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

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`

  This directory contains `.dvi` files for the *User's Guide*, the *Programmer's Guide*, and the *Release Notes*.

- `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
```

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/fmtore/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 fmcment on d1
Testing fmcment on d2
Testing fmcment on d3
Testing fmcment on d4
Testing fmcment on d5
Testing fmcment on d6
Testing fmcment on n1
Testing fmcment 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/fmtore.n2`. Comparing `errors/fmtore.n2` with `fmtore/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 `fmdeterm`, 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
        symbol_exp
```

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, `fm`s are stored as three sets: a set of start states, a set of final states, and a set of `inst`s.

`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 fm<S> and generates an fm<re<S> >; if it was made a member of **fm**, it would result in an infinite template instantiation (the generated fm<re<S> > would itself be a target of **fmtore**, generating an fm<re<re<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 |- and -| 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 **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 (**operator+=**) tests for membership before adding, while the disjoint union does not. It is the programmer's responsibility to check for disjointness.

string in *Grail* 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 trapdoor 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_xzy_exp`. This function determines how a given subexpression compares to an `xyz` expression. In effect, we are using two function calls (the operator and the `compare_xyz_exp`) to determine the actual types of both arguments to the comparison operation. This technique is called *double dispatching.*

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

`star_exp` overloads the `star` operator of `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`.[1] `squeeze.src` will make use of existing functions in `fm`, and it will probably also use other data structures in *Grail*, such as `set`s and `list`s. 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:

1 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:

```
static char re_star<int>;
static char re_plus<int>;
static char re_cat<int>;
static char re_lparen<int>;
static char re_rparen<int>;
static char* re_estring<int>;
static char* re_eset<int>;
static char re_left_delimiter<int>;
static char re_right_delimiter<int>;
static char re_left_symbol_delimiter<int>;
static char re_right_symbol_delimiter<int>;
```

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 catenation 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.

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`[2]

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 `!=`.

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 `fm`, for example, is implemented as two sets of `state`s and one set of `inst`s, 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 `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 `re`. Finite-state machine filters use the prefix `fm`. Finite language filters use the prefix `fl`. We also use these prefixes as suffixes for commands that convert from one type of object to another; for example, `retofm`. [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:

```
<<   ostream.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 into 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.

19. Fixed string/istream.src.

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

21. Ran Purify to test for memory access errors.

22. Ran Quantify to test for gratuitous inefficiences.

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 xfmenum, xfmexec, xfmcross, xfmmin, xfmminrev, xfmcment, 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 fmenum.

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).

6. Merged Makefile and Makefile.wat.

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 xlC 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.

6. End-of-function return values required for regexp/test*.cc.

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 xlC recompilation proceeds correctly.

10. set/include.h and list/include.h designed to handle the default requirements of xlC/Cfront template mechanisms (for xlC, 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 xlC 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.

24. Added functions fa::deterministic_density, xfa::number_of_transitions, xfa::number_of_labels, xfa::number_of_states.

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&).