates a corresponding finite automaton on its standard output. Most of the filters in the Grail directory are relatively simple programs that merely input some structure, call a method defined by a Grail class, and output the result.

A very important directory is tests; it contains a set of automata and regular expressions that can be used for testing the correctness and efficiency of Grail and any modifications that you make to Grail. It also contains scripts for automatically running a set of tests against the Grail code.

### 4.2 Classes

Grail includes the following classes, each in its own directory:

- fa: finite automata (letters on transitions)
- xfa: finite automata (regular expressions on transitions)
- regexp: regular expressions
- set: sets (template definition)
- state: states
- string: strings of characters
- trans: transitions (of automata)
- tset: sets of transitions

### 4.3 A class directory

Grail uses a set of conventions for naming the files in a class directory. It is easier to locate definitions, declarations, and other components in classes if they obey the conventions.
Each class directory contains 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 methods defined for a class is contained in a separate `method.cc` file. When the method is a function call with an alphanumeric name, its filename is the same name (for compatibility with non-flexname file systems, long method names are shortened to fit an 8-character limit). Hence, the method `parse` in the class `regexp` is located in the file `parse.cc`. Some overloaded operators are defined in files with the name used by the C++ class. The definition of the `==` operator, for example, is found in the file `==.cc`. This approach does not work very well for some overloaded operators; in particular, files with names like `<.cc` and `>.cc` would be misinterpreted by many command shells. For these situations we have chosen to use the following standard alphabetic names:

- `operator<<` is defined in `ostream.cc`
- `operator>>` is defined in `istream.cc`
- `operator<` is defined in `lt.cc`
- `operator>` is defined in `gt.cc`
- `operator!=` is defined in `ne.cc`
- `operator+=` is defined in `pluseq.cc`
- `operator-=` is defined in `minuseq.cc`
- `operator^=` is defined in `concat.cc`
- `operator+` is defined in `plus.cc`
- `operator-` is defined in `minus.cc`
- `operator[]` is defined in `index.cc`

We use `classname.cc` for constructors and `~classname.cc` for destructors. Constants, macros, and types that are specific to a class are kept in a file called `defs.h`.

The set of system and local files that are necessary for compilation of methods are specified in the file `include.h`. 
Finally, there is a Makefile for compiling the code. The Makefile does not compile each method separately, but first constructs a single file to represent the whole class. This technique is used for two reasons. First, it requires much less time to compile the whole class than to compile each of its methods separately (though it may lead to larger executables). Second, some C++ compilers use the name of the file to construct an external entry point for the destructor function, which could lead to linking problems if the same filename is used for some future class. We avoid this problem by catenating all the method files into a single classname.c file and then compiling classname.c[6] The disadvantage of this approach is that compilation errors are given relative to classname.c instead of relative to the original component source files, which makes correcting the code slightly more complicated.

4.4 Making the code

The whole system can be compiled by executing

```
make clean
make
```

at the root directory. These commands first create libgrail.a by compiling each class. Then, each filter is created and placed in bin. Finally, the tests are run.

4.5 Test directory

The directory tests contains facilities for testing Grail. By executing

```
make checkout
```

in this directory, all the filters in bin are checked against the existing test automata and regular expressions. make checkout runs the appropriate testing script (fatest for automata, retest for regular expressions). The script executes each filter with each test object as input, and compares the result with a previously obtained result stored in a subdirectory named for the filter. Thus, fatore is run against dfal and the result compared with tests/fatore/dfal. 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. Erroneous tests
are named as `filter.object`; for example, an error when running `fatore` on `dfa1` would result in a file in `errors` named `fatore.dfa1`.

5 Miscellaneous information

5.1 How do I obtain Grail?

Grail is available without charge to researchers and teachers, or anyone who wishes to use the software for their own private purposes. Grail is not in the public domain; you can’t sell it or make it part of a commercial product without our permission. For more information on how to obtain Grail, send e-mail to drraymond@daisy.uwaterloo.ca.

5.2 Related software systems

Several systems for computing with automata have appeared in the literature.

The AUTOMATE system, written in C, supports finite automata and finite semigroups[3]. It can compute deterministic minimal automata, syntactic monoids, and transition monoids of regular languages.

The AMORE system, written in C, supports finite automata, regular expressions, and syntactic monoids[7]. It can produce minimal DFAs, handle ε-NFAs, and perform various tests on syntactic monoids (for example, star-freeness, finiteness, cofiniteness, etc). AMORE can also graphically display its automata.

Both AMORE and AUTOMATE have goals similar to those of Grail—to serve as a research environment, to facilitate the study of automata implementations, and to provide a package for executing automata for other purposes (such as validating concurrent programs). Where Grail differs is in its emphasis on several levels of interface (process, function library, object); AMORE and AUTOMATE appear to be monolithic programs that attempt to provide a single interface to the user.

One interesting experience is the development of automata tools in Nuprl, a proof language based on lambda calculus[11]. Definitions were constructed in Nuprl for finite sets, strings, tuples, and deterministic automata. 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 automata, but an illustration of the utility of the Nuprl proof development system.
We know of two other systems whose motivation was primarily pedagogical. An early effort was GRAMPA, which was only partially implemented[1]. More recently, Hannay has built a Hypercard-based systems for simulating automata[5]. This program appears to be useful for introductory teaching purposes, and for simulating small automata.

In addition to these systems, there is a vast amount of work on using grammars and automata in applications. Many operating system utilities understand a limited form of regular expression, for example, and there is hardly a text editor that does not perform general purpose search-and-replace. It seems that the automata used in such tools are generally custom-built, or perhaps adapted from custom code; operating systems have yet to offer a standard automata package in the same way that they offer a standard sorting routine.

5.3 Future work

There is much work we would like to do on Grail.

In the near term, we intend to expand our collection of processes for manipulating automata and regular expressions, until we attain a reasonably complete set. We are also interested in considering some special cases of the objects that we have already implemented; for example, Glushkov automata[2], and restricted classes of regular expressions (that might be susceptible to automatic minimization).

In the long term we have several goals. The first goal is to increase the number of “peer” classes to include objects such as monoids, digraphs, and relations (n-tuples). Some types of automata computation can be reduced to well-known algorithms on these objects. The second goal is to add more powerful objects such as finite transducers, pushdown automata, Turing machines, and multitape automata. The third goal is to develop simulators for these automata, so that they can be used in text transformation, parsing, and other applications. Finally, we want to improve the usability of the software by providing a variety of graphical ways to view automata and to observe their execution.

5.4 C++ references

The basic C++ reference book is the Annotated C++ Reference Manual, by the language’s author, Bjarne Stroustrup. More expository references are Stroustrup’s introductory book[16] or Lippman’s excellent primer[14]. For a detailed discussion
of the design of a basic C++ class library, see Data Abstraction and Object-Oriented Programming in C++ by Gorlen et al. Handy tips and hints on effective C++ programming are discussed by Meyers[15], and also by Henricson and Nyquist[6].

5.5 Organizational quirks

- The software requires a version of C++ that supports templates. We have successfully compiled it with AT&T cfront 3.0.

- The current approach to Makefile does not work with standard SunOS make, which complains about the macros on the dependency line. The solution: use a real version of make, like dmake.
6 References


11. Christoph Kreitz, “Constructive Automata Theory Implemented with the Nuprl
Proof Development System,” Technical report TR-86-779, Department of
Computer Science, Cornell University, Ithaca, New York (September 1986).

12. E. Leiss, “REGPACK: An Interactive Package for Regular Languages and Fi-
ite Automata,” Research report CS-77-32, Department of Computer Sci-
ence, University of Waterloo, Waterloo, Canada (October 1977).

CS-92-29, Department of Computer Science, University of Waterloo, Water-
loo, Canada (April 1992).

14. Stanley B. Lippman, C++ Primer, 2nd ed., Addison-Wesley, Reading, Mas-

15. Scott Meyers, Effective C++: 50 Specific Ways to Improve Your Programs and


17. David A. Thomas, Wilf R. LaLonde, and John R. Pugh, “Why Exemplars are
Better than Classes,” Technical report SCS-TR-93, School of Computer Sci-
ence, Carleton University, Ottawa, Canada (May 1986).
A  Finite automata: fa

A.1  Definition

fa are standard finite automata that have a single start state, multiple final states, and single-letter transition label.

fa is a specialization of xfa.

A.2  Public functions

fa()
   Constructor. Initializes status.

fa(const fa& a)
   Copy constructor. Copies transitions and status.

~fa()
   Destructor. A no-op.

A.3  Friend functions

istream& operator>>(istream& os, fa& s)
   Inputs s from stream is. Ensures that all transitions are single-lettered.
B  Regular expressions: \texttt{regexp}

B.1 Definition

\texttt{regexp} maintains a regular expression. It uses an instance of \texttt{string} to store the expression:

\begin{verbatim}
string p;
\end{verbatim}

The regular expression is maintained in reverse polish notation; that is, \texttt{p} contains a recursive expression of the following form:

\texttt{X Y size operator}

The regular expression is obtained by recursively doing \texttt{X operator Y}, where the value of \texttt{size} gives the size of \texttt{Y}, and the size of \texttt{X} is the remainder of the string.

B.2 Public functions

\begin{verbatim}
void clear()
    Clears the expression by truncating the string to 0.

void concat(char ch)
    Concatenates \texttt{ch} to the invoking \texttt{regexp}. Checks for non-alphabetic characters.

void concat(char* str)
    Concatenates \texttt{str} (a null-terminated character string) to the invoking \texttt{regexp}, using iostream capabilities (i.e. \texttt{istrstream}). Checks for null strings.

void concat(const \texttt{regexp}& r)
    Concatenates \texttt{r} to the invoking \texttt{regexp} using append.

void epsilon()
    Sets the invoking \texttt{regexp} to an \(\epsilon\)-expression.

void final()
    Sets the invoking \texttt{regexp} to a final expression.
\end{verbatim}
int isfinal()
Returns 1 if the invoking regexp is a final expression; returns 0 otherwise.

int isstart()
Returns 1 if the invoking regexp is a start expression; returns 0 otherwise.

void minus(const regexp& r)
Subtracts r from the invoking regexp. If invoking regexp is empty, it simply appends the argument regexp.

int operator!=(const regexp& r)
Returns 1 if the invoking regexp and r are not identical; returns 0 otherwise. Does not compute language equivalence.

void operator=(const regexp& r)
Assignment operator. Checks for self-assignment and then copies r to the invoking regexp.

void operator=(char c)
Assignment operator. Assigns the invoking regexp to be the single letter c.

int operator==(const regexp& r)
Returns 1 if the invoking regexp and r are identical; returns 0 otherwise. Does not compute language equivalence.

void operator+(const regexp& r)
Disjunction of the argument regexp with the invoking regexp. If invoking regexp is empty, simply appends r.

regexp()
Constructor. A no-op.

regexp(const regexp& r)
Copy constructor.

int size()
Returns the current size of the underlying string.
void star()

Kleene start operator. A no-op if current expression is empty; otherwise merely appends a star. This may result in superfluous stars.

void start()

Sets the invoking regexp to a start expression.

void wterm(char* str, int j, int upperop, ostream& os)

Recursively output terms of the invoking regexp. upperop gives the parent operator of the parse tree, and is used to determine operator precedence (and hence bracketing). j is the length of the right term (and hence the length of the string that is skipped).

~regexp()

Destructor. A no-op.

B.3 Friend functions

xfa& xterm(char* s, int x, xfa& a)

Used in the conversion of regexp to xfa.

void retofa(regexp& r, xfa& a)

Converts a regexp to an xfa. This function is defined in the class xfa.

void wterm(char* s, int x, int y, ostream& os)

Outputs a regexp by outputting terms recursively.

ostream& operator<<(ostream& os, const regexp& r)

Outputs r on stream os. Epsilon, start, and final regexps are treated as special cases; for all other regexps, wterm is invoked.

istream& operator>>(istream& os, regexp& r)

Inputs r from stream is. Calls shift-reduce parser parse repeatedly for each concatenated segment of the regexp.
B.4 Private functions

```c
int term(char* str, int* input)
    Parses individual terms of the input string.
```

```c
int token(char* str, int* i)
    Returns the next non-whitespace, non-operator, non-bracket token from str.
    Returns 0 if at end of the string or if an unacceptable character is found (e.g.,
    non-alphabetic, non-whitespace, non-bracket, and non-operator).
```

```c
int parse(char* str, int* input)
    Parses the string str to generate the corresponding regular expression.
```
C  Sets: set

C.1 Definition

set is a template for a set of objects of class Item.

set maintains the following variables:

Item* p; // array of Items
int max; // maximum size of array
int sz; // number of elements currently in array

Note that operator>> is not defined.

C.2 Public functions

void clear()
    Sets the size to zero. Does not free any space used by current members.

void intersect(const set<Item>& s1, const set<Item>& s2)
    Clears the invoking set, then adds any members belonging to the intersection
    of s1 and s2.

int member(const Item& s)
    Checks for membership of s in the set. Returns 1 if s is a member, and
    returns 0 otherwise.

void operator=(const set<Item>& s)
    Assignment operator. Checks for self-assignment; clears; adds s to the in-
    voking set.

void operator=(const Item& i)
    Assignment operator. Checks for self-assignment; clears; adds i to the in-
    voking set.

void operator+=(const set<Item>& s)
    Addition operator. Checks for self assignment, and adds each member of s
    to the invoking set.
void operator+=(Item q)

Checks q for membership in the set, allocates additional space if necessary, then adds q to the invoking set. q is copied with the assignment operator of the class Item.

void operator==(const set<Item>& s)

Equivalence operator. Tests sets for equivalent sizes, then tests each member of the invoking set for membership in s.

void print(ostream& os)

Prints every element of the set. This function is called by operator<<.

rel operator[](int i) const

Selection operator. Returns the i th Item. Though currently implemented as array selection, it need not be, and programmers should not depend on this.

int size() const

Returns the current size of the set.

set()

Constructor. Allocates space and sets the size to zero.

set(const set<Item>& s)

Copy constructor. Allocates space and copies s to the invoking set.

~set()

Destructor. Deletes the array of Items.

C.3 Friend functions

ostream& operator<<(ostream& os, set<Item>& s)

Outputs s on stream os. This function apparently cannot be defined within the template, since it is a friend function; hence, it is defined outside the template, and it calls the private function print.
D  States: state

D.1 Definition

state is a source or sink for transitions in an automaton. At the moment it is a simple wrapper class for longs. It exists for two primary reasons: first, to ensure that no transition code is written that embeds knowledge of the storage type of state (so that states may become arbitrarily complex in the future); and second, to handle the exceptional states (START) and (FINAL).

state maintains the following variable:

long number; // state number

D.2 Public functions

int operator> (const state& s)
    Returns 1 if the invoking state is strictly larger than s; returns 0 otherwise.

int operator> (int& i)
    Returns 1 if the invoking state is strictly larger than i; returns 0 otherwise.

int isnull ()
    Returns 1 if the invoking state is null; returns 0 otherwise.

int operator< (const state& s)
    Returns 1 if the invoking state is strictly smaller than s; returns 0 otherwise.

int operator< (int& i)
    Returns 1 if the invoking state is strictly smaller than i; returns 0 otherwise.

int operator!= (const state& s)
    Returns 1 if the invoking state is not equal to s; returns 0 otherwise.

int operator!= (int& i)
    Returns 1 if the invoking state is not equal to i; returns 0 otherwise.
void null()
    Sets the invoking state to null.
void operator+=(const state& s)
    Adds value of s to the invoking state.
void operator+=(int& i)
    Adds value of i to the invoking state.
void operator-=(const state& s)
    Checks that the invoking state is greater than s, then subtracts the value of s from the invoking state.
void operator-=(int& i)
    Checks that the invoking state is greater than i, then subtracts the value of s from the invoking state.
void operator=(const state& s)
    Assignment operator. Checks for self-assignment; copies s to the invoking state.
void operator=(int& i)
    Assignment operator. Copies the value of i to the invoking state. Does not check that i is non-negative (otherwise we cannot set state to null).
void operator=(long& i)
    Assignment operator. Copies the value of i to the invoking state. Does not check that i is non-negative (otherwise we cannot set state to null).
int operator==(state& s)
    Equivalence operator. Returns 1 if the invoking state has same value as s; returns 0 otherwise.
state()
    Constructor. Sets value to 0.
state(const state& s)
    Copy constructor. Copies value of s to the invoking state.
long value() const
    Returns value of the invoking state.

~state()
    Destructor. A no-op.

D.3 Friend functions

ostream& operator<<(ostream& os, const state& s)
    Outputs s on stream os.

istream& operator>>(istream& is, state& s)
    Inputs s from stream is.
E Strings: string

E.1 Definition

string maintains a dynamic array of characters. It uses the following variables:

```
char* c; // pointer to characters
int sz; // current length of string
int max; // length of allocated space
```

The character string c is null-terminated for consistency with the standard string package.

E.2 Public functions

```
char* chars() const
    Returns a pointer to the characters.

void operator=(const string& s)
    Assignment operator. Checks for self-assignment, clears, and then appends s to the invoking string.

void operator=(char* s)
    Assignment operator. Clears, then appends null-terminated character sequence s to the invoking string.

void operator=(char ch)
    Assignment operator. Clears, then appends character ch to the invoking string.

int operator==(const string& s)
    Equivalence operator. Returns 1 if s is identical to the invoking string; otherwise returns 0.

int operator!=(const string& s)
    Difference operator. Returns 1 if s is different from the invoking string; otherwise returns 0.
```
```cpp
int operator+=(const char& ch)
    Append the character ch to the invoking string.

int operator+=(const char* str)
    Append the character string str to the invoking string.

int operator+=(const string&* str)
    Append str to the invoking string.

int operator+=(int value)
    Convert integer value to a character string, then append it to the invoking string.

char operator[](int& i) const
    Selection operator. Returns the ith character in the string. If string is shorter than i, return EOF.

int size() const
    Returns the current size of the string.

string()
    Constructor. Allocates space and sets size to zero.

string(const string& s)
    Copy constructor. Allocates space and copies s to the invoking string.

int truncate(int x)
    String truncation. Sets size to x (x may be zero).

~string()
    Destructor. Deletes space occupied by c.
```

### E.3 Friend functions

```cpp
ostream& operator<<(ostream& os, const string& s)
    Outputs s on stream os.

istream& operator>>(istream& os, string& s)
    Inputs s from stream is. Treats either pairs of whitespace or pairs of " as string delimiters.
```
**F Transitions: trans**

**F.1 Definition**

trans is a relation describing a single directed transition. It consists of:

```plaintext
state source; // source state
regexp label; // transition label
state sink; // sink state
```

**F.2 Public functions**

```plaintext
void assign(const state& s1, const regexp& r, const state& s2)
     Assigns the argument values to the invoking trans.

regexp get_regexp()
     Returns the label.

state issfinal()
     If the invoking trans is a final (pseudo)transition, returns the source state;
     otherwise returns null.

state isstart()
     If the invoking trans is a start (pseudo)transition, returns the sink state;
     otherwise returns null.

void make_epsilon(const state& s1, const state& s2)
     Makes the invoking trans an ε-transition, with the given source and sink states.

void make_final(const state& s1)
     Makes the invoking trans a final transition with the given source state.

void make_start(const state& s1)
     Makes the invoking trans a start transition with the given sink state.
```
int operator!=(const trans& t)
If the source, sink and label are equivalent, return 0; otherwise return 1. Tests for identity, not language equality.

int null()
Returns 1 if the invoking trans is null; otherwise, returns 0.

void operator=(const trans& t)
Assignment operator. Checks for self assignment; then assigns components of t to the invoking trans.

int operator==(const trans& t)
If the source, sink and label are equivalent, return 1; otherwise return 0. Tests for identity, not language equality.

void renumber(int bottom)
Renumber the states in the invoking trans by adding bottom to their value.

void reverse()
Swap start and final states of the invoking trans. Handles the special cases of start and final pseudo-transitions.

void setnull()
Sets the invoking trans to null.

int sinkis(const state& s)
Returns 1 if the sink state of the invoking trans is equivalent to s.

int sourceis(const state& s)
Returns 1 if the source state of the invoking trans is equivalent to s.

int labelis(const regexp& r)
Returns 1 if the label of the invoking trans is equivalent to r. Tests for identical regexps, not language-equivalent ones.

trans()
Constructor. A no-op.
trans(const state& s1, const regexp& r, const state& s2)
    Constructor with initializers.

trans(const trans& t)
    Copy constructor.

~trans()
    Destructor. A no-op.

F.3  Friend functions

ostream& operator<< (ostream& os, const trans& t)
    Outputs t on stream os. Correctly handles start and final pseudo transitions.

istream& operator>> (istream& os, trans& t)
    Inputs t from stream is. Correctly handles start and final pseudo transitions.
G  Sets of transitions: tset

G.1  Definition

tset is a specialization of set<trans>. It defines no new variables, but does define several functions that are specific to sets of transitions (and thus is not merely a standard instantiation of the template).

G.2  Public functions

tset add(const trans& a)
    Adds a to the invoking tset.

void cartesian(const set<state>&a, const set<regexp>&b,
               const set<stae>& c)
    Computes the cartesian product a x b x c. The resulting set of trans is added to the invoking tset.

void finals(set<state>& s)
    Returns the set of states that are finals.

void renumber(int b)
    Renumber all trans in the invoking tset by adding b to all states.

void reverse()
    Reverse every trans in the invoking tset.

void rmtrans(const state& s)
    Removes every trans in the invoking tset that has s as a source or sink state.

void rmtrans(int i)
    Remove the single trans denoted by i (returned by a previous call to member).

tset& select(const state& r, int which
    Returns a tset containing all trans of the invoking tset that contain r. The argument which selects whether r should be a source, sink, or a non-pseudo state (that is, one that does not participate in a pseudo-transition).
tset& select(const regexp& r)

Returns a tset containing all trans of the invoking tset whose label is a regexp identical to r.

set<state> sinks()

Returns the set of sink states (all that are the sink of some transition). Creates a new set<state>.

set<state> sources()

Returns the set of source states (all that are the source of some transition). Creates a new set<state>.

void starts(set<state>& s)

Returns the set of states that are start states.

void labels(set<regexp>& r)

Returns the set of regexps that are labels.

tset()

Constructor. Clears the set.

tset(const tset& s)

Copy constructor. Allocates space and copies s to the invoking tset.

~tset()

Destructor. Deletes the set.

void operator=(const tset& s)

Assign s to the invoking tset. Checks for self-assignment; ensure sufficient storage; copy all trans.

void operator+=(const tset& s)

Adds s’s transitions to those of the invoking tset. Assumes a consistent numbering scheme between the two tsets.

void operator-=(const tset& s)

Removes s’s transitions from those of the invoking tset (if any are held in common). Assumes a consistent numbering scheme between the two tsets.
G.3 Friend functions

`ostream& operator<<(ostream& os, const tset& s)`

Output `s` on ostream `os`.

`istream& operator>>(istream& is, tset& s)`

Input `s` from istream `is`. 
H Extended finite automata: xfa

H.1 Definition

xfa are finite automata that have regular expressions for labels, and that allow multiple start and final states.

xfa maintains the following variables:

\[
tset \text{ transitions; // set of transitions}
\]
\[
int \text{ status; // dfa, nfa, etc.}
\]

H.2 Public functions

void clear()

Clears all transitions and other interval variables.

state \text{ max\_state()}

Returns the maximum state value.

int \text{ no\_start()}

Returns the number of start states.

int \text{ no\_trans()}

Returns the number of transitions.

void operator\^=(const xfa& a)

Concatenates a with invoking xfa. Computes the cartesian product of final states of the invoking xfa with the start states of a. Does not introduce \(\varepsilon\)-transitions.

void operator+=(const xfa& a)

Appends the transitions of a to the invoking xfa, renumbering them to avoid collision.

void operator=(const xfa& a)

Assignment operator. Checks for self-assignment; copies a to the invoking xfa.
void quest()
Computes ‘?’ of invoking xfa by copying it, computing the Kleene start of the copy, then concatenating the original xfa with the copy.

void reverse()
Reverses the invoking xfa by reversing its set of transitions. This is acceptable because multiple start states are permitted.

void single(const regexp& r)
Makes the invoking xfa a single-transition automaton (except for the necessary start and final pseudo-transitions), with r as the label for that transition.

void star()
Computes ‘*’ of invoking xfa.

regexp fatore()
Converts the invoking xfa to a regexp using state elimination.

xfa()
Constructor. Initializes status.

xfa(fa& a)
Copy constructor. Copies transitions and status.

~fa()
Destructor. A no-op.

H.3 Friend functions

ostream& operator<<(ostream& os, const xfa& s)
Outputs s on stream os.

istream& operator>>(istream& is, xfa& s)
Inputs s from stream is.

void retofa(regexp&r, xfa& a)
Convert regexp to xfa. Most of the work is done by xterm.
For each term in regexp, it computes the corresponding transitions. Operates recursively.