Release Notes for *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

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 `fl`.
2. There is improved memory usage (partly through the class `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 `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.[3]

Version 2.4 of *Grail* was effectively limited to machines of less than 10,000 states, or in other words, dictionaries of approximately

3 We would particularly like to thank Franz Guenthner and Boubaker Meddeb-Hamrouni for their interest in *Grail* for these purposes.

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 magnitude 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 instance 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 `pool`s. 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.[4] 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 `pool`s at different points of the execution of a program.

4 Unless we permitted rearranging blocks.

`fl`

The class `fl` describes a finite language: that is, a language composed of a finite number of finite-length words. The internal storage mechanism for `fl` is a `set<string<char> >` that contains the enumeration of the language. The input and output functions for `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 *Grail*. Another alternative is to permit user-defined delimiters, perhaps using the current *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 `fl`, and develop a more general technique for user-defined representations of all objects in future releases of *Grail*.

Several new filters accompany the introduction of `fl`.

fmtofl 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 `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.

fltofm 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.

fltore converts a finite language to a regular expression. This conversion is always possible. The expression is not 'minimal'. Given the following finite language:

```
adder
addend
```

```
sum
subtract
```

the resulting expression is `adder+addend+sum+subtract` and not `add(er+end)+su(m+btract)`.

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 `retofm`. For `retofl`, however, each subexpression is converted to a finite language instead of a submachine.

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

flfilter 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.

flprod 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.

flreverse 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.

flunion 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.

fllq 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 is `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.

flrq returns the right quotient of a finite language and a string. This is similar to the `fllq` 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 is `L`.

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.

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:

```
flexec
flfilter
flappend
flprepen
fllq
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 `mlychar`, `mlyint` or `re` languages, due to the problem with the stream operators. The functionality of `flexec` and `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 catenated.

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 `#ifdef`s 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:

```
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
```

xlC 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:

```
LDFLGS = -bnoso -bI:/lib/syscalls.exp -liconv -bnodelcsect
```

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 required 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:

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

3. MSVC does not equate `strstream.h` with the DOS filename `strstrea.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:

```
int function(size_t)
```

and not a:

```
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 `set_new_handler`, even if `new.h` is included. His solution is to do the following:

```
#ifndef MSVC
set_new_handler(&new_error); // error handler for new
#endif
```

Grail, as shipped, does not include changes or `#ifdefs` for Visual C++.

Symantec 7.0

Although it is possible to compile *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 `mealy` and `pair` classes. This is most easily accomplished by searching for `Label` and `Item`, and replacing with `S`. Also, `inst`
   `std.h` uses `T` as a parameter name.

2. An explicit declaration of

   ```
   template <class S>
   class inst<S>;

   template <class S>
   ostream&
   operator<<(ostream&, inst<S>&);
   ```

   is required in `inst`
   `inst.h` before the `inst` class definition. Also, the following must be added to `inst`
   `ostream.src`:

   ```
   #include "../re/re.h"
   ```

Although this is not required for the compilation process, lack of a definition for `re` thwarts the instantiation process.

3. Explicit manual instantiations of

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

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

```
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

```
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 `strstrea.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 `max`. During the initialization of array class (in the constructor), the `max` data member is initialized on an initialization list. This syntax is misunderstood as a call to the `max` macro. Moving the max initialization to an assignment in the constructor body.

4. An error was generated in `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, `_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 *Grail*. A compiler switch can be used to suppress 'informational' warnings.

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

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

Watcom

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 groused about using a `cast` in situations like this in `set.h` and `list.h`:

```
#ifndef WATCOM
      { (array<Item> &) *this = l; return *this; }
#endif
#ifdef WATCOM
      { array<Item>::operator=(l); return *this; }
```

```
#endif
```

Watcom also needed a special instantiation of `string::operator>>` in order to handle `istrstreams` (which should just be a derivation from the operator for `istreams`), 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.

19. Bitmap subset construction removed.

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 2 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.