Programmer's Guide to *Grail*

Version 2.0

Darrell Raymond *

January 1994

*Department of Computer Science, University of Waterloo, Waterloo, Canada

Contents

1 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*. The appendices to this document specify each of the classes in detail, with a brief description of all the functions and operators of each class.

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 grants from the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, and by an IBM Canada Research Fellowship. The author can be reached at `drraymon@daisy.uwaterloo.ca`.

2 Working with *Grail*.

This section is about compiling, testing, profiling, and using *Grail* as it is shipped.

2.1 Organization of the files.

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

- `bin`

  This directory contains symbolic links from each of the *Grail* filters to the main executables (which are found in directory `grail`).

- `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.

- `grail`

  This directory contains source code for the main executables, and the corresponding binary file. *Grail* filters are symbolically linked to these binaries.

- `lib`

  This directory contains `libgrail.a`, the *Grail* library.

- `profiles`

  This directory contains profiling scripts, profiling machines, and the results of previous profiling sessions.

- `tests`

  This directory contains test scripts, test machines, and the expected results for each test.

2.2 Compiling.

Before compiling *Grail*, you need to specify which C++ compiler you're using. In the root-level `Makefile` you will find the following parameters:

```
# set CCC to your compiler's path
CCC=CC

# set SYS to:
# XLC - if you're using IBM's xlC
# ATT - if you're using USL's Cfront
#SYS=XLC
SYS=ATT
```

You need to set the `CCC` variable to the executable of your compiler (use a full pathname if it isn't in your `$PATH`). You should also uncomment the appropriate `SYS` variable. *Grail* compiles cleanly under both IBM's xlC 1.00 and USL's cfront 3.0, but the two systems have slightly different requirements for inclusion of template headers and for file suffixes. The `SYS` variable is used by scripts during the making of *Grail*, to ensure that the right files are included and the right suffixes are generated.

After choosing a compiler and setting the appropriate variables, *Grail* is compiled by doing

```
make clean
make
```

from the root of the *Grail* filesystem. The `make` first compiles each of *Grail*'s classes and creates the library. Then it compiles the filter programs `fm.C`, `re.C`, and `fmre.C`, found in directory `grail`. This produces three executables. Finally, each of the filters is created by symbolically linking files in the `bin` directory to the appropriate executable in `grail`.

Each class is compiled separately. The `Makefile` for each class constructs a single file containing all the individual functions for a class. This technique is used for two reasons. First, it requires less time to compile one file than many (mostly because of preprocessing costs). Second, some C++ compilers use the source filename to construct an external entry point for the destructor function, which could lead to linking problems if the same filename is used for some other class. We avoid this problem by catenating all the function files into a single `classname.C` file and then compiling `classname.C`. The disadvantage of this approach is that compilation errors are relative to `classname.C` instead of to the original component source files, which makes fixing bugs slightly more complicated.

Most of *Grail*'s classes are template classes. These templates aren't compiled in the initial class compilation phase, but rather are instantiated as needed during the compilation of the filter programs `fm.C`, `re.C`, and `fmre.C`. Thus, the `Makefiles` for these classes don't call the C++ compiler, but simply create the class file.

Compiling *Grail* can take an hour or more, depending on system load, resources, and the quality of the template instantiation in your C++ compiler. Most of the compilation time is spent in template instantiation and linking.

2.3 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 check that modifications you make to *Grail* don't affect the correctness of its algorithms. *Grail* is tested by doing

```
make checkout
```

from the root of the *Grail* filesystem. The testing procedure checks all filters in `bin` 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 `dfm1` and the result compared with `tests/fmtore/dfm1`. 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 dfm1
Testing fmcment on dfm2
Testing fmcment on dfm3
Testing fmcment on dfm4
Testing fmcment on dfm5
Testing fmcment on dfm6
Testing fmcment on nfm1
Testing fmcment on nfm2
Testing fmcomp on dfm1
Testing fmcomp on dfm2
Testing fmcomp on dfm3
Testing fmcomp on dfm4
.
.
.
```

(No news is good news.) Some of the tests may put diagnostic messages on standard error (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 `nfm2` would result in the file `errors/fmtore.nfm2`. Comparing `errors/fmtore.nfm2` with `fmtore/nfm2` 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 doesn't test for language equivalence. 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*.

2.4 Profiling.

Grail has its own profiling system. This is useful for checking that 'improvements' to *Grail* actually do result in a performance benefit. *Grail* is profiled by doing

```
make profile
```

from the root of the *Grail* filesystem. The results of profiling are given as a table found in `profiles/profile.results`. The table looks like the following:

	total	dfm1	dfm2	dfm3	nfm1	nfm2	nfm3	nfm4
fmcment	1.02	0.99	1.00	1.00	1.00	1.00	1.03	1.01
fmcomp	1.03	0.99	1.01	1.00	1.01	0.99	1.04	1.01
fmcat	1.08	1.00	1.01	1.01	1.01	1.00	1.09	1.06
fmcross	1.20	1.01	1.02	1.03	1.03	1.01	1.25	1.13
fmdeterm	1.04	1.00	1.01	1.00	1.00	1.00	1.04	1.02
fmenum	1.08	1.05	1.01	1.01	0.95	1.52	1.18	1.08
fmmin	0.90	1.08	1.09	1.11	0.25	0.23	1.03	0.40
fmminrev	1.07	1.00	1.00	1.00	1.02	0.99	1.12	1.12
fmplus	1.04	0.99	1.01	1.00	1.01	0.99	1.04	1.05
fmreach	1.05	0.99	1.01	1.00	1.01	0.99	1.05	1.05
fmrenum	1.10	0.99	1.01	1.01	1.01	0.99	1.12	1.08
fmreverse	1.15	0.99	1.04	1.02	1.03	0.99	1.16	1.11
fmstar	1.04	0.99	1.01	1.00	1.00	0.99	1.04	1.05
fmtore	1.30	1.02	1.25	1.15	1.13	1.03	1.30	1.50
fmunion	1.14	1.01	1.03	1.02	1.02	1.01	1.18	1.09
iscomp	1.16	0.99	1.02	1.01	1.02	0.99	1.19	1.10
isdeterm	1.11	0.99	1.02	1.01	1.02	0.99	1.12	1.12
isomorph	1.09	1.00	1.01	1.02	1.01	1.00	1.13	1.08
isuniv	1.16	0.99	1.02	1.01	1.02	0.99	1.19	1.10
xfmcat	1.08	1.00	1.01	1.00	1.01	1.00	1.10	1.06
xfmreach	1.05	0.99	1.01	1.00	1.01	0.99	1.05	1.05
xfmplus	1.04	0.99	1.01	1.00	1.01	0.99	1.04	1.05
xfmstar	1.04	0.99	1.01	1.00	1.01	0.99	1.04	1.05
xfmtore	1.30	1.02	1.25	1.15	1.13	1.03	1.30	1.50
xfmunion	1.14	1.01	1.02	1.01	1.02	1.01	1.19	1.09

	total	reg1	reg2	reg3
recat	0.98	0.98	0.98	0.98
remin	1.02	0.98	1.04	1.02
restar	1.02	0.98	1.04	1.02
retofm	0.99	0.99	0.99	0.99
reunion	0.98	0.98	0.98	0.98
isempty	1.02	0.98	1.04	1.02
isnull	1.02	0.98	1.04	1.02

`profile.results` shows the cost of each filter for several sample inputs. The cost is shown as a ratio of the number of machine cycles used by the current implementation against a previously stored value. If the current version is significantly different from the previous one the ratio of cycles will be larger or smaller than 1.0—larger if current implementation is less efficient, and smaller if the current implementation is more efficient. The table shows the cycle ratio for each of a set of test cases, and also the total cycle ratio over all cases; this latter value appears in the leftmost column.

In the example table, we see that overall the current implementation is slightly less efficient than previous versions; it might suggest that the 'improvement' most recently added is actually making things worse. It's wise to use some care when interpreting the profile results, however, both because the results are dependent on the type of computer you use, and because the test machines are of different sizes. In particular, `nfm3` is ten times larger than the other test cases; thus, `nfm3` accounts for a disproportionate amount of the cycles in the overall total. Often, improvements will make some cases worse and some better; for example, if your improvement involves a substantial fixed overhead, you may notice the performance of the small test cases is worse, while that of the large test cases is better.

The `profiles` directory contains scripts that automatically instrument and execute each filter (`fmprofile`, `fm2profile`, `reprofile`, `re2profile`, and `xfm2profile`; the '2' scripts are used for filters that take two arguments) and scripts that compute the cycle ratio and produce a new `profile.results` file (`fmdiff` and `rediff`).

It's possible to generate profiles of current *Grail* with respect to older versions. The `profiles` directory contains a collection of previous profile results, named with a date and `.profile` suffix. Copying any of these files onto the file `current.profile` and then doing `make recompute` will generate a new results file that shows how *Grail* has improved since the date of the previous profile.

If you profile *Grail* over a long period of time, you may wish to retain a history of improvements. At each milestone, simply copy `current.profile` into a file named with the date or some other identifying label. Note that it isn't sufficient to save the file `profile.results`. This file is derived data and contains only the cycle ratios. The actual numbers of cycles are stored in files with a `.profile` suffix; they are the files that must be retained.

Grail's profiling mechanism is designed to work in environments that support the *pixie* profiler (provided on DEC MIPS systems). The profile harness should be easily extendible to other profilers. To make the profiling mechanism work with other profilers, write new scripts `fmprofile`, `fm2profile`, `reprofile`, `re2profile`, and `xfm2profile`. They must automatically generate instrumented versions of the code, extract the number of cycles after running a profile, and properly update the intermediate files.

2.5 Filters.

Grail provides 34 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 2.0, we've taken a different approach. All filters that deal with one type of object (that is, a machine or expression of a given type) are implemented by a single executable. This executable determines which function to apply by checking the name by which it was invoked. If the `fm` 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 is 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.

Each of the individual filters in Version 2.0 is actually a symbolic link from `bin` to the appropriate executable in `grail`. Using symbolic links eliminates the cost of storing multiple copies of the files.

2.6 Classes.

Grail employs 15 classes, organized in a relatively flat hierarchy:

```
fm
inst
list
re
set
state
string
subexp
        null_exp
        empty_set
        empty_string
        symbol_exp
        cat_exp
        plus_exp
        star_exp
```

The main classes are `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`, and `string`. The second type 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 null expressions (`null_exp`), empty set expressions (`empty_set`), empty string expressions (`empty_string`) single-symbol expressions (`symbol_exp`), catenation expressions (`cat_exp`), union expressions (`plus_exp`), and Kleene closure expressions (`star_exp`). Null expressions represent neither empty sets nor empty strings; they are an initialization expression that denote a regular expression with no content.

With the exception of `state`, all of *Grail*'s classes are templates that are instantiated with the input alphabet of the machine, language, or expression. *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 general comments about the functions or design of each class.

- fm

  Internally, fms are managed as sets of instructions including the pseudo-instructions. Thus, some routines are more complicated than they should be, because they must treat the pseudo-instructions as special cases. In a future version of *Grail*, pseudo-instructions will be used only for input and output, and not as an internal representation.

  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 simple 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

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

- `subexp`

  `subexp` is an abstract base class. Most of its functions are pure virtual functions.

- `string`

  A `string` in *Grail* is not a `char*`. Even a `string<char>` is not a `char*`, since it is not null-terminated. It is necessary to append a null character to a `string<char>` to handle it with functions such as `strcmp` or `printf`.

  `string` defines a function `ptr()` which returns the array 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 such 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.

- *container classes*

  `lists` and `sets` are both implemented as arrays of objects. Most of their functions are the same, though `lists` can be sorted and `sets` cannot. 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`.

  `list` defines a comparison function that is static; this is so that it can be passed to `qsort`.

  `set` is inefficient. It does a linear scan to determine membership, so updates are costly. This will be fixed in a future release.

  `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.

- *subexpressions*

  The subexpressions are `null_exp`, `empty_set`, `empty_string`, `symbol_exp`, `cat_exp`, `plus_exp`, and `star_exp`. These are all derived from `subexp` and override its virtual functions as appropriate.

  One complexity in the 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.

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 the example in Stroustrup's *The C++ Programming Language, 2nd Edition* on p. 217 for more information.

The `null_exp` class is the only subexpression that does not have a theoretical analogue. It is used as an initialization class and as a return value (it can be used when necessary to return something that has the type 'regular expression', but that actually indicates an error or other exceptional condition).

`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.

3 Changing and extending *Grail*.

This section is about modifying and improving *Grail*, and making it useful in your own applications.

3.1 Adding a new *Grail* filter.

Probably the easiest thing you can do is to create a new *Grail* filter. This filter may simply combine existing *Grail* functions, or it may rely on new functionality that you add to some of *Grail*'s classes. As an example, let us suppose you have discovered a new operation on machines that you call 'squeezing'. The way to add this functionality to *Grail* is to add the squeezing code to `grail/fm.C`. [1] `fm.C` is essentially a large case statement that selects which action is to be executed based on the value of its name; that is, based on the value of `argv[0]`. Simplified, `fm.C` looks like:

```
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 has chosen `fmsqueeze` as the name of the filter. If the executable is called with this name, then it will enter the body of the `if` statement.

[1] If your new filter was to be applied to regular expressions, you would add the code to `grail/re.C`.

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, [2] the squeezed machine is printed on standard output, and the filter returns.

In addition to modifying `grail/fm.C`, you also need to add a line to the `Makefile` to create a symbolic link from `bin/fmsqueeze` to the executable `fm.out`.

To fully integrate your filter with *Grail*, you should also add it to the profile and test directories. 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.) Thus, 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.

To add your filter to the profile directory, add a line to `profiles/Makefile` so that your filter will be profiled (use `fmprofile` if your filter uses only one argument, and `fm2profile` if your filter uses two arguments). The next time you run the profiler, the ratio shown for your filter for all test cases will be 0.0, because the profiler has no baseline. The *second* time you run the profiler, however, you will see some values (1.0 if you haven't improved your filter in the meantime, and some other non-zero value otherwise).

3.2 Parameterizing *Grail* with a new type.

One of the novel features that *Grail* provides is *parameterizable* machines and expressions: You can create new functionality simply by specifying a different type for the machines or expressions. As distributed, *Grail* implements `fm<char>`,

[2] Assuming that you have added this function to the definition of the class `fm`; see Section 3.3 for information on how to do this.

`fm<re<char> >`, and `re<char>`, but you can parameterize *Grail* with any base type, any *Grail* class, or any class that you create. All of the functionality of *Grail* is carried over to your parameterized class.

Parameterizing should be the easiest way to modify *Grail*, but it isn't. The reason it isn't is that template handling in most C++ compilers is still immature, and it is necessary to manually instantiate some of the templates. Presently, there are some files included by `classes/re/re.h` which define 'bogus' variables whose only purpose is to instantiate needed templates. If you parameterize with your own types, you will likely need to add some explicit instantiations to these files.

Parameterizing `Grail` is a new feature that hasn't been fully explored. We recommend that it be attempted only by experienced C++ programmers.

3.2.1 Parameterizing over a base type.

Suppose you want to create finite-state machines whose instruction labels are instances of `int`. You can do this with the following steps.

- Make a copy of `grail/fm.C`. Call it `grail/fmint.C`.
- Edit `fmint.C`. Change all variables of type `fm<char>` to `fm<int>`. Remove any of the filter definitions that you think aren't applicable to `int`s.
- Define `fm<int>::left_delimiter` and `fm<int>::right_delimiter`. These are two distinct characters which are used to delimit the printable representation of the type, and they will be used when outputting or inputting `fm<int>`s. If these are not specified, they default to the space character. One possible choice for delimiters is `‘` and `’`.
- Edit `grail/Makefile`. Add a target so that `fmint.C` will be compiled. Add a list of symbolic links from the `bin` directory to `fmint.out`. Be sure to use names that are distinct from the existing links.
- Compile *Grail* (which, if you've done the previous steps correctly, will compile your class and filters as well).
- If your compilation succeeds, and the filters operate properly, add test cases and profile information as discussed in Section 3.1. Write *man* pages for the `fmint` 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).

A similar process is used to parameterize `re`. The main difference is that you will need to define the following characters and strings:

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

These variables are used to specify the operators and other special symbols that are used in alphanumeric representations of `re`'s. The `chars` must be a single character; the `char*`'s must be exactly two characters. The defaults for these symbols are as given below:

re_star	star operator (default is '*')
re_plus	union operator (default is '+')
re_cat	catenation operator (default is '0')
re_lparen	left parenthesis (default is '(')
re_rparen	right parenthesis (default is ')')
re_estring	empty string (default is '""')
re_eset	empty set (default is '{}')
re_lambda	lambda symbol (default is '#'
re_left_delimiter	left delimiter of an `re` (default is ' ')
re_right_delimiter	right delimiter of an `re` (default is ' ')
re_left_symbol_delimiter	left delimiter of a symbol (default is '0')
re_right_symbol_delimiter	right delimiter of a symbol (default is '0')

The default symbols are found in `classes/re/std.h`. You may add your own definitions there, or put them in the header to your main routine.

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 delimiter are both `0`. If these values are specified (and only for the delimiters or catenation) then no output is generated for those symbols.

3.2.2 Parameterizing over your own types.

Parameterizing over your own types is much the same as parameterizing over base types or *Grail* types. However, there are two problems that are more likely to arise with parameterization of your own types.

The first issue 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. Consequently, we require as small a number of functions as possible (all of them are operators):

```
=
==
!=
<
>
<<
>>
```

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 us 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 only if they are identical. The second way, based on language equivalence, treats two re<char>s as equivalent only 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 affirmatively 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 `!=`.

3.3 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:

- Maintain the separation between a class's interface and its implementation. The class `fm`, for example, is implemented as a set of instructions, but this should not be visible outside the class. As much as possible, ensure that the interface is restricted to logical functionality.

- 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?

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

- If you create important new functionality, consider making it available through a separate filter. Follow the procedure that we described in Section 3.1 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- or three-letter prefixes for filters. Regular expression filters use the prefix `re`. Finite-state machine filters use the prefix `fm`. Filters that operate on finite-state machine with regular expression labels (so-called *extended* finite-state machines) use the prefix `xfm`. 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.

[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.

Each of the functions defined for a class is contained in a separate `function.cc` 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.cc`. Since operator functions don't have alphabetic names, we've chosen to use the following standard alphabetic names for operators:

<<	`ostream.cc`
>>	`istream.cc`
<	`lt.cc`
>	`gt.cc`
==	`eq.cc`
!=	`neq.cc`
+=	`pluseq.cc`
-=	`minuseq.cc`
^=	`concat.cc`
+	`plus.cc`
-	`minus.cc`
[]	`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 `defs.h`. The set of system and local files that are necessary for compilation of functions are specified in `include.h`.

3.4 Miscellaneous.

Some odds and ends:

- The template classes each contain a script `mksys`. This script merely converts the suffix of the `classname.C` file to the appropriate suffix for the compiler as determined by the `SYS` variable. This hack seems to be necessary because compilers have different ideas about what suffixes they support during template instantiation.

- The class headers include an `#ifdef` to ensure that a class is defined only once. This hack should be avoidable by proper use of the `#include` facility, but it doesn't seem possible (again, due to how template instantiation works).

- The classes derived from `subexp` (namely, `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.

- Why haven't we made *Grail* work with GNU C++? The main reason is that *Grail* depends heavily on templates, and GNU's support for templates is still incomplete.

3.5 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.

3.6 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&).

A Catenation expressions: cat_exp

A.1 Definition

cat_exps are parameterizable catenation subexpressions of regular expressions. cat_exp maintains two variables:

```
protected:

subexp<S>* left;
subexp<S>* right;
```

A.2 Public functions

```
cat_exp<S>& operator=(const cat_exp<S>& r) const
```

Assignment operator. Assigns the left and right of r to the left and right of the invoking cat_exp.

```
int operator==(const subexp<S>* r) const
```

Equivalence operator. Calls the argument's compare_cat_exp.

```
int operator!=(const subexp<S>* r) const
```

Inequivalence operator. Calls the argument's compare_cat_exp.

```
int operator<(const subexp<S>* r) const
```

Less-than operator. Calls the argument's compare_cat_exp.

```
int operator>(const subexp<S>* r) const
```

Greater-than operator. Calls the argument's compare_cat_exp.

```
int compare_cat_exp(const subexp<S>*,const subexp<S>*) const
```

Returns 1 if the invoking cat_exp is greater than the arguments, 0 if it is equal, and -1 if it is less than the arguments.

```
int compare_plus_exp(const subexp<S>*,const subexp<S>*) const
```

Returns -1 (cat_exp is always less than plus_exp.)

```
int compare_star_exp(const subexp<S>*) const
```

Returns -1 (cat_exp is always less than star_exp.)

```
subexp<S>* clone()
```

Clone operation. Simulates virtual copy constructor.

```
int contains_empty_string() const
```

Returns 1 if left or right contains the empty string, and returns 0 otherwise.

`int contains_cat_exp() const`

Returns 1 if `left` or `right` contains the empty set, and returns 0 otherwise.

`void convert_subexp(fm<S>& a) const`

Converts the invoking `cat_exp` into a `fm` and returns the result in `a`. Makes calls to other subexpression classes.

`void copy(const subexp<S>&)`

Copy operation. Used by `clone`.

`int is_empty_string() const`

Returns 1 if `left` and `right` are the empty string, and returns 0 otherwise.

`int is_cat_exp() const`

Returns 1 if `left` or `right` contains the empty set, and returns 0 otherwise.

`subexp<S>* minimize()`

Applies minimization heuristics.

`subexp<S>* new_subexp()`

Creation function. Simulates virtual constructor. Returns new `cat_exp`.

`void print(ostream& os, int i) const`

Prints an alphanumeric representation of the invoking `cat_exp` on the stream `os`. The symbol for the union operation is defined in the variable `re<S>::re_cat`.

`int size() const`

Returns 1 plus the size of `left` and the size of `right`.

`cat_exp()`

Constructor. Assigns `left` and `right` to `null_exp`.

`cat_exp(const re<S>& l, const re<S>& r)`

Copy constructor. Class `clone` on each of `l` and `r`.

`cat_exp(const cat_exp<S>&)`

Copy constructor. Calls `clone`.

`~cat_exp()`

Destructor. Explicitly calls the destructors for the `left` and `right` `subexp`.

B Empty set expressions: empty_set

B.1 Definition

empty_sets are parameterizable empty set subexpressions of regular expressions. empty_set maintains no variables; it simply exists to stand for an empty set, and to define the value of some comparison functions.

B.2 Public functions

`empty_set<S>& operator=(const empty_set<S>&)`

Assignment operator. A no-op.

`int operator==(const subexp<S>* r) const`

Equivalence operator. Calls the argument's compare_empty_set.

`int operator!=(const subexp<S>* r) const`

Inequivalence operator. Calls the argument's compare_empty_set.

`int operator<(const subexp<S>* r) const`

Less-than operator. Calls the argument's compare_empty_set.

`int operator>(const subexp<S>* r) const`

Greater-than operator. Calls the argument's compare_empty_set.

`subexp<S>* clone() const`

Returns a new empty_set.

`int compare_empty_set() const`

Returns 0 (every empty_set is equal to every other).

`int compare_empty_string() const`

Returns -1 (every empty_set is less than every empty_string).

`int compare_cat_exp(const subexp<S>*, const subexp<S>*) const`

Returns -1 (every empty_set is less than every cat_exp).

`int compare_plus_exp(const subexp<S>*, const subexp<S>*) const`

Returns -1 (every empty_set is less than every plus_exp).

`int compare_star_exp(const subexp<S>*) const`

Returns -1 (every empty_set is less than every star_exp).

`int compare_symbol_exp(const S&) const`

Returns -1 (every empty_set is less than every symbol_exp).

int contains_empty_set() const

Returns 1.

void convert_subexp(fm<S>& a) const

Converts the invoking empty_set into a fm and returns the result in a.

int is_empty_set() const

Returns 1.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new empty_set.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the empty_set on the stream os. The representation used is defined by the variable re<S>::re_empty_set.

int size() const

Returns 1.

empty_set()

Constructor. A no-op.

empty_set(const empty_set<S>&)

Copy constructor. A no-op.

~empty_set()

Destructor. A no-op.

C Empty string expressions: empty_string

C.1 Definition

empty_strings are parameterizable empty string subexpressions of regular expressions. empty_string maintains no variables; it simply exists to stand for an empty string, and to define the value of some comparison functions.

C.2 Public functions

```
empty_string<S>& operator=(const empty_string<S>&)
```

Assignment operator. A no-op.

```
int operator==(const subexp<S>* r) const
```

Equivalence operator. Calls the argument's compare_empty_string.

```
int operator!=(const subexp<S>* r) const
```

Inequivalence operator. Calls the argument's compare_empty_string.

```
int operator<(const subexp<S>* r) const
```

Less-than operator. Calls the argument's compare_empty_string.

```
int operator>(const subexp<S>* r) const
```

Greater-than operator. Calls the argument's compare_empty_string.

```
subexp<S>* clone() const
```

Returns a new empty_string.

```
int compare_empty_string() const
```

Returns 0 (every empty_string is equal to every other).

```
int compare_cat_exp(const subexp<S>*, const subexp<S>*) const
```

Returns -1 (every empty_string is less than every cat_exp).

```
int compare_plus_exp(const subexp<S>*, const subexp<S>*) const
```

Returns -1 (every empty_string is less than every plus_exp).

```
int compare_star_exp(const subexp<S>*) const
```

Returns -1 (every empty_string is less than every star_exp).

```
int compare_symbol_exp(const S&) const
```

Returns -1 (every empty_string is less than every symbol_exp).

```
int contains_empty_string() const
```

Returns 1.

void convert_subexp(fm<S>& a) const

Converts the invoking empty_string into a fm and returns the result in a.

int is_empty_string() const

Returns 1.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new empty_string.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the empty_string on the stream os. The representation used is defined by the variable re<S>::re_empty_string.

int size() const

Returns 1.

empty_string()

Constructor. A no-op.

empty_string(const empty_string<S>&)

Copy constructor. A no-op.

~empty_string()

Destructor. A no-op.

D Finite-state machines: fm

D.1 Definition

fms are parameterizable finite-state machines. fms consist of a set of instructions whose label type is specified by the parameter.

fms can have multiple final states, as is customary, but they can also have multiple start states. By definition, any fm with more than one start state is nondeterministic. fms contain pseudo-instructions to denote the states that are start and final.

fm maintains the following variables:

```
protected:

set<inst<Label> > arcs;
```

D.2 Public functions

`fm<Label>& operator=(const fm<Label>& a)`

Assignment operator. Checks for self-assignment, and then copies a to the invoking fm.

`int operator==(const fm<Label>& a)`

Equivalence operator. Returns 1 if a is identical to the invoking machine, and returns 0 otherwise. Note that this operator checks for identity, not for language equivalence.

`int operator!=(const fm<Label>& a)`

Inequivalence operator. Returns 1 if a is different from the invoking machine, and returns 0 otherwise. Note that this operator checks for identity, not language equivalence.

`fm<Label>& operator+=(const fm<Label>& a)`

Returns the union of a with the invoking machine.

`fm<Label>& operator^=(const fm<Label>& a)`

Catenation operator. Catenates a with the invoking machine. Computes the Cartesian product of penultimate states of the invoking fm with the start states of a. Does not introduce empty-string instructions.

`fm<Label>& operator-=(const fm<Label>& a)`

Difference operator. Deletes instructions in the invoking machine that are also present in a.

`inst<Label>& operator[](int i) const`

Selection operator. Returns the ith instruction in the invoking machine.

void cartesian(const set<state>&, const set<Label>&, const set<state>&)

Assigns the Cartesian product of the arguments to the invoking fm.

int canonical_numbering()

Renumbers all states according to a breadth-first traversal of the fm. Will not renumber a nondeterministic fm.

void clear()

Clears the set of arcs.

void complement()

Complements the invoking fm. Assumes that the alphabet is defined by the set of Labels already present in the fm.

void complete()

Completes the invoking fm—that is, it ensures that each state has a instruction on each Label in the alphabet. Assumes that the alphabet is defined by the set of Labels already present in the fm.

void cross_product(const fm<Label>&, fm<Label>)

Assigns the cross product of the two argument fms to the invoking fm.

fm<Label>& disjoint_union(const fm<Label>& t)

Efficient union of t with the invoking fm. It is the programmer's responsibility to ensure that the two machines are disjoint.

fm<Label>& disjoint_union(const inst<Label>& a)

Efficient union of a with the invoking fm. It is the programmer's responsibility to ensure that a is not contained in the invoking machine.

fm<Label>& empty_string_machine()

Makes the invoking machine one that accepts only the empty string.

int enumerate(int i, set<string<Label> >& s) const

Generate the first i strings in the language of the machine and return them in s. Strings are ordered first according to size, then lexicographically.

set<state>& finals(set<state>&) const

Return the set of final states in the invoking machine.

int is_complete() const

Returns 1 if the invoking machine is complete, and returns 0 otherwise.

int is_deterministic() const

Returns 1 if the invoking machine is deterministic, and returns 0 otherwise.

`int is_universal() const`

Returns 1 if the invoking machine is universal, and returns 0 otherwise.

`set<Label>& labels(set<Label>&) const`

Return the set of `Labels` in the invoking machine.

`fm<Label>& empty_string_machine()`

Makes the invoking machine one that accepts only the empty string.

`state max_state()`

Returns the maximum `state`.

`int member_of_language(char* s, int d) const`

Returns 1 if `s` is a member of the language of the machine; returns 0 otherwise. If `d` is 1, then the function prints diagnostic statements on standard output describing its traversal of the machine.

`fm<Label>& min_by_partition()`

Minimizes the invoking machine according to Hopcroft's partition method. Should only be applied to deterministic machines.

`int number_of_final_states() const`

Returns the number of final `states` in the invoking machine.

`int number_of_labels() const`

Returns the number of distinct `Labels` in the invoking machine.

`int number_of_start_states() const`

Returns the number of start `states` in the invoking machine.

`int number_of_states() const`

Returns the number of `states` in the invoking machine.

`int number_of_instructions() const`

Returns the number of non-pseudo instructions in the invoking machine (the total number of instructions can be found by executing `arcs.size()`.

`void plus()`

Computes the '+' of the invoking machine; that is, it converts the invoking machine into one that accepts strings that are catenations of one or more strings in the original machine.

`void reachable_fm()`

Reduces the invoking machine to the subset of instructions that correspond to reachable states.

void reachable_states(set<state>& s) const

Computes the set of reachable states and assigns them to s.

void remove(const state& s)

Removes from the invoking machine any instruction that refers to state s.

void renumber(int i)

Renumbers the invoking machine by adding i to the states.

void reverse()

Reverses the invoking machine. Note that this may result in multiple start states (and hence, a nondeterministic machine).

fm<Label>& select(const state& s, int w, fm<Label>& a) const

Returns in a the submachine consisting of instructions that refer to the state s. w specifies that s is to be a source state, sink state, or either.

fm<Label>& select(const Label& l, fm<Label>& a) const

Returns in a the submachine consisting of instructions whose Label is l.

fm<Label>& select(const Label& l, const state& s, int w, fm<Label>& a) const

Returns in a the submachine consisting of instructions whose Label is l and which refer to the state s. w specifies that s is to be either a source state or a sink state.

fm<Label>& single(const Label& r)

Makes the invoking machine a single-instruction machine with the instruction Label being r.

set<state>& sinks(set<state>& s) const

Returns the set of sink states in the invoking machine in s (a sink state is a state on the right hand side of a regular instruction).

int size() const

Returns the size of the invoking machine.

set<state>& sources(set<state>& s) const

Returns the set of source states in the invoking machine in s (a source state is a state on the left hand side of a regular instruction).

fm<Label>& star()

Computes '*' of invoking machine; that is, it converts the invoking machine into one that accepts strings that are catenations of zero or more strings in the original machine.

`set<state>& starts(set<state>& s) const`

Returns the set of start states of the invoking machine in `s`.

`set<state>& states(set<state>&) const`

Returns the set of states of the invoking machine in `s`.

`fm<Label>& subset()`

Converts the invoking (nondeterministic) machine into a deterministic machine by subset construction.

`fm()`

Constructor. A no-op.

`fm(fm<Label>& a)`

Copy constructor. Copies the set of instructions.

`~fm()`

Destructor. A no-op.

D.3 Private functions

`int find_part(set<set<state> >& p, state s)`

Finds the member of the partition `p` containing the state `s`. Returns the partition index if successful, and −1 otherwise. Used by `min_by_partition`.

`void merge_inverse(set<set<state> >& p, set<int>& k, set<state>& s)`

Given a set of states `s`, merge it with the existing partition `p`. Adjusts the index of partition elements (`k`) that must be processed in successive steps of the minimization. Used by `min_by_partition`.

D.4 Friend functions

`ostream& operator<<(ostream& os, const fm<Label>& s)`

Outputs `s` on stream `os`.

`istream& operator>>(istream& os, fm<Label>& s)`

Inputs `s` from stream `is`.

E Instructions: inst

E.1 Definition

insts are parameterizable instructions in a finite-state machine. Each instruction consists of two states (a source state and a sink state) and the instruction label, which is the template parameter.

inst provides support for pseudo-start and pseudo-final instructions. These instructions use the form of an instruction to denote the start and final states in a finite-state machine:

```
(START) |- 5
7 -| (FINAL)
```

The instruction labels for these pseudo-instructions are purely decorative, but can be thought of as 'end markers' on an input tape. The tokens (START) and (FINAL) represent the pseudo-start and pseudo-final states, respectively (they are represented by state values of 1 and 0, respectively). inst maintains the following private variables:

```
private:

state       source;         // source state
Label       label;          // instruction label
state       sink;           // sink state
```

The following public variables are maintained:

```
static      char      left_delimiter;
static      char      right_delimiter;
```

E.2 Public functions

`inst<Label>& operator=(const inst<Label>& t)`

Assignment operator. Checks for self assignment; then assigns components of t to the invoking inst.

`int operator==(const inst<Label>& t)`

Returns 1 if source, sink, and label of the invoking inst are equivalent to those of t and otherwise returns 0.

`int operator!=(const inst<Label>& t)`

Returns 0 if source, sink, and label are equivalent to those of t, and otherwise returns 1.

`void assign(const state& s1, const Label& r, const state& s2)`

Assigns the argument values to the invoking inst.

`Label& get_label()`

Returns `label`.

`state& get_sink()`

Returns `sink`.

`state& get_source()`

Returns `source`.

`state is_final()`

Returns 1 if the invoking `inst` is a pseudo-final instruction, and otherwise returns 0.

`state is_start()`

Returns 1 if the invoking `inst` is a pseudo-start instruction, and otherwise returns 0.

`state is_null()`

Returns 1 if the invoking `inst` is a null instruction, and otherwise returns 0.

`int labelis(const Label& l)`

Returns 1 if `label` is equivalent to `l`, and 0 otherwise.

`void make_final(const state& s)`

Makes the invoking `inst` a pseudo-final instruction with a `source` of `s`.

`void make_start(const state& s)`

Makes the invoking `inst` a pseudo-start instruction with a `sink` of `s`.

`void renumber(int bottom)`

Renumbers the states in the invoking `inst` by adding `bottom` to their value.

`void reverse()`

Swap start and final `states` of the invoking `inst`. Converts pseudo-start instructions to pseudo-final instructions and vice versa.

`void null()`

Makes the invoking `inst` null.

`int sinkis(const state& s)`

Returns 1 if `sink` of the invoking `inst` is equivalent to `s`, and returns 0 otherwise.

`int sourceis(const state& s)`

Returns 1 if `source` of the invoking `inst` is equivalent to `s`, and returns 0 otherwise.

inst()

Constructor. A no-op.

inst(const state& s1, const Label& r, const state& s2)

Constructor with initializers.

inst(const inst<Label>& t)

Copy constructor.

~inst()

Destructor. A no-op.

E.3 Friend functions

ostream& operator<<(ostream& os, const inst& t)

Outputs t on stream os. Correctly handles pseudo-start and pseudo-final instructions.

istream& operator>>(istream& os, inst& t)

Inputs t from stream is. Correctly handles pseudo-start and pseudo-final instructions.

F Lists: list

F.1 Definition

lists are parameterizable, dynamic, homogeneous lists of Items.

Each list stores its objects directly. If you want to use a list to share objects with some other container, you should declare a list that stores pointers or references to the objects you want to share. lists can contain multiple copies of a object, and can be sorted. The order in which objects are appended to a list is preserved.

It is possible to convert a list to a set without copying all the elements. This is because sets and lists are both implemented with a pointer to an array of the contained objects; thus, it is possible to simply copy the pointer, and leave the array intact. The from_set function converts a set to a list. list maintains the following variables:

```
protected:

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.

F.2 Public functions

`void clear()`

Sets the size to 0. Does not free any space used by current members.

`int contain(const list<Item>& s) const`

Checks to see if s is contained in the invoking list. Returns 1 if s is contained, and returns 0 otherwise.

`void intersect(const list<Item>& s1, const list<Item>& s2)`

Clears the invoking list, then adds any members belonging to the intersection of s1 and s2.

`int is_sorted()`

Returns 1 if the invoking list is sorted, and returns 0 otherwise.

`static int compare(const Item*, const Item*)`

Comparison of two Items. Returns 1 if the first argument is greater than the second, 0 if the two arguments are equal, and −1 if the second argument is greater than the first. This function is static so that its pointer can be passed as an argument to qsort().

```
int member(const Item& s)
```

Returns 1 if s is a member of the invoking list, and returns 0 otherwise.

```
list<Item>& operator=(const list<Item>& s)
```

Assignment operator. Checks for self-assignment, clears, and adds s to the invoking list.

```
list<Item>& operator=(const Item& i)
```

Assignment operator. Checks for self-assignment, clears, and adds i to the invoking list.

```
int operator==(const list<Item>& s) const
```

Equivalence operator. Returns 1 if s and the invoking list contain exactly the same Items in the same order, and returns 0 otherwise.

```
int operator!=(const list<Item>& s) const
```

Inequivalence operator. Returns 1 if s and the invoking list do not contain exactly the same Items in the same order, and returns 0 otherwise.

```
int operator<(const list<Item>& s) const
```

Less-than operator. Returns 1 if the invoking list is less than s, and returns 0 otherwise. list a is less than list b if a has fewer members than b, or if $a_i < b_i$ and a_i and b_i are the smallest elements of a and b that are not contained in both. This function is used when sorting collections of lists.

```
int operator>(const list<Item>&) const
```

Greater-than operator. Returns 1 if the invoking list is greater than s, and returns 0 otherwise. list a is greater than list b if a has more members than b, or if $a_i > b_i$ and a_i and b_i are the smallest elements of a and b that are not contained in both. This function is used when sorting collections of lists.

```
void operator+=(const list<Item>& s)
```

Union operator. Checks for self assignment, and adds each member of s to the invoking list.

```
void operator+=(Item q)
```

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

```
void operator-=(const list<Item>& s)
```

Difference operator. Checks for self-deletion, and then deletes each member of s from the invoking list.

`void operator-=(const Item& s)`

Difference operator. Checks for membership of `s` in the invoking `list`, and then deletes it.

`Item& 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.

`void remove(int i)`

Removes the ith `Item` from the invoking `list`. This function is not defined as an overloaded `operator-=` in order to avoid ambiguity; in particular, removing the `i`th `Item` from a `list` of `int` would not be distinguishable from removing `i` itself from the `list`.

`int size() const`

Returns the current size of the invoking `list`.

`void sort() const`

Sorts the `Item`s of the invoking `list`. Calls `qsort()` to do the sorting.

`void unique() const`

Removes duplicate `Item`s from the invoking `list`. This function first sorts the list, so the order of the `Item`s is not retained.

`void from_set(set<Item>&)`

Efficiently converts a `set` to a `list`. Note that the `set` is no longer available after this call; the array of `Item`s in the `set` has been transferred directly to the `list`.

`list()`

Constructor. Allocates space and sets the size to 0.

`list(const list<Item>& s)`

Copy constructor. Allocates space and copies `s` to the invoking list.

`~list()`

Destructor. Deletes the array of `Item`s.

F.3 External functions

`ostream& operator<<(ostream& os, list<Item>& s`

Outputs `s` on stream `os`.

G Null expressions: null_exp

G.1 Definition

null_exps are parameterizable null subexpressions of regular expressions. null_exp are used as initializers for regular expressions when no other value is available.

G.2 Public functions

```
null_exp<S>& operator=(const null_exp<S>&)
```

Assignment operator. A no-op.

```
int operator==(const subexp<S>* r) const
```

Equivalence operator. Calls the argument's compare_null_exp.

```
int operator!=(const subexp<S>* r) const
```

Inequivalence operator. Calls the argument's compare_null_exp.

```
int operator<(const subexp<S>* r) const
```

Less-than operator. Calls the argument's compare_null_exp.

```
int operator>(const subexp<S>* r) const
```

Greater-than operator. Calls the argument's compare_null_exp.

```
subexp<S>* clone() const
```

Returns a new null_exp.

```
int compare_null_exp() const
```

Returns 0 (every null_exp is equal to every other).

```
int compare_empty_set() const
```

Returns -1 (every null_exp is less than every empty_set).

```
int compare_empty_string() const
```

Returns -1 (every null_exp is less than every empty_string).

```
int compare_cat_exp(const subexp<S>*, const subexp<S>*) const
```

Returns -1 (every null_exp is less than every cat_exp).

```
int compare_plus_exp(const subexp<S>*, const subexp<S>*) const
```

Returns -1 (every null_exp is less than every plus_exp).

```
int compare_star_exp(const subexp<S>*) const
```

Returns -1 (every null_exp is less than every star_exp).

void convert_subexp(fm<S>& a) const

A no-op.

int is_null_exp() const

Returns 1.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new **null_exp**.

void print(ostream& os, int i) const

A no-op.

int size() const

Returns 0.

null_exp()

Constructor. A no-op.

null_exp(const null_exp<S>&)

Copy constructor. A no-op.

~null_exp()

Destructor. A no-op.

H Union expressions: plus_exp

H.1 Definition

plus_exps are parameterizable union subexpressions of regular expressions (re). plus_exp maintains two variables:

```
protected:

subexp<S>* left;
subexp<S>* right;
```

H.2 Public functions

`plus_exp<S>& operator=(const plus_exp<S>& r) const`

Assignment operator. Assigns the left and right of r to the left and right of the invoking plus_exp.

`int operator==(const subexp<S>* r) const`

Equivalence operator. Calls the argument's compare_plus_exp.

`int operator!=(const subexp<S>* r) const`

Inequivalence operator. Calls the argument's compare_plus_exp.

`int operator<(const subexp<S>* r) const`

Less-than operator. Calls the argument's compare_plus_exp.

`int operator>(const subexp<S>* r) const`

Greater-than operator. Calls the argument's compare_plus_exp.

`int compare_plus_exp(const subexp<S>*, const subexp<S>*) const`

Returns 0 if the invoking plus_exp is equal to the arguments, -1 if it is less than the arguments, and 1 if it is greater than the arguments.

`int compare_star_exp(const subexp<S>*) const`

Returns -1 (every plus_exp is less than every star_exp).

`subexp<S>* clone()`

Clone operation. Simulates virtual copy constructor.

`int contains_empty_string() const`

Returns 1 if the left or right contains the empty string, and returns 0 otherwise.

`int contains_plus_exp() const`

Returns 1 if the left or right contains the empty set, and returns 0 otherwise.

void copy(const subexp<S>&)

Copy operation. Used by clone.

int is_empty_string() const

Returns 1 if left and right are the empty string, and returns 0 otherwise.

int is_plus_exp() const

Returns 1 if left and right contains the empty set, and returns 0 otherwise.

subset<S>* minimize()

Applies minimization heuristics.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new cat_exp.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the plus_exp on the stream os. The symbol for the union operation is defined in the variable re<S>::re_plus.

int size() const

Returns 1 plus the size of left and the size of right.

plus_exp()

Constructor. Assigns left and right to null_exp.

plus_exp(const re<S>& l, const re<S>& r)

Copy constructor. Copies the subexp* of l to left and the subexp* of r to right.

plus_exp(const plus_exp<S>&)

Copy constructor. Calls clone.

~plus_exp()

Destructor. Explicitly calls the destructors for the left and right subexp.

I Regular expressions: re

I.1 Definition

res are parameterizable regular expressions. re maintains the following variable:

```
protected:

subexp<S>* p;
```

A subexp is a subexpression (also a template class). There are several derivations of subexp; p can point to any one of them. re also maintains the following static variables:

```
public:

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

I.2 Public functions

`re<S>& operator=(const re<S>& r)`

Assignment operator. Checks for self-assignment, and then copies r to the invoking fm.

`re<S>& operator=(subexp<S>& r)`

Assignment operator. Checks for self-assignment, and then copies r to the invoking re. Actually just copies the subexpression pointer.

`int operator==(const re<S>& r) const`

Equivalence operator. Returns 1 if the invoking re is equal to r, and 0 otherwise. Returns 0 if either re is null.

`int operator!=(const re<S>& r) const`

Inequivalence operator. Returns 1 if the invoking re is not equal to r, and 0 otherwise. Returns 0 if either re is null.

int operator<(const re<S>& r) const

Less-than operator. Returns 1 if the invoking re is less than r, and 0 otherwise. Returns 0 if either re is null.

int operator>(const re<S>& r) const

Greater-than operator. Returns 1 if the invoking re is greater than r, and 0 otherwise. Returns 0 if either re is null.

re<S>& operator^(const re<S>& r)

Catenation operator. Catenates the invoking re with r.

re<S>& operator+(const re<S>& r)

Union operator. Computes the union of the invoking re with r.

re<S>& operator^=(const re<S>& r)

Catenation operator. Catenates the invoking re with r, without producing an intermediate re.

re<S>& operator+=(const re<S>& r)

Union operator. Computes the union of the invoking re with r, without producing an intermediate re.

void clear()

Clears the content of the re.

int contains_empty_set() const

Returns 1 if the invoking re contains the empty set, and 0 otherwise. Returns 0 if the invoking re is null.

int contains_empty_string() const

Returns 1 if the invoking re contains the empty string, and 0 otherwise. Returns 0 if the invoking re is null.

void fmtore(fm<S>& a)

Converts finite-state machine a to an re, and returns it as the invoking re.

int is_empty_set() const

Returns 1 if the invoking re is the empty set, and 0 otherwise. Returns 0 if the invoking re is null.

int is_empty_string() const

Returns 1 if the invoking re is the empty string, and 0 otherwise. Returns 0 if the invoking re is null.

int is_null() const

Returns 1 if the invoking re is null (that is, uninitialized) and 0 otherwise.

re<S>& make_empty_string() const

Make the empty string re.

re<S>& make_empty_set() const

Make the empty set re.

re<S>& make_null_exp() const

Makes the null (uninitialized) re.

re<S>& make_symbol(const S& s) const

Makes a single symbol re, using symbol s.

re<S>& minimize()

Applies minimization heuristics (removing subexpressions that are catenated with empty_set, removing empty_string from catenations, removing union of equivalent expressions, eliminating unnecessary parentheses).

re<S>* parse(char* str, int* i, int size)

Parses the string str of size size, starting at position i, and returns the corresponding subexp.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the re on the stream os. i is the priority used to determine whether the expression should be surrounded by parentheses.

fm<S>& retofm() const

Converts the invoking re to a finite-state machine. Employs the convert_subexp functions of the subexpressions of the invoking re.

int size() const

Returns the size of the invoking re.

re<S>& star()

Computes the Kleene star of the invoking re.

re<S>* term(char* str, int* i, int size)

Finds the next term in str starting from i, and returns a pointer to the corresponding subexp. Used by parse.

token_type token(char* str, int& i)

Finds the next token in str starting from i, and returns an indicator of the type of the token. Used by parse and term.

re()

Constructor. Initializes p to empty_set.

```
re(const re<S>& a)
```

Copy constructor. Tests for equivalence, and then copies the subexpression pointer.

```
~re()
```

Destructor. Explicitly deletes the subexpression through its pointer.

I.3 Friend functions

```
ostream& operator<<(ostream& os, const re<S>& s)
```

Outputs s on stream os.

```
istream& operator>>(istream& os, re<S>& s)
```

Inputs s from stream is.

J Sets: set

J.1 Definition

sets are parameterizable, dynamic, homogeneous sets of Items. Each set stores its objects directly. If you want to use a set to share objects with some other container, you should declare a set that stores pointers or references to the objects you want to share.

It is possible to convert a set to a list without copying all the elements. This is because lists and setss are both implemented with a pointer to an array of the contained objects; thus, it is possible to simply copy the pointer, and leave the array intact. The from_list function converts a list to a set.

set maintains the following variables:

```
protected:

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.

J.2 Public functions

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

Assignment operator. Checks for self-assignment; clears; adds s to the invoking set.

```
set<Item>& operator=(const Item& i)
```

Assignment operator. Checks for self-assignment; clears; adds i to the invoking set.

```
int operator==(const set<Item>& s) const
```

Equivalence operator. Returns 1 if s and the invoking set contain exactly the same Items, and returns 0 otherwise.

```
int operator!=(const set<Item>& s) const
```

Inequivalence operator. Returns 1 if s and the invoking set do not contain exactly the same Items, and and returns 0 otherwise.

```
int operator<(const set<Item>& s) const
```

Less-than operator. Returns 1 if the invoking set is less than s, and returns 0 otherwise. set a is less than set b if a has fewer members than b, or if $a_i < b_i$ and a_i and b_i are the smallest elements of a and b that are not contained in both. This function is used when sorting collections of sets.

```
int operator>(const set<Item>&) const
```

Greater-than operator. Returns 1 if the invoking set is greater than s, and returns 0 otherwise. set a is greater than set b if a has more members than b, or if $a_i > b_i$ and a_i and b_i are the smallest elements of a and b that are not contained in both. This function is used when sorting collections of sets.

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

Union operator. Checks for self assignment, and adds each member of s to the invoking set.

```
void operator+=(Item q)
```

Union operator. 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)
```

Difference operator. Checks for self-deletion, and then deletes each member of s from the invoking set.

```
void operator-=(const Item& s)
```

Difference operator. Checks for membership of s in the invoking set, and then deletes it.

```
Item& operator[](int i) const
```

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

```
void clear()
```

Sets the size to zero. Does not free any space used by current members.

```
int contain(const set<Item>& s) const
```

Checks to see if s is contained in the invoking set. Returns 1 if s is contained, and returns 0 otherwise.

```
set<Item>& disjoint_union(const set<Item>& s)
```

Computes a fast union of s with the invoking set, based on the assumption that the two sets are disjoint. It is the programmer's responsibility to ensure that the sets are disjoint.

```
set<Item>& disjoint_union(const Item& s)
```

Computes a fast union of s with the invoking set, based on the assumption that s does not appear in the invoking set. It is the programmer's responsibility to ensure that s does not appear in the invoking set.

void from_list(list<Item>&)

Efficiently converts a list to a set. Duplicates are removed. Note that the list is no longer available after this call; the array of Items has been transferred directly to the set.

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 invoking set. Returns 1 if s is a member, and returns 0 otherwise.

void remove(int i)

Removes the ith Item from the invoking set. This function is not an overloaded operator-= in order to avoid ambiguity; in particular, removing the ith member of a set of int would not be distinguishable from removing I itself from the set.

int size() const

Returns the current size of the invoking 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.

J.3 External functions

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

Outputs s on stream os.

K Star expressions: star_exp

K.1 Definition

star_exps are parameterizable star (or closure) subexpressions of regular expressions. star_exp maintains one variable:

```
protected:

subexp<S>* left;
```

K.2 Public functions

`star_exp<S>& operator=(const star_exp<S>& r) const`

Assignment operator. Assigns left of r to the left of the invoking star_exp.

`int operator==(const subexp<S>* r) const`

Equivalence operator. Calls the argument's compare_star_exp.

`int operator!=(const subexp<S>* r) const`

Inequivalence operator. Calls the argument's compare_star_exp.

`int operator<(const subexp<S>* r) const`

Less-than operator. Calls the argument's compare_star_exp.

`int operator>(const subexp<S>* r) const`

Greater-than operator. Calls the argument's compare_star_exp.

`int compare_star_exp(const subexp<S>*) const`

Returns -1 if the invoking star_exp is less than the argument, 0 if equal to the argument, and 1 if greater than the argument.

`subexp<S>* clone()`

Clone operation. Simulates virtual copy constructor.

`int contains_empty_string() const`

Returns 1 if left contains the empty string, and returns 0 otherwise.

`int contains_star_exp() const`

Returns 1 if left contains the empty set, and returns 0 otherwise.

`void convert_subexp(fm<S>& a) const`

Converts the invoking star_exp into a fm and returns the result in a. Makes calls to other subexpression classes.

void copy(const subexp<S>&)

Copy operation. Used by clone.

int is_empty_string() const

Returns 1 if left is the empty string, and returns 0 otherwise.

int is_star_exp() const

Returns 1 if left is the empty set, and returns 0 otherwise.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new star_exp.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the star_exp on the stream os. The symbol for the star operation is defined in the variable re<S>::re_star.

int size() const

Returns 1 plus the size of left.

subexp<S>* star()

Star operation. A no-op, since star of star is still star.

star_exp()

Constructor. Assigns left to null_exp.

star_exp(const re<S>& s)

Copy constructor. Assigns left of s to left.

star_exp(const star_exp<S>& s)

Copy constructor. Calls clone.

~star_exp()

Destructor. Explicitly calls the destructor for left.

L States: state

L.1 Definition

states are the states of finite-state machines. This class is a simple wrapper for ints. It exists for two reasons: first, to ensure that no code is written that embeds knowledge of the representation of states—as might happen if states were represented as ints, for example. The second reason is to support the pseudo-states (START) and (FINAL).

states can have the value of any non-negative integer. Internally, each state's value is offset by two; the values zero and one are reserved for the pseudo-start state and pseudo-final state, respectively. There is also a null state, whose value is -1; this state is used by functions whose return value is state, and who wish to signal an exceptional condition.

state maintains the following private variable:

```
private:

int number; // state number
```

L.2 Public functions

`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.

`int operator==(const state& s)`

Returns 1 if the invoking state is equal to s; returns 0 otherwise.

`int operator!=(const state& s)`

Returns 1 if the invoking state is not equal to s; returns 0 otherwise.

`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 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.

`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 `i` from the invoking `state`.

`int is_null() const`

Returns 1 if the invoking `state` is null, and returns 0 otherwise.

`int is_final() const`

Returns 1 if the invoking `state` is pseudo-final, and returns 0 otherwise.

`int is_start() const`

Returns 1 if the invoking `state` is pseudo-start, and returns 0 otherwise.

`void final()`

Sets the invoking `state` to pseudo-final value.

`void null()`

Sets the invoking `state` to null.

`void start()`

Sets the invoking `state` to pseudo-start value.

`int value() const`

Returns integer value of the invoking `state`.

`state()`

Constructor. Sets `number` to 0.

`state(const state& s)`

Copy constructor. Copies value of `s` to the invoking `state`.

`~state()`

Destructor. A no-op.

L.3 Friend functions

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

Outputs `s` on stream `os`.

`istream& operator>>(istream& os, state& s)`

Inputs `s` from stream `is`.

M Strings: string

M.1 Definition

strings are parameterizable dynamic arrays of symbols. *Grail*'s strings use a slightly unconventional notion of order; strings are ordered first according to size and then lexicographically. This is done in order to make it easier to enumerate the strings belonging to languages.

strings are not null-terminated, as are char *s. string maintains the following private variables:

```
private:

S*        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.

M.2 Public functions

`void operator=(const string<S>& s)`

Assignment operator. Checks for self-assignment, clears, and then assigns s to the invoking string.

`void operator=(const S* s)`

Assignment operator. Clears, then assigns s to the invoking string.

`int operator==(const string<S>& s)`

Equivalence operator. Returns 1 if s is identical to the invoking string, and returns 0 otherwise.

`int operator==(const S* s)`

Equivalence operator. Returns 1 if s is identical to the invoking string, and returns 0 otherwise.

`int operator<(const string<S>& s)`

Less-than operator. Returns 1 if invoking string is less than s, and returns 0 otherwise. (strings are ordered first by size, then lexicographically.)

`int operator>(const string<S>& s)`

Greater-than operator. Returns 1 if invoking string is greater than s, and returns 0 otherwise. (strings are ordered first by size, then lexicographically.)

int operator!=(const string<S>& s)

Inequivalence operator. Returns 1 if s is different from the invoking string, and returns 0 otherwise.

int operator!=(const S* s

Inequivalence operator. Returns 1 if s is different from the invoking string, and returns 0 otherwise.

int operator+=(const string<S>& str)

Catenation operator. Append str to the invoking string.

int operator+=(const S* str)

Catenation operator. Append str to the invoking string.

int operator+=(const char& ch)

Catenation operator. Append the character ch to the invoking string.

S& operator[](int& i) const

Selection operator. Returns the ith S in the string.

S* ptr() const

Returns a pointer to the S array.

void clear()

Sets the current length to 0.

int is_null()

Returns 1 if the invoking string is empty, and returns 0 otherwise.

int size() const

Returns the current size of the invoking string.

int truncate(int x)

Truncation. Sets size to x (x may be zero).

string()

Constructor. Allocates space and sets size to zero.

string(const string<S>& s)

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

~string()

Destructor. Deletes space occupied by c.

M.3 Friend functions

```
ostream& operator<<(ostream& os, const string<S>& s)
```

Outputs `s` on stream `os`.

```
istream& operator>>(istream& os, string& s)
```

Inputs `s` from stream `is`. Treats either whitespace or pairs of " as string delimiters.

N Subexpressions: subexp

N.1 Definition

subexps are parameterizable subexpressions of regular expressions (subexp). subexp is the abstract base class for empty_set, empty_string, symbol_exp, plus_exp, cat_exp, star_exp). subexp is an abstract base class and cannot be instantiated.

Many of the functions of subexp return 0 as a default value. These functions are overridden as appropriate by the derived classes.

N.2 Public functions

```
virtual int operator==(const subexp<S>* r) const = 0
```

Pure virtual function.

```
virtual int operator!=(const subexp<S>* r) const = 0
```

Pure virtual function.

```
virtual int operator<(const subexp<S>* r) const = 0
```

Pure virtual function.

```
virtual int operator>(const subexp<S>* r) const = 0
```

Pure virtual function.

```
subexp<S>* operator∧(subexp<S>& r)
```

Catenation operator. Catenates the invoking subexp with r.

```
subexp<S>* operator+(subexp<S>& r)
```

Union operator. Computes the union of the invoking subexp with r.

```
virtual int compare_null_exp() const
```

Returns 1.

```
virtual int compare_empty_string() const
```

Returns 1.

```
virtual int compare_empty_set() const
```

Returns 1.

```
virtual int compare_symbol_exp(const S&) const
```

Returns 1.

```
virtual int compare_cat_exp(const subexp<S>*, const subexp<S>*) const
```

Returns 1.

`virtual int compare_plus_exp(const subexp<S>*, const subexp<S>*) const`

Returns 1.

`virtual int compare_star_exp(const subexp<S>*) const`

Returns 1.

`virtual subexp<S>* clone()`

Clone operation. Simulates virtual copy constructor.

`virtual int contains_empty_set() const`

Returns 0.

`virtual int contains_empty_string() const`

Returns 0.

`virtual void convert_subexp(fm<S>&) const = 0`

Pure virtual function.

`void copy(const subexp<S>&)`

Copy operation. Used by `clone`.

`virtual int is_empty_set() const`

Returns 0.

`virtual int is_empty_string() const`

Returns 0.

`virtual subexp<S>* minimize()`

Returns `this`.

`virtual subexp<S>* new_subexp()`

Creation function. Simulates virtual constructor.

`virtual void print(ostream& os, int i) const = 0`

Pure virtual function.

`virtual int size() const = 0`

Pure virtual function.

`virtual subexp<S>* star()`

Computes the Kleene star of the invoking `subexp`.

`subexp()`

Constructor. A no-op. Protected (to ensure that instances of `subexp` are not created)

`virtual ~subexp()`

Destructor. A no-op.

O Symbol expressions: symbol_exp

O.1 Definition

symbol_exps are parameterizable symbol subexpressions of regular expressions. symbol_exp maintains one variable:

```
protected:

S content;
```

O.2 Public functions

`int operator=(const symbol_exp& s)`

Assignment operator. Assigns s's content to content.

`int operator==(const subexp<S>* r) const`

Equivalence operator. Calls the argument's compare_symbol_exp.

`int operator!=(const subexp<S>* r) const`

Inequivalence operator. Calls the argument's compare_symbol_exp.

`int operator<(const subexp<S>* r) const`

Less-than operator. Calls the argument's compare_symbol_exp.

`int operator>(const subexp<S>* r) const`

Greater-than operator. Calls the argument's compare_symbol_exp.

`compare_symbol_exp(const S& s) const`

Returns 0 if s == content, −1 if s < content, and 1 if s > content

`compare_cat_exp(const subexp<S>*, const subexp<S>*) const`

Returns −1 (every symbol_exp is less than every cat_exp).

`compare_plus_exp(const subexp<S>*, const subexp<S>*) const`

Returns −1 (every symbol_exp is less than every plus_exp).

`compare_star_exp(const subexp<S>*) const`

Returns −1 (every symbol_exp is less than every star_exp).

`subexp<S>* clone()`

Clone operation. Simulates virtual copy constructor.

`void contains_symbol(const S&) const`

Returns 1 if content is equal to s, and returns 0 otherwise.

void convert_subexp(fm<S>& a) const

Converts the invoking symbol_exp into a fm and returns the result in a.

void copy(const subexp<S>&)

Copy operation. Used by clone.

subexp<S>* new_subexp()

Creation function. Simulates virtual constructor. Returns new symbol_exp.

void print(ostream& os, int i) const

Prints an alphanumeric representation of the symbol_exp on the stream os. Requires that the symbol class define operator <<.

int size() const

Returns 1.

symbol_exp()

Constructor. A no-op.

symbol_exp(const S& s)

Copy constructor. Assigns content to s. Requires that the symbol class define operator =.

symbol_exp(const symbol_exp& s)

Copy constructor. Calls clone.

~symbol_exp()

Destructor. A no-op.