GURGLE - GNU Report Generator Language

User Manual
Version 1.61 as at 21 January 2010.

Tim Colles (timc@inf.ed.ac.uk)

This manual is for Gurgle (version 1.61, updated 21 January 2010), which is a utility to produce formatted output from a variety of database inputs.

Table of Contents

0.1 Introduction

The **gurgle** program reads record and field information from a **dBase3+** file, delimited ascii text file or from an SQL query to a RDBMS and produces a report listing. Although the program was originally designed to produce *TeX/LaTeX* formatted output, plain ascii text, *troff*, *PostScript*, *HTML*, *XML*, shell scripts or any other kind of ascii based output format can be produced just as easily.

The program is ideal for generating large bodies of text where small parts of that text are substituted with information from a database. So its great for generating static web pages which have some dynamic content.

The formatting process of is controlled by a definition file which holds the report, page, and record layouts, what fields to display, and where. Other useful functions supported in the definition file include sorting, filtering, and data manipulation of records in the databases.

Below is a summary of the main features of **gurgle**.

- Support ASCII delimited text and dBase3+ databases
- Supports GNUSQL, PostgreSQL, MySQL and CA-Ingres databases
- Multiple input databases or queries
- Sorting of database records
- Automatic banner placement at the start of each sorted group
- Filters using regular expressions or user defined functions
- Five main text bodies - header, footer, record, 1st page, and Nth page
- User defined macros and text bodies
- User defined functions on field contents, including conditionals
- General purpose processing language
- User configurable input parsing patterns (default is like *awk*).
- Include files
- Environment variables
- System variables
- Multiple file output
- Optional GUILE support

0.2 Running GURGLE

The `gurgle` program takes one argument, the name of the definition file to use. This can be given with or without a `.grg` extension. If the `.grg` extension is not given with the filename and a filename without the extension is found then this will be used, else the `.grg` extension will be automatically appended and a filename must exist with this extension.

The default output file name will be the basename of the GRG file with `.tex` as the replacement extension. For example, if the GRG file to be used is `test.grg` then the program would be called as `gurgle test` (or `gurgle test.grg`) and the output file produced by the program, according to the directions given in the `test.grg` file would be `test.tex`.

The default output filename and extension can be easily changed from within the GRG file. The output filename can also be changed at certain points during generation of the output so that multiple files can be created from one input database file.

There is only one fixed option that can be given to the `gurgle` program, and this is `-d[1234]`. All variants enable debug mode. When the `-d` option is used the program will display the GRG file subsequent to preprocessing and will also dump the contents of user defined macros, equates, environment variables, as well as the definitions assigned to system variables, and the text bodies. The `-d1` variant dumps DFA state tables (from pattern matching definitions). The `-d2` variant does the same as `-d1` but also dumps the actual processing of pattern definitions. The `-d3` variant does the same as `-d` but also dumps the memory allocations during pre-preocessing and output processing. The `-d4` variant dumps details of the loading and field references of any `.dbf` file.

The GRG file is processed in two stages (see Section 0.3.1 [Structure], page 3 and Section 0.3.2 [Processing Sequence], page 3 for more details). The first stage reads all the predefined macro definitions and expands any user defined macros. The entire file is broken down into its component parts and stored internally. The output file is then generated in the second stage by writing the stored text bodies at the appropriate places. The record text body is written for each record in the master database file (subject to sort and filter definitions), and the page text bodies are written at the start of each page. The header text body is written at the start before anything else and the footer text body is written at the end after everything else. References to database fields, interpolation of user text bodies, and processing of equates are all done at this stage.

The GRG file can have as its first line a `#!` sequence followed by the path of the `gurgle` program and can then be made self-executable exactly like a shell script. Note however that only a line starting with `#` as the first line in the GRG file is interpreted this way and that `#` does not start a comment line in any other place in the file (as it would in a shell script). Command line options and arguments can be included after the path so a line like `#!/usr/local/bin/gurgle -d` is acceptable. This is one situation where it is common to not have the `.grg` extension as part of the filename.

The above allows the GRG file to act as a filter in a pipe by defining the database input file to be standard input and redefining the output file to be standard output.

The `gurgle` program will accept any number of additional command line arguments where each is a predefined macro and its arguments (these are discussed in more detail in Section 0.3.4 [Predefined Macros], page 8). These are processed after parsing the GRG file. This is so that the command line arguments can overwrite any value in the GRG file. The processing order is described in more detail in Section 0.3.2 [Processing Sequence], page 3. A predefined macro argument can include quotes and escaped quotes. These must be given differently in a command line argument to prevent shell escape. The format for a nested quote is `\"` and for a nested escaped quote is `\\\"`. For example, `gurgle fubar.grg "%%equate eq_init 1>>_eq_trace"`, is a command line which starts up tracing while processing equates in `fubar.grg`. Note that it is safest when using command line predefined macros to always include a terminating macro to avoid any possible continuation errors (when the actual file is processed). So it is better to do:

```
gurgle fubar.grg "%%equate eq_init 1>>_eq_trace" "%%end"
```

The gurgle program will also accept any number of additional command line arguments in any format which custom processing can be added to either convert them into predefined macros or to set internal values etc. See later section on this.

0.3 GURGLE File Format

The GRG file defines what the input database files are, what the formatted output is to look like, and how to process the input to get to the output. It takes the form of a list of predefined macros (one per line), most of which take arguments, and all of which are optional (except DATABASE of which there must be at least one, see Section 0.3.4.3 [DATABASE], page 10). These have three general forms. They can define an internal structure or process (such as how to sort or filter the data). They can define user macros and equations that are expanded within text bodies. They can define text bodies themselves. A text body is a block of text associated with a particular page or output position.

0.3.1 Structure

A GRG file is processed in three modes. The default is STANDARD mode where only predefined macro lines, comment lines or blank lines may be seen. Anything else will be treated as an error. The ARGS mode is used to process each of the arguments following a predefined macro name, until a newline where the mode reverts to STANDARD mode again. Some predefined macros may switch processing to the TEXT BODY mode. Within this mode user defined macros are expanded and equates are called. User defined macros are not expanded anywhere else. From TEXT BODY mode STANDARD mode is returned to when a predefined macro is encountered. The EQUATE mode is a special variant of TEXT BODY mode used only for processing equate definitions (user defined macros are also expanded in this mode).

Comments can be included in a GRG file in the STANDARD mode and the TEXT BODY mode. They are flagged by a %% sequence, followed by at least one whitespace character (space, tab or newline). This sequence and anything else up to and including a newline will be discarded. In TEXT BODY mode comments can only occur at the very start of a line (no preceeding whitespace) whereas in STANDARD mode there can be any amount of preceeding whitespace on the line. In the EQUATE mode C style comments are also supported and can occur anywhere. They are started with a /* sequence and ended with a */ sequence. Unlike C, nested comments are allowed.

0.3.2 Processing Sequence

This is the processing sequence including command line arguments, built in equates, text bodies, and the actual database file. Some of the terms may not be familiar yet but are explained further in later sections. Not all events are listed below, just the more important ones.

1. Initialise built in equates to void
2. Initialise built in system variables
3. Parse command line arguments
4. Parse .grg file
5. Parse command line arguments (second time)
6. Update environment variables and system variables

7. Execute `eq_init` equate
8. Load database files
9. Open output file
10. Set `_eq_totrec` to size of master database file
11. Sort records in master database file
12. Execute `eq_pre_header`
13. Process HEADER text body
14. Execute `eq_post_header`
15. For each record in sorted/filtered master database
    1. Set `_eq_currec`
    2. If this is page 1
        1. Execute `eq_pre_page01`
        2. Process PAGE01 text body
        3. Execute `eq_post_page01`
    3. If this is page N
        1. Execute `eq_pre_pagenn`
        2. Process PAGENN text body
        3. Execute `eq_post_pagenn`
    4. If this is a valid banner point
        1. Set `_eq_banner_val`
        2. Set `_eq_banner_nest`
        3. Execute `eq_pre_banner`
        4. Process BANNER text body
        5. Execute `eq_post_banner`
    5. Execute `eq_pre_record`
    6. Process RECORD text body
    7. Execute `eq_post_record`
16. Execute `eq_pre_footer`
17. Process FOOTER text body
18. Execute `eq_post_footer`
19. Execute `eq_exit` equate
20. Close output file

0.3.3 Environment Variables

Environment variables are internal defaults which can be altered directly from within the GRG file. Normally this is done with the DEFINE predefined macro to redefine their value, although there are a few exceptions.

0.3.3.1 NPAGE

The **NPAGE** variable holds the value `grg` uses to start a new page. Its default value is `\newpage` (for LaTeX), but it can be redefined to anything (a linefeed control character for example). The example program fragment below resets the **NPAGE** variable to the null string.

```
%%DEFINE NPAGE
```

0.3.3.2 FESCSUB

The **FESCSUB** variable controls whether the percent escaping is done for real. Normally a `\%` sequence in a text body is left as it is (since *LaTeX* needs the percent character to be escaped), however if this variable is set, then the `\%` sequence will be replaced by just a `%`. The percent character has to be escaped because it conflicts with the field name identifier. The example program fragment below sets the **FESCSUB** variable. There is no way to reset it.

```
%%DEFINE FESCSUB
```

0.3.3.3 TEXEXT

The **TEXEXT** variable holds the file extension of the output filename (that which replaces the `.grg` part). Normally a `.tex` extension is used (for LaTeX). The example program fragment below sets the value to `.html`.

```
%%DEFINE TEXEXT .html
```

0.3.3.4 DELIM

The **DELIM** variable takes an argument each character of which is the value to use as a field delimiter in a database file which is delimited ascii text. The default is space and tab for *awk* style processing. Actually sets the contents of the `<del>` token. The example program fragment below sets the value to `~`. the first character of the **DELIM** value is also used as a field separator in the default header and record text bodies.

```
%%DEFINE DELIM ~
```

0.3.3.5 DFAMODE

The **DFAMODE** variable holds the initial mode of the DFA. Normally set to `<awk><nul>` for processing delimited ascii text like input to *awk* (fields separated by space or tab, one record per line). It would only be necessary to change this when a new pattern matching style has been defined so that it can be used instead. The example program fragment below sets the value to `<usr><nul>` where this would be a user defined pattern.

```
%%DEFINE DFAMODE <usr><nul>
```

0.3.3.6 CONCAT

The **CONCAT** variable when set means that generated output is appended to the defined output file, rather than the output file being recreated on each run which is the default. Has no affect on any files generated during processing other than the main output file. The example program fragment below sets the variable, there is no way to reset it.

0.3.3.7 MKDIR

The **MKDIR** variable when set means that the directory path of the generated output file will be created if it does not already exist. This allows the construction of trees of output files. The example program fragment below sets the variable, there is no way to reset it.

```
%%DEFINE MKDIR
```

0.3.3.8 PHYSDB

The **PHYSDB** variable holds the name of the *Ingres* physical database to which an SQL query is to be directed. This variable **must** be defined whenever an SQL database file is used otherwise an error will be generated. Only one physical database can therefore be referred to from one GRG file. The example program fragment below sets the physical database name to `mydb`.

```
%%DEFINE PHYSDB mydb
```

This variable is not needed if the *GNU SQL Server* database is being used as this does not support multiple physical databases.

0.3.3.9 DBHOSTNM

The **DBHOSTNM** variable holds the name of the database server to which an SQL query is to be directed. Normally just localhost. The example program fragment below sets the database server host to `dbhost.dummy.uni.ac.uk`.

```
%%DEFINE DBHOSTNM dbhost.dummy.uni.ac.uk
```

0.3.3.10 DBUSERNM

The **DBUSERNM** variable holds the name of the user to use when an SQL query is made. Defaults to current user. The example program fragment below sets the user to `jbloggs`.

```
%%DEFINE DBUSERNM jbloggs
```

0.3.3.11 DBPASSWD

The **DBPASSWD** variable holds the password of the user to use when an SQL query is made. Not required if the user can connect without giving a password. The example program fragment below sets the password to `Joe25`.

```
%%DEFINE DBPASSWD Joe25
```

Including a plain text password within the definition file is insecure and the file should generally be protected against anyone seeing it. The CA-Ingres database will automatically prompt for a password if not given in the definition file. Alternatively custom user arguments can be added to process a password given on the command line and to set this variable.

0.3.3.12 EXPAND

The **EXPAND** variable when set means that any SQL as a database is pre-processed and any # or % reference substituted with the actual equate or field value exactly as in text bodies. This allows the results of one SQL query to be fed into the construction of another one. Normally only user defined macros (%%) are expanded in an SQL query but only when the definition file is loaded. The default is not to expand # or % due to the possible clashes with regular SQL. Note that when set the expansion applies to every defined query.

The example program fragment below sets expansion, there is no way to reset it.

```
%%DEFINE EXPAND
```

0.3.3.13 NAMCOL

The **NAMCOL** variable when set means that the field names given to columns from an SQL query will take the name given in the SQL query itself. The default behaviour is to generate numbered names for columns in the same way as for delimited ascii files. Note that internally column names are held in uppercase and have a maximum of ten characters so SQL column names from the query may be truncated and will be mapped to uppercase for the purposes of field reference within the GRG file. The example program fragment below sets **NAMCOL** so that columns are named, there is no way to reset it once the value has been changed. Note also that it applies to all queries so either all database files defined as SQL queries will have named columns or none of them will.

```
%%DEFINE NAMCOL
```

If the **NAMCOL** variable is set then the column names for any text delimited database files will be taken from the field values of the first record in the file and that record will then be discarded.

0.3.3.14 DEFCOL

The **DEFCOL** variable when set means that the field names given to columns from an SQL query will take the name given in the SQL query itself. The default behaviour is to generate numbered names for columns in the same way as for delimited ascii files. Note that internally column names are held in uppercase and have a maximum of ten characters so SQL column names from the query may be truncated and will be mapped to uppercase for the purposes of field reference within the GRG file.

If the **DEFCOL** variable is set then the column types for any text delimited database files will be taken from the field values of the first record in the file (or second if **NAMCOL** is also used) and that record will then be discarded. The example program fragment below sets **DEFCOL** so that columns are typed, there is no way to reset it once the value has been changed. Note also that it applies to all databases so either all database files defined as delimited text will have typed columns or none of them will.

```
%%DEFINE DEFCOL
```

An example input file with named and typed columns might look like the below. The first two records are absorbed and discarded after setting the column names and types from them.

```
FIRST LAST AGE
C C N
Joe Bloggs 55
```

The possible type codes are C for character data, N for numeric data, D for date data and L for logical (boolean) data.

0.3.3.15 NULL

The **NULL** variable holds a value which defines what will be used for `null` values returned from an SQL query. This is required because GURGLE does not have an explicit `null` value. The default is to just leave `null` values as an empty string. The example program fragment below defines **NULL** so that `null` values are replaced with the dash character.

```
%%DEFINE NULL -
```

0.3.3.16 PAGE1 and PAGEN

The **PAGE1** and **PAGEN** variables set the number of records per page for the first page and every other page respectively. They are normally both set to one. Unlike the other environment variables these are set with two unique predefined macros. Each macro can also be used to define the text body associated with the start of the first page or every other page. In the example program fragment below the number of records is set to three on the first page and four on every subsequent page.

```
%%PAGE01 3
%%PAGENN 4
```

The PAGE01 and PAGENN predefined macros can also take a second numeric argument. This is the number of lines per page. If the number of lines per page is execeeded then this will cause a page break irrespective of the number of records per page. The number of records per page however is not reset, and so this also will cause a page break. By setting one or other of these paramaters to 0 then the behaviour can be defined as n records per page, or n lines per page. It is illegal to set both parameters to 0. The default with no arguments specified is 0 for lines per page (not used) and infinite for records per page (there will be no automatic page breaks). Note that lines per page includes lines from the header and footer text bodies, but that records per page doesn't. This mechanism can be used to handle variable length records in plaintext based output, it is of less use for *LaTeX* output. If there is no PAGE01 definition then all pages will use the definition given for PAGENN. See Section 0.3.4.13 [PAGE01], page 15 and Section 0.3.4.14 [PAGENN], page 15 for further details.

0.3.4 Predefined Macros

Predefined macros must start at the beginning of a line (or only be preceeded by whitespace on the line). There must be nothing else on the line other than any arguments to the predefined macro. Predefined macros are not case sensitive (all upper case or all lower case or mixed case will be treated as the same macro name), for example `%%Sorton` is identical to `%%SORTON`. Predefined macros cannot be used within a text body. They can occur in any order (there are no dependencies during parsing) with the exception that user defined macros must be defined before they are used (otherwise they will not be expanded).

All predefined macros start with a `%%` sequence, followed by the rest of the macro name.

Some predefined macros have arguments. These have the following forms. A *numeric* argument is any decimal number. A *string* argument is enclosed in double quotes and can include any character except a double quote. A *string* argument cannot extend over a newline. A *field* argument is the name of a database field. As in text bodies the name must be in capitals and should be preceeded by the `%` character. A *macro* argument is a macro name followed by a space or a newline. If followed by a space then anything else up to a newline is taken to be the macros definition. A macro name must start with an alphabetic or underscore character, but can be followed by any upper or lower case alphabetic character, numeric character, or underscore character. Note that unlike predefined macros the macro name is case sensitive.

Other predefined macros start a text body on the following line. A text body can include almost any characters whatsoever. A text body is terminated by any line starting with a

%% sequence and followed by a valid predefined macro. A line starting with %% followed by whitespace will be treated as a comment and discarded from the text body. Apart from at the start of a line a %% sequence is treated as the start of a user defined macro or equate the name of which should immediately follow. In the former case the user defined macro is expanded and the definition replaces the %% and user defined macro name sequence. Similarly an equate is processed and the result expanded in place (an equate need not return a result in which case the result will be blank and the %% plus equate name sequence is removed). The # sequence can also be used to refer to an equate to process and expand (but not a user defined macro). This could be used where an equate name and a user defined macro name share the same name, since if a macro of the same name exists this will always be favoured over the equate, but by using a hash character instead it makes it explicit that an equate is being called and not a user defined macro.

The name of an equate can be immediately followed by `(...)` which contains a comma separated list of arguments to the equate, each argument can be any expression (or expressions) that results in a value. The equate name and the entire contents of the brackets is replaced by the evaluation of the equate and argument expressions. Note that the argument expressions should be given in unreversed order if the equate was defined unreversed and reversed order otherwise.

If the result of an equate that is called from within a text body is a string which is itself an equate (or includes an equate) then this will get reprocessed and so on. This is similar to the way user defined macros will continually get expanded.

A `\<newline>` sequence at the end of a line in a text body will escape the newline. This is useful if a text body is a one line equate call that may not print anything, without adding this to the end you will always get at least one blank line.

A % sequence followed by a field name in a text body will be expanded with the contents of that field from the current record of the master database file.

0.3.4.1 INCLUDE

Use this macro to include another file (or multiple files) into the current file. This macro should be followed by one or more *string* arguments. Each should be the full name of a file to include at this point. The included file can also include other files. Each include file cannot be bigger than the maximum text body size, and if multiple include files are given as arguments to this macro then the summed size of all the include files given cannot be more than the maximum text body size. The example below shows three files being included.

```
%%INCLUDE "header.grg" "footer.grg"
%%INCLUDE "record.type1"
```

0.3.4.2 DEFINE

Use this macro to define or redefine user macros. These can then be used in text bodies where they will be fully expanded. A macro definition can itself include user macros in the same way as text bodies (these will only be expanded from within the text body). However, predefined macros cannot be used in user macro definitions, nor can they be defined or redefined. This macro takes a *macro* argument which can either be just the name of the user macro being defined, or can be the name followed by the replacement text for that macro. To include a user macro in a text body or a macro definition it should be preceeded by a double percent character sequence. The example program fragment below defines three user macros, and also shows how they are used within a macro definition and a text body.

```
%%DEFINE TITLE This is the Title
%%DEFINE BOLD \bf
%%DEFINE BOLDTITLE {%%BOLD %%TITLE}
%%HEADER
\centerline{%%BOLDTITLE as at {%%BOLD \today}}
```

Note that a user macro definition can also include field names and equate macros. However, if using equate macros the name of the equate macro must be preceeded by a hash character rather than a double percent character sequence as done normally. The equate macro name need not have been defined before it is used in a user macro definition (since processing of equates occurs at a later stage.

There is no error when redefining a user macro. The new replacement text is just substituted for the old.

0.3.4.3 DATABASE

Use this macro to specify the database files, the names of the *dBase3+* files (or delimited text files) from which the records and fields are being taken. There can be one or more *string* arguments which are the full names (or pathnames) of the file to use (the `.dbf` file extension is also required in the name). If the extension is not `.dbf` then the file will be read in as a delimited ascii file using *awk* field/record style input by default, see Section 0.7 [Text Processing], page 37. The one exception to this is where the extension is `.sql` where the database file name is then just a table to query in a RDBMS, see Section 0.8 [RDBMS Queries], page 43. The example program fragment below shows all three file types in use.

```
%%DATABASE "/usr/local/lib/dbase/example.dbf"
%%DATABASE "ref1.txt" "ref2.txt"
%%DATABASE "people.sql"
```

The first database file has a path of `/usr/local/lib/dbase/`, a name of `example` and a type of `.dbf` and will be opened as a *dBase3+* file. The next two database files have no path (so must be in the current directory), names of `ref1` and `ref2` respectively and are both opened as delimited ascii files. The third database file has no path (does not need one) and is treated as an SQL query selecting the entire contents of the `people` table (`people` will also be the name of the database file) from a RDBMS.

The first database file defined is always taken as the master database file (that which text body processing, sorting and filtering acts on and through which records are cycled). Other database files can only be accessed indirectly through the pointer mechanism of equate processing (see Section 0.4.8 [Field Reference], page 21).

The special case filename which is just `"-"` can be used to read a database file from standard input (this would normally always be specified as the first database file so that it is also the master). A database file specified as `"-"` will always be loaded as a delimited ascii file.

Note that there must be at least one DATABASE command in every GRG file, in fact the smallest GRG file would just consist of one of these lines to define the database file — the default header and record text bodies would then define the format of the output file automatically.

0.3.4.4 MASTERDB

Identical to the DATABASE macro except that it sets this database to be the master database. The master database is the one through which record looping occurs and the default text bodies are output. Without using this macro the master database is always the first database loaded. This macro can be used repeatedly in place of the DATABASE macro, the master database would then be the last database loaded.

0.3.4.5 SORTON

Use this macro to define how to sort the records in the master database file. If this is not used the records will be written out from the database file in the order they are stored in the file (or retrieved by the SQL query). This macro can be followed by up to four *field* arguments (or the predefined macro can be used up to four times if only given one argument each time). The records will be sorted alphabetically on the first field given, and each sorted group (of the same value) will then be resorted on the second field given, etcetera. The BANNER macro can be used to produce a header at the start of each sorted group. The example below sorts first on the `YEAR` field, and then on the `SURNAME` field for each sorted group within this.

```
%%SORTON %YEAR %SURNAME
```

Note that sorting on boolean type data fields works as follows. A boolean field with a value of `T` or `Y` is treated as true and identical, anything else is treated as false and also identical. This supports the *dBase3*+ syntax for logical fields.

0.3.4.6 REVSORT

Just like SORTON except that it sorts the given field arguments in descending order (SORTON sorts in ascending order). The two sorts can be freely mixed. The example below sorts on the `YEAR` field in descending order, and then for each sorted year group sorts on the `SURNAME` field in ascending order.

```
%%REVSORT %YEAR
%%SORTON %SURNAME
```

0.3.4.7 FILTER

Use this macro to define how to filter the records (restrict the output) in the master database file. By default no filtering is done. There is a maximum number of filter conditions, which can either be given as multiple arguments to this macro or this macro can be used more than once. Each filter condition given as a multiple argument to one predefined macro is or'ed together. Each predefined macro call is and'ed with any others. Each filter condition is given as a *string* argument. Only records matching the given conditions will be processed. A record is written out if it matches at least one of the or'ed filter conditions from all of the and'ed filter conditions.

The filter condition has the syntax `field=re`. The `field` is the name of a database field, including the `%` character identifier. The `re` is a regular expression with the same syntax as used in the UNIX `ed(1)` and `sed(1)` commands. There should be no spaces in the condition unless they are required in the regular expression. There is an example program fragment below with some filter conditions defined.

```
%%FILTER "%STATUS=ENDED" "%STATUS=FAILED"
```

```
%%FILTER "%YEAR=199[123]"
```

The above would write out all records that have a STATUS field of *ENDED* or *FAILED* and have a YEAR field with the value *1991*, *1992*, or *1993*.

You can define a filter condition that is an equate rather than a regular expression by using the special field name %_EQ. The equate expression should return a boolean, numeric, date, or string type. The filter condition matches if the number or boolean is not 0 or the date or string is not empty. The two filters conditions given below are identical. See Section 0.3.4.8 [EQUATE], page 12 for details of equate expression syntax. Note that the equate must be a reversed equate expression, it will not currently be auto reversed if it is not and keywords will not be recognised. Nested string quotes must be escaped within a filter condition.

```
%%FILTER "%STATUS=ENDED"
%%FILTER "%_EQ=%STATUS\"ENDED\"="
```

Beware of recursive definitions of a filter condition using a (...) looping construct — since this in itself depends on the filter condition.

0.3.4.8 EQUATE

This macro works in a similar way to DEFINE except that it defines a user macro whose ultimate value is dependent on the processing of the equate expression. It provides a level of processing that occurs after preprocessing (unlike ordinary user macros). Although user defined macros are not expanded within an equate definition, equates will be but they should be prefixed by the # character (not %%).

Only simple conditional equate expressions are discussed here (that can be used to do something like an ifdef pragma). In fact equate expressions can be much more complex but the details of this are given in Section 0.4 [Equate Expressions], page 16 on equate expressions and Section 0.5 [Reversed Equate Expressions], page 28 on reversed equate expressions.

The argument given to this predefined macro is a *macro* argument as for defining user macros. However the macro definition part is slightly more complex in that it supports the ? operator. The syntax for this is ?field["string"]:["string"];. The square brackets surround optional syntax.

The semantics of the ? operator are that if the value of the given field is null (or equals 0 or 0.0 in the case of numeric fields) then do not print the field at all. So in the program fragment below

```
%%EQUATE EXAMPLE ?YEAR:;
```

the EXAMPLE equate macro has a definition saying if the value of the YEAR field is null print nothing else print the value of the YEAR field. This is a shorthand notation for the example below.

```
%%EQUATE EXAMPLE ?YEAR"%YEAR":;
```

The definition for this example says that if the value of the YEAR field is null print nothing, else evaluate the following string. The example below shows the third possible form of this operator.

```
%%EQUATE EXAMPLE ?YEAR"%YEAR":"Unknown";
```

The definition for this example says that if the value of the `YEAR` field is null evaluate the string following the colon character, else evaluate the string preceeding the colon character.

Below is a more complex example program fragment which also shows how the equate macro is used within a text body.

```
%%EQUATE NAME ?NAME1:"";?NAME2"; %NAME2":"";+
%%EQUATE SALARY \
  ?SALARY"Salary is %SALARY":"No Salary for #NAME";
%%RECORD
Details: %%NAME, %%SALARY
```

In the above example the `NAME` equate prints the contents of the `NAME1` field only if it isn't null and follows this with nothing if the contents of the `NAME2` is null else with a semicolon followed by the contents of the `NAME2` field. Note the `+` operator to concatenate the results from each condition into one (an equate must return only one value) and as a result of this each condition must always return a value so `""` is used in the `else` clause to return an empty string when the field is empty. The `SALARY` equate prints a different string dependent on whether the contents of the `SALARY` field is null or not. Note that the last string actually includes a nested call to the `NAME` equate macro. Note also the use of the backslash character to carry the definition onto the next line (the backslash can be used to escape newlines in an equate definition).

An equate macro is used in a text body in the same way as a user defined macro, that is, by using the defined name preceeded by the `%%` or the `#` character sequence. The preprocessing stage will replace a `%%` sequence with a `#` character sequence if the name does not match a user defined macro. This triggers an evaluation of the equate definition associated with the subsequent name at that point in the text body, the result replacing the `#` and the equate macro name in the same way as user macros. Note that because a hash is used to identify an equate macro any other hash in a text body should be escaped (by using `\#` which will be substituted with `#`).

The equate definition macro can be used to redefine previous macros. This will not produce any error messages. An equate macro definition can also have a null argument (as in user macros) in which case it is given a null definition.

0.3.4.9 EQGUILE

Works identically to EQUATE except that it defines an equate that is written in scheme to be run under the GUILE interpreter. The **gurgle** program must have been compiled with support for GUILE otherwise this macro will be treated the same as EQUATE.

There should be one macro argument to be the name of the equate as seen from the GURGLE side. This is so that the equate can be called from the GURGLE side as if it was any other equate. Any number of optional whitespace separated arguments may also be given to this macro (after the name). Any arguments passed to the equate from the GURGLE side will be passed to the function on the GUILE side. No error checking is done. Argument types are converted in a limited sense to the GUILE side equivalent. An equate defined in scheme is only allowed to return one argument and the GURGLE side will pick this up, convert the type and it can be used as normal. Only simple numbers and strings should be returned from the GUILE side.

Here is the definition of factorial as an equate written in scheme.

```
%%EQGUILE fact n
  (if (= n 1) 1 (* n (fact (- n 1))))
```

This will be wrapped up on the GUILE side as shown below.

```
(define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))
```

It can be called in exactly the same way as any other equate.

```
%%EQUATE FUBAR
  OUTPUTS(fact(4))
%%RECORD
Direct call: ... #fact(2) ... and then via another equate: ... #FUBAR ...
```

There are some additional procedures made available under the GUILE side from the GURGLE side, see Section 0.9 [Using GUILE], page 45 for more details.

0.3.4.10 HEADER

This macro defines a text body that is to be written at the very start of the output. The text body starts on the line following the macro and continues up to a line starting with a %% sequence (that is not a comment line) or the end of the file. A text body will have user macros fully expanded and equate macros substituted for evaluation later on. Note that user macros cannot be used at the start of a line within a text body as they will conflict with the predefined macros. Note also that because % is used as a macro identifier flag in a text body any other % should be escaped by using the \% sequence. This is not normally a problem since a % symbol has to be escaped for *LaTeX* anyway. The program fragment below is an example of this macro. The # character should also be escaped in a text body in the same way as this can flag the start of an equate.

```
%%HEADER
\documentstyle[a4]{article}
\begin{document}
```

If no HEADER is defined a default header is generated which is the name of each field in the master database file separated by the first character in the value of **DELIM**. This behaviour can be stopped by creating an empty HEADER definition or by creating a RECORD definition (empty or otherwise).

0.3.4.11 FOOTER

This macro works in an identical way to HEADER except that it defines the text body to be written right at the very end of the output. The example program fragment belows shows this macro being used.

```
%%FOOTER
\end{document}
```

0.3.4.12 BANNER

This macro defines a text body that is to be written at the top of each sorted group. Any field references within the text body will be taken from the first record in the sorted group. The macro can take one or more field arguments which defines the level at which the banners occur (for which sorted group with reference to the sorton list defined by the SORTON predefined macro). These field arguments must always correlate with the fields used for sorting. In the example program fragment below a banner is defined on the `YEAR` field and a header will be written out preceeding each sorted group of years.

```
%%SORTON %YEAR %SURNAME
%%BANNER %YEAR
\flushleft{\underline{%YEAR}}
```

Fields given as multiple arguments to one BANNER predefined macro will share the same text body, special equates can be used to distinguish which banner it is (see Section 0.4.2 [Predefined System Variables], page 18), however, fields given to separate predefined macros will each have their own unique text body. The banners can be defined in any order as they will be matched to the sorted groups according to their field arguments.

If an SQL query is being used that is ordered it it not neccessary to apply an order using SORTON to use banners. Simply define the banners with field arguments that correlate with those the SQL query is using as below.

```
%%DATABASE "people.sql"
  select * from people order by year, surname
%%BANNER %YEAR
\flushleft{\underline{%YEAR}}
```

0.3.4.13 PAGE01

This macro defines a text body that is to be written at the start of the first page of the output file. The macro optionally takes one or two *numeric* arguments which define the number of records or number of lines to be placed on the first page. The default without this argument is infinite (ie. there will be no page breaks). The program fragment below gives an example.

```
%%PAGE01 3
\vspace*{1ex}
\centerline{{\bf DEPARTMENT OF ARTIFICIAL INTELLIGENCE}}
\centerline{{\bf %%TITLE as at \today}}
```

Page breaks will only be generated if one or both of PAGE01 and PAGENN are defined.

0.3.4.14 PAGENN

This macro is similar to PAGE01 except that it defines the text body to be written at the start of every page of the output except for the first page. It also optionally takes *numeric* arguments defining the number of records or the number of lines to be put on each page excepting the first page. The default without this argument is infinite (ie. there will be no page breaks). The program fragment below is an example.

```
%%PAGENN 4
\vspace*{1ex}
```

Page breaks will only be generated if one or both of PAGE01 and PAGENN are defined.

0.3.4.15 RECORD

This macro defines the text body for each record. This is where references to fields will normally be made. References made to fields in other text bodies will always use the first or last records in the master database file. The record text body is written out for each record in the master database file (after all the sorting and filtering has been done). Below is an example program fragment which gives some idea of what a record text body field might look like.

```
%%RECORD
\begin{list}
\item[{\bf Name:}] %TITLE %SURNAME,%INITS
\item[{\bf Start Year:}] %YEAR
\item[{\bf A/C No:}] %ACCOUNT_NO
\item[{\bf Balance:}] \pounds%BALANCE
\end{list}
```

If no RECORD is defined a default record is generated which is the contents of each field in the master database file separated by the first character in the value of **DELIM**. This behaviour can be stopped by creating a RECORD definition (which could just be empty).

0.3.4.16 BLOCK

Defines a user placeable text body. This predefined macro should be followed by the name of the block (on the same line) by which it can be referred to. A user placeable text body can be included anywhere within another text body (including the text body of another block) by using the standard equate or user defined macro start character sequence. The example below shows a user text body block being defined and used.

```
%%BLOCK TITLE
This is the title
%%RECORD
#TITLE
Rest of record
```

0.3.4.17 PATTERN

Defines a pattern for parsing delimited text files and creating a database file from them. For more details see Section 0.7.4 [Redefining Patterns], page 38 on pattern definition.

0.3.4.18 END

The END predefined macro can be used to terminate text bodies and is particularly useful at the end of these in include files, since when the include file is used it guarantees that the processor is back in STANDARD mode rather than relying on their being a predefined macro straight after the include file to do this. There is no requirement to explicitly terminate a text body normally as it is always followed by another predefined macro or the end of the file both of which terminate the text body.

0.4 Equate Expressions

The equate expression language is a reasonably powerful but simple language designed solely for data manipulation within the GRG file. It has a very limited file i/o capability and no system i/o. It has conditions and loops, local variables, system (global) variables, support for record searching and matching in database files, basic maths, logical and relational operators, function call support (with arguments), and five interchangeable data types. The language is designed to supplement the text bodies, and equates can be called from text bodies (like user defined macros) to insert text that requires more complex processing to generate. They are often used to build things like summary tables from database files which are not so easy to generate directly using predefined and user defined macros.

Equates are automatically converted to a *reversed equate* which is a lower level interpreted langauge which can also be used directly if preferred (see Section 0.5 [Reversed Equate Expressions], page 28).

0.4.1 Predefined Equates

There are some equates which are pre-defined as void but whose definition can be overwritten by the user. These equates are called at specific points during generation of the output (mostly before and after text bodies) and allow additional processing to be carried out at these stages. For most of these predefined equates if a value is returned that value will be written to the output stream with the text body. This allows the equates surrounding text bodies to write values to the output without relying on a system variable and another equate within the text body. These equates are always called (the text body equates are only called if the text body itself is being written to the output stream) but have no affect unless redefined to something other than void.

- `eq_init` is called before any output is generated. Can be usefully used to enable tracing, disable the banner message, or replace the banner message with something else.
- `eq_args` is called to process each user defined command line argument.
- `eq_pre_header` is called before writing a header text body.
- `eq_pre_footer` is called before writing a footer text body.
- `eq_pre_page01` is called before writing a page 1 text body.
- `eq_pre_pagenn` is called before writing a page N text body.
- `eq_pre_banner` is called before writing a banner text body.
- `eq_pre_record` is called before writing a record text body.
- `eq_pre_block` is called before writing a block text body.
- `eq_pre_database` is called before loading a database.
- `eq_post_header` is called after writing a header text body.
- `eq_post_footer` is called after writing a footer text body.
- `eq_post_page01` is called after writing a page 1 text body.
- `eq_post_pagenn` is called after writing a page N text body.
- `eq_post_banner` is called after writing a banner text body.
- `eq_post_record` is called after writing a record text body.
- `eq_post_block` is called after writing a block text body.
- `eq_post_database` is called after loading a database.
- `eq_exit` is called at the very end of processing just before the output file is closed.

The exact call order of these is given in Section 0.3.2 [Processing Sequence], page 3 on the processing sequence. These can be redefined just like any other equate. However `eq_texinit` cannot use any database operations (since it is called before any databases have been loaded) or write to the output file (since this won't have been opened).

0.4.2 Predefined System Variables

There are some global variables predefined as a result of parsing the GRG file which can be accessed from within equates. Writing new values to some of these predefined variables may also change the processing behaviour.

- `_eq_trace` defines whether trace output of reversed equate expressions is produced. Assign `1` to this variable to turn on the output. This is normally done from the `eq_init` equate, although it can be toggled on and off at will for more selective control of tracing.
- `_eq_version` defines whether the copyright banner message is displayed (it may be useful to remove this when the program is being employed as a filter in a pipe for example). The banner can be turned off by assigning `0` to this variable. This would have to be done in the `eq_init` equate to be of any use.
- `_eq_verbose` defines whether the informative message about what file is being created is displayed. The message can be turned off by assigning `0` to this variable. This would have to be done in the `eq_init` equate to be of any use.
- `_eq_clock` contains the number of seconds since 1/1/1970 at the start of processing.
- `_eq_datenow` contains the current date at the start of processing as a *date* type (ie. *dBase3*+ format, a string of CCYYMMDD).
- `_eq_timenow` contains the current time at the start of processing as a 24hr colon separated *string* type including seconds (ie. HH:MM:SS).
- `_eq_banner_val` contains the value of the banner sort field of the current banner (the value that all the records for this banner group have in common).
- `_eq_banner_nest` contains the nesting level of the the current banner which goes from `1` to the number of defined banners and indicates which level of sort field the current banner being processed is associated with.
- `_eq_totrec` contains the total number of records (pre filtering) in the master database file.
- `_eq_currec` contains the index of the current record being processed. This value has a range from `1` to `_eq_totrec`.
- `_eq_file` contains the full name of the GRG file being processed including the path and the extension. The `_eq_ptfile` system variable which the above replaces is deprecated and should not be used.
- `_eq_base` contains the full name of the output file excluding the extension. The `_eq_texbase` system variable which the above replaces is deprecated and should not be used.
- `_eq_extn` contains the extension of the output file. The `_eq_texext` system variable which the above replaces is deprecated and should not be used.
- `_eq_outfile` contains the full name of the output file including the extension. If the special case name of `"-"` is assigned to this the output file will be set to standard output. The name of the output file is checked before the start of processing every text body. If it has changed the existing file is closed and the new file is opened, output is then written to the new file.
- `_eq_dbfpath` contains the path component of the master database file name. Can be rewritten in `eq_init` to override the default.

- `_eq_dbfname` contains the name component of the master database file name. Can be rewritten in `eq_init` to override the default.
- `_eq_dbftype` contains the type component (extension) of the master database file name. Can be rewritten in `eq_init` to override the default.
- `_eq_block` contains the name of the current user text body block being processed (this is not set for built in text body blocks such as headers and footers).
- `_eq_db_name` contains the name of the database being loaded.
- `_eq_db_name` contains the size (number of records) of the database being loaded.
- `_eq_clarg` contains the current used defined command line argument. Would be used in `eq_args`.

There are a few other more specialised system variables which are discussed in the sections to which they are relevant.

0.4.3 Keywords

The following keywords are reserved in equate expressions and should not be used to name equates or local or global variables. Keywords are only expanded in the special EQUATE variant of TEXT BODY mode. Keywords are not case sensitive and can be used in lower, upper or mixed case.

```
while, do, endwhile, roll, through, inputs,
outputs, not, and, or, xor, if, then,
else, endif, elseif, break, exit, read,
write, send, exec, expand, is, into
```

In addition `wend` and `elif` are also reserved keywords but have been deprecated and should no longer be used.

0.4.4 Defining

Equate expressions are defined using the EQUATE predefined macro. This has one argument which is the name of the equate which can be in upper or lower or mixed case and is case sensitive. The first character of the equate name should be an underscore or an alphabetic character (A–Z or a–z), the remaining characters can be these as well as the numeric characters (0–9). There is a maximum name length (see Section 0.6 [Hard Limits], page 36). The equate definition is given within a text body on the subsequent line and continues until the text body terminates. Only one equate definition can be within the text body so it can only include the definition of one function, named by the predefined macro.

An equate that takes arguments can be defined by immediately following the name with an open bracket, comma separated list of variable names (none is allowed), and then a close bracket. Alternatively the `inputs` keyword can be used as described in Section 0.4.11 [Calling Other Equates], page 24. Below are some examples of equate definitions (none of them have a body so none of them would do anything). The body of an equate is defined as a text body on the line immediately below the name/arguments definition.

```
%%EQUATE test
%%EQUATE substr(string,start_ndx,end_ndx)
%%EQUATE eval()
```

Note that the `eval` equate is commonly defined and left as void to provide a means to process equate expressions within a text body (such as a header, footer or record) where the desired equate expression is simply given as an argument in the text body and the equate itself has no affect. Note however that keywords are not recognised in this situation and therefore if operators are needed the raw reversed operator symbols must be used (see Section 0.5.5 [Reversed Equate Operators], page 31).

0.4.5 Data Types

There are six data types — based on database field types. These are strings, integers, decimals, dates, booleans, and fields. A string literal is a sequence of characters delimited by double quotes. The double quote can be included in the string by using a `\"` sequence, and the backslash can be included in a string by using a `\\` sequence. An integer literal is a sequence of digits optionally starting with the minus sign. A decimal literal is like an integer literal but can include a decimal point. Note that negative number syntax is handled at the parsing stage for literals and so there is no unary minus operator. Date and boolean literals can only be created from a database field of that type. A field literal is the name of a field rather than its contents. It is created using the `%%` field operator. The example program fragment below shows the way to create each data type.

```
"Hello World"
12345
0.25
%START_DATE
%IS_TRUE
%%SURNAME
```

Note that `START_DATE` is a date field, getting its value with the `%` operator creates a data type of date. Similarly the `IS_TRUE` field is a boolean field.

0.4.6 Variables

Variables can hold any data type. Variables can be local to the current equate or global to all equates. Once defined a variable can never be undefined (unless local where it will be undefined when the equate finishes), nor can the data type it was defined with be changed. The data type of a variable (and the variable itself) is declared when a value is written to the variable (the given name will be created as a new variable if it does not exist). A variable cannot be used until an initialising value has first been written to it to declare the variable and its type. Writing a different data type to a variable will coerce the value to the data type of the variable.

0.4.6.1 Local Variables

These variables are local to each equate. Variable names are unique to each equate. Their value can be passed to another equate only by giving them as arguments (this is call-by-value, there is no call-by-reference facility). Their value (and declaration) is lost on exiting from the equate.

The characters A–Z and a–z can be used to start the name of a local variable and the rest of the variable name can include these same characters as well as the digits 0–9 and the underscore _ character.

Care must be taken when using local variables within a text body (where they are part of the arguments passed to an equate) as these are created as global variables (but not neccessarily with the leading underscore character).

0.4.6.2 Global Variables

These are almost identical to local variables, except that the name of the variable must start with the underscore character. This distinction in naming identifies the variable as global to all equates (any equate can read or write the value at any time, although it must always be ensured that the variable has been written to prior to any read from the variable takes place). System variables are simply a special case of global variable where the variable has already been declared with a value generated by parsing the GRG file.

System variables maintain their type and will complain if a different data type from that which they were defined with is written into them (which is not the case for normal global variables).

0.4.7 Variable Assignment

A value is assigned to a variable using the assignment (`>>`) operator. This operator requires a left side which is an expression resulting in a value and a right side which is the name of a variable. If the variable does not exist it will be created and its type will be the type of the value resulting from the expression. If the variable already exists then its current value will be overwritten with the new one. If the type of the value resulting from the expression differs from the variable type the value will be coerced to the type of the variable. Some examples are shown below.

```
"hello" >> s1
"" >> s2
123 >> s2
s1 + s2 >> s3
```

The first line creates a string variable `s1` and assigns the string literal `"hello"` to it. The second line creates a string variable `s2` and assigns an empty string to it. The third line assigns the numeric literal `123` to the string variable `s2` thus converting it into a string. The fourth line concatenates the two string variables contents and assigns the result to a new string variable `s3` which has the value `"hello123"`.

0.4.8 Field Reference

The contents of fields from a database file are retrieved in the same way as from within a text body using the `%` operator which must be immediately followed by the name of the field (always in upper case). A field reference is by default made to the current database and current record of that database. All the various modes of the `%` operator are described below. Note that there is no way to change the contents of a field, database files cannot be altered with the `grg` program.

The `%` operator gets the contents of the field from the current record of the current database.

The `%%` operator gets the name of the field. This is similar to but not identical to just quoting the field name.

The `%#` operator gets the length of the field. This is the defined maximum length in characters.

The `%$` operator gets the type of the field This is a one character string which will be `C` (character/string), `N` (numeric), `D` (date), `L` (logical), or `M` (memo) although this last type is not supported as a data type.

The given field name can be more complex than a simple name to support referencing more that one database file and direct record indexing within that database file. The syntax of this is given below using the get field operator `%` as an example although any of the field operators above accept the same syntax variants.

The `%field` syntax just gets the value of the `field` from the current record and is the default syntax given above.

The `%field[index]` syntax gets the value of the `field` from the record with the given `index` (record indecies go from one to the number of records in the database).

The `%database->field` syntax gets the value of the `field` from the current record of the given `database` (name should match that given when the database file was declared using the DATABASE predefined macro without the pathname or extension).

The `%database->field[index]` syntax gets the value of the `field` from the record with the given `index` of the given `database`.

The `index` can be a simple numeric literal or any normal expression that returns a numeric value (which must be greater than or equal to `1` and less than or equal to the total number of records in the `database`). There should be no whitespace in the above syntax forms except where the `index` is a more complex expression which can include whitespace if neccessary.

The above syntax forms all bypass any active filters. The current record forms when referring to a database other than the master database will always refer to the first record of that database.

Here are some examples of field referencing from an equate.

```
%NAME >> s1
%CATS->NAME + %CATS->TYPE >> s2
1 >> a
%DOGS->NAME[2+a] >> s3
```

The first example gets the contents of the `NAME` field from the current record of the current database (the current database is always the master database unless operating under a `roll ... through` loop through another database). The second example gets the contents of the `NAME` and `TYPE` fields of the current record in the `cats` database, adds them together and assigns the result to the `s2` variable (the two fields must have the same data type). The last example gets the contents of the `NAME` field of the 3rd record in the `dogs` database and assigns the value to the `s3` variable.

0.4.9 Constructs

There are two looping constructs and a conditional construct. The basic looping construct has the form `while ... do ... endwhile`. While the expression to the left of the `do` keyword is true do the expression to the right of the `do` keyword. The expression on the left must produce a boolean value. The left and right expressions can include any complex expression, such as nested looping and conditional constructs or function calls. An example of this looping construct is given below with the definition of `strlen` (to return the length of a string).

```
%%EQUATE strlen(s)
  0 >> x
  while s'x <> 0 do
    x + 1 >> x
  endwhile
  outputs(x)
```

The other looping construct has the form `roll ... through`. The expression between the keywords is executed once for every record in the master database (filter conditions still apply). For example here is the definition of a function to return the number of records in the database.

```
%%EQUATE nrecs
  0 >> n
  roll n + 1 >> n through
  outputs(n)
```

The looping syntax can optionally include a colon suffix form. This allows looping through a named database (no filter conditions apply when this is the master database). This form is `roll ... through:database` where `database` is the name of a valid loaded database (without the pathname or extension). Fields from other databases cannot be accessed within the loop unless the `->` syntax is used, so if looping through a database other than the master database then this form must be used within the loop to access fields of the master database. Note that this applies even to nested equate calls within the loop body for example.

The conditional construct has one of the forms `if ... then ... endif` and `if ... then ... else ... endif`. If the boolean expression bewteen the `if` and `then` keywords is true execute the expression between the `then` and `else` or `endif` keywords otherwise execute the expression between the `else` and the `endif` keywords if it exists. Each expression can be as complex as required (including nested conditional and looping constructs, and function calls) but need not have any contents. Below are two examples of the conditional construct.

```
%%EQUATE test
  inputs(a,c)
  if a > 0 and a < 12 then "hello" >> b endif
  if not (c = 5) then
    5 >> c
    "no" >> b
  else
    if not (c = 3) then 3 >> c "world" >> b
    endif
  endif
  outputs(b,c)
```

Note that the `endif` keyword must always be present. The `elseif` keyword can be used for multiple conditions grouped up under one `endif` keyword (this saves having to nest conditions). The example above is shown below using the `elseif` keyword to simplify the statements.

```
%%EQUATE test
  inputs(a,c)
```

```
if a > 0 and a < 12 then "hello" >> b endif
if not (c = 5) then 5 >> c "no" >> b
elseif not (c = 3) then 3 >> c "world" >> b
endif
outputs(b,c)
```

0.4.10 Flow Control

The following two keywords may occasionlly be needed. The `break` keyword can normally be simulated in other ways but is useful in record loops. Its function is to break one level back. Normally it is used within the body of a loop to break out of the loop on certain conditions (as an addition to the standard loop condition in `while ... do ... endwhile` loops). The `exit` keyword exits the entire equate expression all the way back to where it was initially called from within a text body or in a filter condition. The example below searches all the records in the `names` database for a matching value and returns the record number of the first record the field matched.

```
%%EQUATE FIND
  1 >> n
  roll
    if %NAME = "Smith" then break
    else ++n >> n
    endif
  through:names
  outputs(n)
```

0.4.11 Calling Other Equates

Any equate can be called from within an equate. It can be passed arguments and any value returned can be assigned to a variable or used in an expression. An equate is called by simply giving its name (with optional brackets). If the equate matches the name of a local variable then the value of the variable will be returned else the equate will be called to return a value. An equate need not take any arguments nor return any value although they commonly do. Arguments are passed to an equate as a comma separated list of expressions between brackets that immediately follow the equate name. If the equate returns one argument this can be simply assigned to a variable, or used in an expression. If the equate returns multiple arguments then the `@(...)` operator sequence should be used to assign the results to multiple variables. Some examples of equate calls are shown below.

```
%%EQUATE test
  "hello" >> s
  "b" + substr(s,1,1+3) + "ws" >> s
```

Using the previously defined equate `substr()` the above would result in `"bellows"` being assigned to `s`.

Arguments to an equate are defined using the `inputs(...)` keyword sequence. Between the brackets there can be nothing or there should be a comma separated list of variable names to which the arguments passed into the equate will be assigned (the variables will be created with the data type of the argument passed in). Instead of using the `inputs` keyword it is also possible to put the bracket sequence immediately following the definition of the equate name. The return values from an equate are similarly defined using the `outputs(...)` keyword sequence. This takes a comma separated list of expressions (normally just one) the values of which are returned to the calling equate. Both the `inputs` and `outputs` keyword sequences are optional but if they are used the `inputs` must be the first line of the equate and the `outputs` must be the last line of the equate (or at least the last executed expression) - the `outputs` keyword cannot be used as a return escape. The example below shows the definition of an equate that takes two arguments and swaps the values around while also adding 10 to each value returning the two swapped arguments and then an example of that equate being called.

```
%%EQUATE swap
  inputs(a,b)
  outputs(b+10,a+10)
%%EQUATE test
  5 >> x
  7 >> y
  swap(x,y)@(x,y)
```

The `test` equate will result in `x` equaling 17 and `y` equaling 15. Note the use of the `@` operator to handle the passing back of multiple arguments. Note that `swap(x,y) >> y >> x` would be equivalent to the above but the assignments must be put in reverse order to the arguments sent back.

0.4.12 Numeric Operators

These operators are fairly self explanatory. The `+`, `-`, `/`, and `*` operators respectively add, subtract, multiply, and divide two values. The `++` and `--` unary operators respectively add and subtract one to their operand (they don't modify the operand in place like C, so `++a` is invalid, use `++a >> a` instead). They can only be used with numeric operands, with the exception of `+` which can also be used with string operands. The `+`, `++`, `-`, and `--` operators can also be used with date operands, where they add or subtract days (or add or subtract two dates).

0.4.13 Comparison Operators

These comprise the following: `>`, greater than; `<`, less than; `>=`, greater than or equals; `<=`, less than or equals; `=`, equals; `<>` not equals. These perform the respective comparison and produce a boolean data type of true or false. Can be used with numeric, string and date operands. The boolean data type is interchangeable with the numeric data type, but in general can only subsequently be used by the boolean operators or the conditional operator.

0.4.14 Boolean Operators

These perform logical operations on the boolean data type or bitwise operations on the numeric data type, but are generally interchangeable. The operator keywords `and`, `or`, and `xor` will logically AND, OR, and exclusive OR their arguments respectively. The `not` keyword operator wll logically negate its argument.

The operators `&`, `|`, and `^` bitwise and, or, and exclusive or their arguments respectively. The `~` operator performs a ones complement on its argument.

0.4.15 String Operators

The + operator and all the comparison operators also work on strings. The + operator concatenates two string arguments together, whereas the comparison operators will compare strings character by character. A string is less than another if it is earlier in ascii dictionary order and is greater than another if it is later in ascii dictionary order.

The index operators ' and ` are more complex. The ' takes two arguments, a string to index and a numeric index position. The result is the numeric value of the character in the string at the index position. The index can not go beyond the end of the string. The ` operator is the reverse. It takes three arguments, the additonal one being the numeric value to write into the string at the given index position. Below is the definition of `substr` as an example of how these operators are used.

```
%%EQUATE substr(s1,s,e)
  "" >> s2
  0 >> x
  while s <= e do
    s2,(s1's)`x >> s2
    s + 1 >> s
    x + 1 >> x
  endwhile
  outputs(s2,0`x)
```

The function takes three arguments, the string to produce a substring of, the start index and end index for the string. It pushes the resultant substring.

0.4.16 I/O Operators

The `write` keyword writes its operand on the standard output. This can be useful for debugging or displaying progress information. The `write` keyword acts as a expression terminator (like the assignment operator) and so must always appear at the end of an expression.

The `read` keyword reads from the standard input characters up to a newline, returning the string entered (including the newline character). The maximum number of characters that can be read is STRMAX. Since it returns a value it can be used anywhere in an expression.

The `send` keyword will write its operand to the output file stream (as opposed to `write` which sends its output to standard output). This is useful for writing larger blocks of text generated from equates that cannot be written into the output file in the ordinary way (by returning the value). Like the `write` keyword it acts as an expression terminator.

Here is an example of the read, write and send operators.

```
%%EQUATE feedback
  read >> s
  "Input: " + s write
  s send
```

0.4.17 Miscellaneous Operators

The `exec` keyword or `$` operator is very useful. It equates its operand. So, for example, a string containing a valid equate expression could be given as an argument to this operator which would then execute the string as an equate returning any value resulting. For example, here is a definition of `strlen` (called `gstrlen`) for any database field.

```
%%EQUATE gstrlen
  inputs(str)
  outputs(strlen(exec(str)))
```

The function might be called with `gstrlen(%%NAME)`. Note that the `%%` operator is used to put the field name on the stack. Then within the function the `$` operator is used to evaluate the argument as an equate expression, thus getting the contents of the field.

There is no dynamic array support in equate expressions currently, but a simplified mechanism can be written using the `exec` operator to create named variables with a numeric index to simulate this. The size of an array would be limited to a few hundred elements (or less, depending on how much local variable space is available on the stack). The example below defines `aget` to get the contents of an array element and `aput` to assign a value to an array element and `example` to show the equates being used. The definitions below use global variables and work only with string values.

```
%%EQUATE aget(name,index)
  "" >> index_str
  index >> index_str
  exec("_"+name+index_str)
%%EQUATE aset(name,index,value)
  "" >> index_str
  index >> index_str
  exec(value+">>_"+name+index_str)
%%EQUATE example
  /* create 5 element array, each element is set to "hello" */
  0 >> i
  while i < 5 do aset("ex",i,"hello") ++i >> i endwhile
  /* read back contents of array, send to stdout */
  0 >> i
  while i < 5 do aget("ex",i) write ++i >> i endwhile
```

The `expand` keyword or `$$` operator expands its operand as a text body. So, for example, a string containing the contents of a text body (including embedded equate and field references) can be given as an argument to this operator which will expand it (calling any embedded equates and substituting their value as necessary) and return the result as a string.

```
%%equate doblock
  "This string is #strlen(s) characters long" >> s
  outputs(expand(s))
```

0.4.18 Operator Precedence

In most cases the default precedences of the operators will be correct, however in any case where they are not (such as some mathematical and relational expressions) then bracket pairs can be used to enforce a certain ordering. Pairs of brackets can be used in other circumstances to increase legibility (such as surrounding the condition part of an `if .. then` or `while ... do` statement). Some examples are shown below.

```
%%EQUATE example
  /* default precedence below would
  be a + (b * c) + d */
  (a + b) * (c + d) >> x
  /* brackets add clarity below and also
    affect the conditon which would be
    (not a) and b by default */
  if (not (a and b)) then 1 >> c endif
  /* brackets just add clarity below */
  while ( a > 5 ) do --a >> a endwhile
```

The full operator precedence table from lowest to highest is included below.

```
or
xor
and
|
^
&
= <>
< <= > >=
+ -
* /
++ --
not ~
+ - (unary) ' ` !  \$ \$\$
```

0.4.19 Comments

You can include comments anywhere in an equate expression by enclosing them in `/*` and `*/` brackets. Nested comments are allowed. A comment may extend over a line without the newline being escaped.

0.4.20 Debugging

The best thing to do is to start up the tracer. This will dump to standard output precisely what is happening. This is however done at the level of reversed equate expressions (see Section 0.5 [Reversed Equate Expressions], page 28) as there is no symbolic debugger. You can also use the `write` keyword to print information onto standard output. You can turn on the tracer by assigning `1` to the `_eq_trace` system variable, ie. `1 >> _eq_trace`. The tracer can be turned off by assigning `0` in the same way. Control of this variable allows tracing to be turned on and off at selected points during equate processing so that the particular problem area can be focused on.

0.5 Reversed Equate Expressions

The reversed equate expression language is a postfix (reverse polish) notation (read cryptic but efficient in terms of space and speed) language. It is rather FORTH like in its appearance. The normal equate expressions are tokenised and converted into this language for processing. However an equate can be defined directly in this langauge if preferred (it is retained for compatibility since it used to be all there was). An understanding of this language can be useful for debugging purposes as this is what any trace output will currently show. However for most users there should be no need to know anything about this (except perhaps if defining equate based filter expressions or patterns which currently require reversed equates) so you can skip to the next part (Section 0.6 [Hard Limits], page 36).

All operators are single or double character combinations. Five interchangeable data types are supported. The language is stack based but in addition global variables, local variables, and system variables are available. Most operators accept an immediately following variable operand instead of one stack operand. Looping and conditional structures can also be used - as well as a function call mechanism. All in all there are a total of about 34 unique operators, some of which have multiple functions dependent on the data types of their operands.

Due to the large number of operators some have become ambiguous and it is reccommended that separators are used almost exclusively between them, the most obvious separator being whitespace. Note that some combinations need special attention, for example `~~=` which has a different meaning from `=~~`.

0.5.1 Defining Reversed Equates

Reversed equate expressions are defined using the EQUATE predefined macro. This takes two arguments separated by whitespace. The first is the name of the equate which can be in upper or lower or mixed case and is case sensitive. The first character of the equate name should be an underscore or an alphabetic character (A–Z or a–z), the remaining characters can be these as well as the numeric characters (0–9). There is a maximum name length (see Section 0.6 [Hard Limits], page 36 on hard limits). The second argument is the definition of the equate which must be on one line only. The backslash character can be used to escape the newline and allow the equate definition to be formatted over multiple lines.

0.5.2 The Stack

The stack is where everything happens. Arguments to operators are pushed onto the stack, the operator takes these values off the stack, produces the result and pushes it back on the stack. Most operators will optionally take a operand immediately following as a variable name, and use this for one of their arguments instead of the stack. Most operators are binary, although a few are unary, and there is one tertiary operator. As an example of the stack the fragment below shows four different ways to add 1 to a number (using the binary plus operator and the unary increment operator).

```
1 <<X + >>X
1 +X >>X
<<X ++ >>X
++X >>X
```

The first example pushes the numeric literal `1` on the stack, followed by the value of the `X` variable. Then the `+` operator takes both of these values off the stack, adds them and pushes the result back on the stack. Then the value is taken off the stack and written back into `X`. The second example behaves identically except that the form of the operator `+X` means that the left argument to add is read from the variable `X` and not from the stack. Note that the result is still put on the stack though. The last two examples use the increment operator instead.

0.5.3 Reversed Equate Data Types

There are five data types - based on database field types. These are strings, numbers, dates, booleans, and fields. A string literal is a sequence of characters delimited by double quotes. The double quote can be included in the string by using a `\"` sequence, and the backslash can be included in a string by using a `\\` sequence. A numeric literal is a sequence of digits optionally starting with the unary minus operator. Decimal numbers are also supported (ie. include a decimal point) and will be stored as the EQ_DEC type. Note that negative number syntax is handled at the parsing stage for literals and so there is no unary minus operator. Date and boolean literals can only be created if there is a database field of that type to use. A field literal is the name of a field rather than its contents. It is created using the `%%` operator. The example fragment below shows the way to create each data type (as an argument that gets pushed onto the stack).

```
"Hello World"
12345
0.25
%START_DATE
%IS_TRUE
%%SURNAME
```

0.5.4 Reversed Equate Variables

Variables can be used to permanently store information that would otherwise be lost on the stack. They can hold any data type. Variables can be local to the current equate or global to all equates. Once defined a variable can never be undefined (unless local where it will be undefined when the equate finishes), nor can the data type it was defined with be changed. The data type of a variable (and the variable itself) is declared when a value is written to the variable (the given name will be created as a new variable if it does not exist). A variable cannot be used until an initialising value has first been written to it to declare the variable and its type. Writing a different data type to a variable will coerce the value to the data type of the variable.

0.5.4.1 Reversed Equate Local Variables

These variables are local to the each equate. Variable names are unique to each equate. Their value can be passed to another equate only by giving them as arguments (this is call-by-value, there is no call-by-reference facility). Their value is lost on the equate returning. Local variables are held on the equate stack but build in the opposite direction to that used for general equate processing.

The characters A–Z and a–z can be used to start the name of a local variable and the rest of the variable name can include these same characters as well as the digits 0–9 and the underscore _.

Care must be taken when using local variables within a text body (where they are part of the arguments passed to an equate) as these are created as global variables (but without the underscore character).

0.5.4.2 Reversed Equate Global Variables

These are almost identical to local variables, except that the name of the variable must start with the underscore character. This distinction in naming identifies the variable as global to all equates (any equate can read or write the value at any time, although it must always be ensured that the variable is written to before any read takes place). System variables are simply a special case of global variable where the variable has already been declared with a value generated by parsing the GRG file.

System variables maintain their type and will complain if a different data type from that which they were defined with is written into them.

The system variable whose name is the underscore character alone represents the system stack pointer. So doing `0>>_` for example would clear the stack.

0.5.5 Reversed Equate Operators

This section gives a brief description, usage, and example of each operator.

0.5.5.1 Reversed Equate Variable Operators

The `>>` and `<<` operators are used to write a value to a variable and read a value from a variable respectively. Both operators must be followed by a variable name, although it can be of any type. The `>>` operator pops a value off the stack and assigns it to the designated variable - the variable will be created if neccessary. If so, then it will be created with the type of the value popped off the stack. If the variable already exists then its current value will be overwritten with the new one. However, if the type of the value popped off the stack differs from the variable type it will be cast to the same type as the variable. The `<<` operator takes the value of the designated variable and pushes it on the stack. The type of the stack argument will be the type of the variable. There is an error if the variable has not been defined. There are some examples below.

```
"hello" >>S
<<S 123 >>S <<S + .
```

The first example creates a string variable `S` and assigns the string literal `"hello"` to it. The second pushes the contents of `S` back on the stack and then assigns the numeric literal `123` to the string variable `S` thus converting it into a string. This is then pushed onto the stack and the two strings are concatenated and the result printed, which will be the string literal `"hello123"`.

It is not actually necessary to use the `<<` operator as any variable name will be looked up first as an equate call and if it is not an equate call then as a variable reference. The `<<` operator can be used where there is a name clash for example.

0.5.5.2 Reversed Equate Numeric Operators

These operators are fairly self explanatory. The `+`, `-`, `/`, and `*` operators respectively add, subtract, multiply, and divide two stack values and push the result on the stack. The `++` and `--` operators respectively add and subtract one to their operand. All these operators can take a variable designator to replace one of their stack operands. They can only be used with numeric operands, with the exception of `+` which can be used with any operand (see Section 0.5.5.5 [Reversed Equate String Operators], page 32). The `+`, `++`, `-`, and `--` operators can also be applied to dates, where they add or subtract days appropriately (or add or subtract two dates).

0.5.5.3 Reversed Equate Comparison Operators

These comprise the following: `>`, greater than; `<`, less than; `>=`, greater than or equals; `<=`, less than or equals; `=`, equals; `<>` not equals. These take two stack operands (or one stack operand and a variable designator) perform the respective comparison and push a boolean data type of true or false on the stack dependent on the result. They can be used with numeric or string operands (see Section 0.5.5.5 [Reversed Equate String Operators], page 32). The boolean data type is interchangeable with the numeric data type, but in general can only subsequently be used by the boolean operators or the conditional operator. All these operands can also be used to compare date types, taking account of the format of a date.

0.5.5.4 Reversed Equate Boolean Operators

These perform logical operations on the boolean data type or bitwise operations on the numeric data type, but are generally interchangeable. The operators `&&`, `||`, and `^^` will logically and, or, and exclusive or their arguments respectively. The `~~` operator logically negates its argument. All these operators take two boolean stack arguments (or one and a variable designator) and push a boolean result, except the negation operator which is unary. The operators `&`, `|`, and `^` bitwise and, or, and exclusive or their arguments respectively. The `~` operator performs a ones complement on its argument. All these operators take two numeric stack arguments (except the ones complement operator) or one stack argument and a variable designator and push a numeric result.

0.5.5.5 Reversed Equate String Operators

The `+` operator and all the comparison operators also work on strings. The `+` operator concatenates two string argumennts together, whereas the comparison operators will compare strings character by character. A string is less than another if it is earlier in ascii dictionary order and is greater than another if it is later in ascii dictionary order. The index operators `'` and `` ` `` are more complex. The `'` takes two arguments, a string to index (which can be on the stack or taken from a variable) and a numeric index position. The result pushed is the numeric value of the character in the string at the index position. The index cannot go beyond the end of the string. The `` ` `` operator is the reverse. It takes three arguments, the additonal one being the numeric value to write into the string at the given index position. Below is the definition of `substr` as an example of how these operators are used.

```
%%EQUATE SUBSTR      >>E>>S>>S1"">>S2\0>>X\
[<<S<<E<=;<<S'S1\<<X`S2>>S2++S>>S++X>>X]\
0<<X`S2
```

The function takes three arguments, the string to produce a substring of, the start index and end index for the string. It pushes the resultant substring.

0.5.5.6 Reversed Equate Stack Operators

The `@` operator duplicates the value at the top of the stack. The `!` operator removes the value at the top of the stack. Both operators don't care what data type they are using. The `@` operator can take a variable designator instead - in which case it pushes the variables value twice.

0.5.5.7 Reversed Equate Field Operators

All field operators must be immediately followed by the name of the field to which they are referring (always in upper case).

The `%` operator pushes the contents of the designated field name from the current record onto the stack.

The `%%` operator pushes the name of the given field on the stack. This is similar to but not identical to quoting the field name.

The `%#` operator pushes the length of the given field name on the stack. This is the defined maximum length in characters of the given field.

The `%$` operator pushes the type of the given field name on the stack. This is a one character string which is the type of the given field, this will be `C` (character/string), `N` (numeric), `D` (date), `L` (logical), or `M` (memo) although this last type is not supported.

The given field name can be more complex than a simple name to support referencing more that one database file and direct record indexing within that database file. The syntax of this is given below using the get field operator `%` as an example although any of the field operators above accept the same syntax variants.

The `%field` syntax just gets the value of the `field` from the current record and is the default syntax given above.

The `%field[index]` syntax gets the value of the `field` from the record with the given `index` (record indecies go from one to the number of records in the database).

The `%database->field` syntax gets the value of the `field` from the current record of the given `database` (name should match that given when the database file was declared using the DATABASE predefined macro without the pathname or extension).

The `%database->field[index]` syntax gets the value of the `field` from the record with the given `index` of the given `database`.

The `index` can be a simple numeric literal or any normal equate processing that returns a numeric value. There should be no whitespace in the above syntax forms except where the `index` value is a more complex equate which can include whitespace if neccessary.

The above syntax forms all bypass any active filters. The current record variants, when referring to a database other than the master database, will always refer to the first record of that database.

0.5.5.8 Reversed Equate Calling Mechanism

The `#` operator is immediately followed by the name of an equate. It results in a call to that equate. Arguments can be passed by pushing values on the stack before calling the function, and any results can be left on the stack before returning where they can subsequently be popped off. The equate need not be defined before it is called.

It is not actually necessary to use the `#` operator as any variable name will be looked up first as an equate call and if it is not an equate call then as a variable reference. The `#` operator is still required to flag an equate call from within a text body so it is retained for compatibility.

Equate calls can be given arguments as comma separated items between brackets. Each item must leave a value on the stack. This is identical to just pushing arguments on the stack before calling the function, but is clearer, and makes the separation of each argument more obvious. It also has the advantage that the brackets surround effectively any equate and the same syntax can be used in text bodies (to pass field arguments for example). This can be a useful way of doing some equate processing within the text body itself before calling the function. Note that user defined macros will be expanded within a text body even if within the equate function call. A function can be called with no arguments like `f()` if preferred. The comma separator is not neccessary, any separator can be used (or even none). For example, the call `substr(1,3,"hello")` is functionally identical to `1 3 "hello" substr`, or `substr(1 3 "hello")`, or even `1/3"hello"#substr()` but is much nicer and will work identically in a text body. Also `%%substr(1,3,%%str)` will have the user defined macro `str` expanded before the equate is called.

Since `%%` will be treated as a user defined macro or equate within a text body the field name operand cannot be used within an equate function call argument, for example, `%%substr(1,1,%%field_name)` will not work, the processor will interpret `%%field_name` as a user defined macro or equate. Since this can be a useful way to make general functions the trick is to quote the field name instead, ie. `%%substr(1,1,"%field_name")` will work. This is slightly different because the data type will end up as EQ_STR rather than EQ_FLD as in the previous equate, but this is usually not a problem.

0.5.5.9 Reversed Equate I/O Operators

The `.` (or `.<`) operator writes on the standard output the operand from the stack or variable designator as a string. This can be useful for debugging or displaying progress information.

The `.>` operator reads from the standard input characters up to a newline, returning the string entered (including the newline character). The maximum number of characters that can be read is STRMAX.

The `..` operator will send output to the output file stream (as opposed to `.` which sends its output to standard output). This is useful for writing larger blocks of text generated from equates that cannot be written into the output file in the ordinary way (by returning the value on the stack).

All of these operators can take a local or system variable argument.

0.5.5.10 Reversed Equate Miscellaneous Operators

The `$` operator is very useful. It equates the contents of the top of the stack (or variable designator). So, for example, a string could be pushed as an argument to a function, and within that function this operator could be used to equate the argument. For example, here is a definition of `strlen` for any database field.

```
%%EQUATE STRLEN $>>Y 0>>X [<<Y<<X'0=~~;++X>>X]<<X
```

The function might be called with `%%NAME#STRLEN`. Note that the `%%` operator is used to put the field name on the stack. Then within the function the `$` operator is used to evaluate the stack argument as an equate expression, thus getting the contents of the field.

The `$$` operator expands its argument as a text body.

The `\` is the null operator. It does nothing, but is useful for separating operands or for escaping a newline character.

0.5.5.11 Reversed Equate Constructs

There are two looping constructs and a conditional construct. The basic looping construct has the form `[...;...]`, either side of the semicolon may be null. While the equate expression to the left of the semicolon is true do the equate expression to the right of the semicolon. The expression on the left must leave a boolean value on the stack. An example of this looping construct was given earlier for the definition of `strlen`. The left and right expressions can include anything, such as nested looping and conditional constructs or function calls.

The other looping construct has the form `(...)`. The expression between the brackets is executed once for every record in the database. For example here is the definition of a function to return the number of records in the master database.

```
%%EQUATE NRECS 0>>NRECS(++NRECS>>NRECS)<<NRECS
```

The looping syntax can optionally include a colon suffix form. This allows looping through a named database (if this is the master database then any filtering conditions will be ignored). This form is `():database` where `database` is the name of a valid loaded database (without the pathname or extension). Fields from other databases cannot be accessed within the loop unless the `->` syntax is used, so if looping through a database other than the master database then this form must be used within the loop to access fields of the master database. Note that this applies even to nested equate calls within the loop body for example.

The conditional construct has the form `?...:...;`. This can take a field name designator, in which case the behaviour is slightly different. The action is to pop a boolean off the stack and if it is true to execute the equate expression up to the colon, and if not to execute the equate expression from the colon up to the semicolon. Each expression can be as complex as required (including nested conditional and looping constructs, and function calls) but need not have any contents. If the `?` is immediately followed by a field name and the colon and semicolon fields are null, then if the contents of the field are empty the result is null, else the result is the contents of the field. Below are two examples of the conditional construct.

```
%%EQUATE SUBDATE1 ?SUB_DATE:"In preparation";
%%EQUATE SUBDATE2 %SUB_DATE""=?"In preparation":%SUB_DATE;
```

Both the above conditions have the same action. If the `%SUBDATE` field has a value push the value on the stack, else push the default string on the stack.

0.5.5.12 Reversed Equate Flow Control

The following two operators may occasionlly be needed. The `\b` operator can be simulated in other ways but is useful to have. Its function is to break one level back. Normally it is used within the body of a loop to break out of the loop on certain conditions (as an addition to the standard loop condition in while loops). The `\e` operator exits the entire equate expression all the way back to where it was initially called from within a text body or in a filter evaluation. Neither of these operators will alter the stack. For example, the fragment below searches all the records for a matching field and returns the record number of the record the field matched.

```
%%EQUATE FIND 1>>N(%NAME"Smith"=?\b:++N>>N;)<<N
```

0.5.5.13 Reversed Equate Comments

You can include comments anywhere in an equate expression by enclosing them in `{` and `}` brackets. Nested comments are allowed. A comment may extend over a line without the newline being escaped.

This comment style is deprecated and no comments should be used within a reversed equate expression definition (comments can be placed before the definition if required using the `%%` sequence).

0.5.5.14 Reversed Equate Debugging

The best thing to do is to start up the stack and operator tracer. This will dump to standard output precisely what is happening. You can also use the `.` operator to print information onto standard output. You can turn on the tracer by writing `1` to the `_eq_trace` system variable, ie. `1>>_eq_trace`. The tracer can be turned off by writing `0` in the same way. Control of this variable allows tracing to be turned on and off at selected points during processing.

0.6 Hard Limits

The `gurgle` program internally limits the size of some data structures. These are listed here with their default maximum sizes. Exceeding the given sizes will produce a fatal error. In some cases writing the GRG file in a slightly different way will solve the problem, in others it will not. In the latter case the relevant values must be increased in the source code and the `gurgle` program recompiled. The following limits are all maximums and size is size in characters unless specified otherwise.

A number of the static buffers in GURGLE have been rewritten to scale dynamically and hence have no fixed limit anymore. These are indicated by the asterix against an entry below. You can get more information by looking at the debug output under the operating limits section.

Name	Size	Description (Dynamic?)
TEXMAXTEX	4096	text body (*)
TEXPBMAX	16384	pushback buffer (*)
MAXMACRONAME	32	length of a macro/equate/variable name
MAXMACRODEF	256	length of a macro definition (*)
MAXEQUATEDEF	1024	length of a reversed equate definition (*)
MAXMACROS	64	number of user macros (*)
MAXEQUATES	128	number of equates (*)
TEXSORTONMAX	4	sort depth
TEXBANNERMAX	4	banner sort groups (*)
TEXDBFFILEMAX	8	number of DBF files (*)
TEXFILTERMAX	8	number of filters
DBFFIELDMAX	16	length of a DBF field name (actually 10 stucturally)
STRMAX	256	length of a string argument and data type (*)
TEXBLOCKMAX	8	number of user text bodies (*)
REGEXPMAX	256	length of a regular expression

0.7 Text Processing

The `gurgle` program can parse an input file in delimited ascii format and load it internally as an ordinary database file. All the normal `gurgle` operations such as sorting, filtering, field evaluating, and equates can be used. This section gives an overview of how to use this feature.

The simplest way is to use the built in patterns. These make `gurgle` process input text in the same way as *awk*, so that each line of input is treated as a record and each whitespace separated group on a line is treated as a field of the record. Like *awk* the field delimiter can be changed by using an environment variable.

0.7.1 Declaring Text Input Files

You declare a file as text input using the predefined macro DATABASE. The name of the file must not have a `.dbf` extension. The special case filename which is just `-` can be used to indicate that the input file is to come from standard input (piped into `grg`). Note that file types can be mixed and matched as in the example below.

```
%%DATABASE "database.dbf"
%%DATABASE "mode.txt" "-"
```

In the above the `database.dbf` file is opened as normal, but the `mode.txt` file and `stdin` are read in and parsed as *awk* structured files and converted into database files. The names of the database files for reference in equates would be `database`, `mode`, and `-`. Since `-` could clash in an equate the `-` should ideally be put first in the list of databases (making it master).

0.7.2 Changing the Delimiter

You can redefine the delimiter used to separate fields in the input by using the DEFINE predefined macro and setting the environment variable **DELIM**, as in the example below.

```
%%DEFINE DELIM :
```

The above sets the delimiter to the colon character, equivalent to doing `-F:` with *awk*. You can give more than one character to **DELIM** in which case any of the given characters would count as a delimiter.

0.7.3 Referencing Fields

Once loaded, the text input file can be treated identically to a normal database file. Each field in the input record is given a numeric name so as they can be distinguished. These names have the form `Unnn` where `nnn` is a three digit number. For example, to reference the first three fields in the input you would do as in the example below.

```
%%EQUATE test "ABC"%U001=?%U002:%U003;
%%RECORD
%U001 %U002 %U003
```

The example shows fields being referenced in an equate and a text body in the same way as fields are normally referenced. Records can also be sorted on the fields or filtered as normal. The system variables `_eq_totrec` and `_eq_currec` also get set for a text input file as normal. Also direct field indexing works as normal so that the contents of any partiulcar record (input line) can be accessed.

Real column names can be setup by using the **NAMCOL** environment variable by simply defining it. Then instead of the form `Unnn` the values of the fields in the first record in the input file will be taken as the column names and that record will be ignored. These will always be used in uppercase and truncated to ten characters. Do not use non-supported field characters. The example below sets **NAMCOL** and prints fields from `dogs.txt` also shown below.

```
Contents of dogs.txt:
Name      Type      Id
Bounce    Scottie   119
Jack      Terrier   102
```

```
%%DATABASE  "dogs.txt"
%%DEFINE NAMCOL
%%RECORD
%NAME %TYPE %ID
```

Note that the **NAMCOL** directive is globally applied to all databases in the GRG file and cannot be set individually for each input file.

To establish which field is which when not using **NAMCOL** and what the column names are a simple GRG file can be created that uses the default header and record text bodies.

By default all fields loaded from a delimited text file default to character strings. The type of each column can be explicitly setup using the **DEFCOL** environment variable by simply defining it. The field values in the first record of the input file (or second record if **NAMCOL** is also used) will be taken as the field types and that record will be ignored. Field types are single upper case characters. They can be C for character data, N for numeric/decimal data, D for date data, and L for boolean data (1 or 0). The example below extends the input file above to include a record with the field types.

```
Contents of dogs.txt:
Name      Type      Id
C         C         N
Bounce    Scottie   119
Jack      Terrier   102
```

Note that the **DEFCOL** directive is globally applied to all databases in the GRG file and cannot be set individually for each input file.

0.7.4 Redefining Patterns

This is not for the faint hearted. You can skip to the next part (see Section 0.8 [RDBMS Queries], page 43).

Sometimes the input text does not match an *awk* like structure. In this case the patterns defining how to process the input can be changed. This is done using the PATTERN predefined macro. This takes five arguments. The example below shows the pattern definitions to process *awk* like input (these are built in to `gurgle`).

```
%%PATTERN "<awk><nul>" "<nul>" "" "<awk><sor>" "<sor>"
%%PATTERN "<awk><sor>" "<wht>" "" "<awk><sor>" "<nul>"
%%PATTERN "<awk><sor>" "<nul>" "" "<awk><sof>" "<sof>"
%%PATTERN "<awk><sof>" "<del>" "" "<awk><sor>" "<eof>"
%%PATTERN "<awk><sof>" "<any>" "" "<awk><sof>" "<fld>"
```

```
%%PATTERN "<awk><sof>" "<nul>" "" "<awk><sor>" "<eof>"
%%PATTERN "<awk><sor>" "<new>" "" "<awk><nul>" "<eor>"
```

The arguments are as follows. The first argument is the *mode context*. Each pattern can only apply if the current mode is the same as its *mode context*. The default mode is `<awk><nul>`, and therefore in the above only the first pattern will match. The second argument is the tokens in the text that must be present for the pattern to match, this is discussed in more detail later. The third argument is an equate expression that will be run if the pattern matched, this can then also have a say in whether the pattern matches or not. The fourth argument is the new mode to enter if the pattern matches. The fifth argument is the token to return to the parser if the pattern matches. These arguments are now discussed in more detail. Note that the first three arguments essentially define what the pattern to match is and the last two define what steps to take if the pattern does match.

0.7.4.1 Tokens and Syntax

A token is a three character sequence enclosed in angle brackets. Any characters can be used and tokens are case sensitive. With the exception of the input text pattern and the equate expressions the other fields can consist only of tokens.

The *mode context* argument can consist of tokens, the bar character `|` or the brackets `(` and `)`. The *new mode* argument can consist of tokens only. The *token* argument can consist of one token only. The *input* argument can consist of tokens, characters, and some special characters.

0.7.4.2 Mode

Only those patterns which match the current mode can match, the others are ignored. A mode is a sequence of one or more tokens, the names of the tokens or sequences can be user defined. The initial mode is `<awk><nul>` but this can be changed using the environment variable `DFAMODE`. If you are defining additional patterns it is advisable to set this to exclude the awk inbuilt patterns from matching as in the example below.

```
%%DEFINE DFAMODE <usr><nul>
%%PATTERN "<usr><nul>"
%%PATTERN "<usr><u00>|<usr><u01>"
```

Note that in the above the bar operator separates options, so the second pattern is actually applicable in two modes. Since the bar operator has a lower precedence than the concatenation operator then the two modes are `<usr><u00>` or `<usr><u01>`. Brackets can be used if neccessary to avoid ambiguities in the input. Note that instead of the bar two patterns could have been defined which were identical in the other fields but one was defined for each mode it was to match under. The final result of using a bar or multiple patterns is identical.

0.7.4.3 Input Pattern

For each pattern that matches the current mode the input pattern must match the input stream of characters for the whole pattern to match. The input pattern can consist of tokens or characters, however certain characters must be escaped and the tokens cannot be user defined. Each token generally represents a set of characters and is for convenience sake. The characters that must be escaped using a `\c` mechanism (or the special tokens) are left and right brackets, the bar character, and the star character. The first two are used to resolved ambiguities in priorities of operators, the bar is used to separate options, and the star is used to indicate zero or more occurences of a sequence. Note that the bar and star operators are not supported, although the bar operator can be simulated by defining multiple patterns, one for each option, like the modes. Some example input patterns are shown below.

```
"<dec><dec>"
"f1<new>"
```

The first pattern matches a sequence of characters that are two decimal digits, the second a sequence which is f1 followed by a newline. The complete set of predefined tokens is given below:

Token — Character Set

<nul> — NUL Special Character

<sot> — Start of Text (before first input character), not implemented

<eot> — End of Text (after last input character), not implemented

<sol> — Start of Line (before first input character of line), not implemented

<eol> — End of Line (after last input character of line), not implemented

<spc> — " "

<tab> — "\t"

<wht> — " \t"

<new> — "\n"

<car> — "\r"

<del> — " \t"

<abc> — "abcdefghijklmnopqrstuvwxyz"

<ABC> — "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

<dec> — "0123456789"

<hex> — "0123456789abcdef"

<HEX> — "0123456789ABCDEF"

<sym> — "!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"

<any> — "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ \t"

<all> — "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ \t\n\r"

<bar> — "|"

<not> — "!"

<mul> — "*"

<lbr> — "("

<rbr>	")"

The <nul> token is special because it effectively matches any character, but when it does the matching character is left on the input stream. So for example the patterns <nul>A and A both match A on the input stream. The <nul> pattern has a lower matching priority though as a more specific match is preferred over a more general one. If more than one input pattern matches the input stream, the longer match is preferred. For example if the input stream had ABC then the patterns A, AB, and ABC could all match, but the last would be chosen. If two patterns match and are the same length then the earlier one (in the order the patterns were defined) is chosen.

0.7.4.4 Equates

For each pattern that matched the given reversed equate expression is executed. If there is none then the pattern matches by default. If the equate expression leaves no value on the stack then it is assumed to be true, however if it does leave a value on the stack then the value is interpreted as a boolean type, only if it is true does the pattern match. This facility allows more complex work to be done to further control the patterns that match. Some additional system variables can be used within the equate. These are _eq_pat, which has the sequence of characters the pattern matched. The following variables, _eq_pfn, _eq_pfl, and _eq_pft can be used to set the name, length, and type of the field respectively. By default each field is named numerically, with a length equal to the length of the longest value of the field in the input and with a string type. However, for a pattern that matches a field (ie. returns the <sof>) token then these can be used to set these attributes explicitly, as in the example below.

```
"\"field1\">>_eq_pfn 32>>_eq_pfl 78>>_eq_pft"
"<<_eq_pat\"abc\"="
```

Note that for the type the ascii character value of C, N, D, or B must be used to indicate the string, numeric, date, or logical types for a field. The field will be created with the values given, so that for example if an input sequence exceeds the field length then it will be truncated. The second equate shows a boolean value being returned so that the pattern will only match if it matches the sequence abc.

0.7.4.5 New Mode

Two things happen on a matching pattern, the first of these is that a new mode is set. This field must always have a value but it could set the mode to be the same as the current mode if no change is neccessary.

0.7.4.6 Token

Any set of patterns defining an input stream must eventually result in the correct sequence of parse tokens. These represent when a field starts and ends, when a record starts and ends, and the contents of fields. This allows the parse engine to build the database file correctly. The sequence of these tokens must be returned as zero or more start of record followed by zero or more start of field, field contents, end of field tokens, followed by end of record tokens. The only valid parse tokens that can be used are given in the table below.

Token	Description
<nul>	Do nothing with the input

`<sor>` Flag start of a record

`<sof>` Flag start of a field

`<fld>` Contents of a field (can be returned multiple times)

`<eof>` Flag end of a field

`<eor>` Flag end of a record

`<err>` Generate an error

0.7.4.7 AWK Patterns

In this section the built in awk patterns are dissected one by one so as how they work can be shown.

```
%%PATTERN  "<awk><nul>" "<nul>" "" "<awk><sor>" "<sor>"
%%PATTERN  "<awk><sor>" "<wht>" "" "<awk><sor>" "<nul>"
%%PATTERN  "<awk><sor>" "<nul>" "" "<awk><sof>" "<sof>"
%%PATTERN  "<awk><sof>" "<del>" "" "<awk><sor>" "<eof>"
%%PATTERN  "<awk><sof>" "<any>" "" "<awk><sof>" "<fld>"
%%PATTERN  "<awk><sof>" "<nul>" "" "<awk><sor>" "<eof>"
%%PATTERN  "<awk><sor>" "<new>" "" "<awk><nul>" "<eor>"
```

The first which matches on the default initial mode matches any character on the input stream (but leaves the input stream as it is) jumps to the start of record mode, returning the start of record token in the process. In the start of record mode there are three possible matching patterns. They match on whitespace, the nul character and on the newline character. Whitespace is ignored (so that multiple spaces and tabs in the input are not interpreted as multiple fields), whereas the `<nul>` flags the start of a field and sets the mode to start of field mode. Since the `<nul>` is defined after the pattern for whitespace it will only match when the input is not whitespace. The newline character (which should actually also be defined above the `<nul>` pattern) sets the mode back to the initial mode and flags the end of the record. The remaining patterns only match in start of field mode. The three possible matching characters are the delimiter (normally whitespace), any other character except newline, and the nul character again. The order these are declared in is important. A character which is a delimiter character will always match first and sets the mode back to start of record mode (in prepartion for another field or end of record) and flags the end of field). Otherwise the character will match the next pattern (except if newline) and this leaves the mode the same but flags that the character is to be appended to the field definition. Finally on a newline the `<nul>` character matches and flags the end of the field and puts the mode back to start of record mode. Note that just matching a newline here (rather than null) wouldn't work as the newline signals the end of the record but would be taken off to flag the end of the field and end of record would not then be flagged. So using the `<nul>` matches the newline to return end of field but leaves the newline on the input stream so that in the start of record mode it can be matched by the last pattern definition to indicate the end of the record. This is neccessary since usually the last field does not have a delimiter after it.

0.7.4.8 Pattern Debugging

When a set of patterns is not working the parser will either produce the wrong output, loop indefinitely, or crash. You can turn on debugging with the `-d1` and `-d2` flags to see what the

DFA (Determinate Finite Automata, which isn't quite true) representation of your input patterns is. The `-d1` option shows the various arrays defining the DFA, and the `-d2` options shows a trace of the build of the main state arrays. It is beyond the scope of this document to describe the contents and structure of these arrays.

0.7.4.9 Limits

There are no limits on the dynamic database that aren't part of the limits of an ordinary database file. The DFA constructor and pattern parser although essentially dynamic do have a number of internal limits. Some of these are displayed in debug output but not all of them. Generally if they are exceeded an error will be produced.

Name	Size	Description
TEXPATTERNMAX	64	maximum number of pattern defintions
DYNRECINI	16	dynamic record initialiser/step
DYNFLDINI	16	dynamic field initialiser/step
TEXMAXUNQCHARS	32	number of unique characters/position
TEXMAXUNQMODES	32	number of unique modes
TEXMAXUNQPOSTS	16	number of unique positions
TEXTEXTPATMAX	64	input pattern maximum length (characters)
TEXMODEPATMAX	64	mode pattern maximum length (characters)
TEXMAXCHARSET	128	input pattern character set size
TEXDFAMAXSTATES	16	maximum number of unique states/postition

0.8 RDBMS Queries

The `gurgle` program can make an SQL query to *GNU SQL*, *PostgreSQL*, *MySQL* or an *CA-Ingres* RDBMS and load the results internally as a database file. Subsequent to this it can be treated exactly as if it was a database file initially, all the normal operations such as sorting, filtering, field evaluatiing, and equate processing may be used. The section gives an overview of how to use this feature. This feature will only work if support was added for it when the program was compiled.

You must generally run **gurgle** on the database server for the query to work unless you have some form of networked service in operation.

0.8.1 Declaring SQL Input Files

You declare a file as an SQL query using the predefined macro DATABASE. The name of the file must have a `.sql` extension. This is shown in the example below which loads two files as SQL queries.

```
%%DATABASE "cat.sql" "dog.sql"
```

In the above the `cat.sql` file and `dog.sql` files are treated as SQL queries (they do not need to exist as files). In the former this would be `select * from cat` and in the latter this would be `select * from dog`. The names of the database files for reference in equates would be `cat` and `dog`. In this example `cat` would be the master database file since it was loaded first.

It is possible to use any SQL `select` statement if the default is not suitable since the DATABASE predefined macro allows a text body extension to define the SQL `select` statement for this type only. The example below shows how this would be done.

```
%%DATABASE  "cat.sql"
%%DATABASE  "dog.sql"
select name, type from dog order by name
```

In the above the `cat.sql` file is interpreted as in the previous example but the `dog.sql` file has a customised SQL `select` statement defining what that database file should contain. The SQL `select` statement can be of any complexity up to the ordinary size limits of a text body, but there can be only one (unless formed from a `union`) and no other SQL statement can be used.

0.8.2 Physical Database

For any RDBMS SQL Query to work a physical database must be defined to which the SQL query is directed. Only one physical database can be defined for all the queries in the GRG file. The physical database is defined as shown in the example below.

```
%%DEFINE PHYSDB animals
```

Remember that there is no default physical database. Without defining this the loading of SQL Query database files will fail.

A physical database is only required for the *Ingres* RDBMS since this has multiple ones. If you are using the *GNU SQL Server* you do not need to use this option.

0.8.3 Referencing Columns

Once loaded, the SQL Query input file can be treated identically to a normal database file. Each field in the input record is given a numeric name so as they can be distinguished. These names have the form `Unnn` where `nnn` is a three digit number. For example, to reference the first three columns in the SQL Query you would do as in the example below.

```
%%RECORD
%U001 %U002 %U003
```

The real column names from the SQL query can be used instead by setting the **NAMCOL** environment variable by simply defining it. Then instead of the form `Unnn` the real column names will be used, converted to uppercase and truncated to 10 characters. Any non-supported field character may cause problems. Any columns that do not cleanly map can be renamed using the SQL `select x as y` syntax if neccessary (this can be often since the column naming conventions for *Ingres* are quite flexible). The example below sets **NAMCOL** and prints fields from `dog.sql` as used in an earlier example.

```
%%DEFINE PHYSDB animals
%%DEFINE NAMCOL
%%DATABASE  "dog.sql"
select name, type, id# as id from dog order by name
%%RECORD
%NAME %TYPE %ID
```

Note that like **PHYSDB** the **NAMCOL** directive is globally applied to all queries in the GRG file and cannot be set on a per-query basis. Note that setting NAMCOL will also affect the behaviour of any delimited text file databases in the same GRG file.

To establish which field is which when not using **NAMCOL** and what the column names have been mapped to a simple GRG file can be created that uses the default header and record text bodies.

0.8.4 NULL Value Handling

By default any column value in the query that is `null` will just be created in the database file as an empty string value. For some applications it may be neccessary to distinguish the `null` value from empty strings. You can redefine the value used for a `null` by using the DEFINE predefined macro and setting the environment variable **NULL**, as in the example below.

```
%%DEFINE NULL -
```

The above sets the value to use for a `null` to the dash character.

Note that like **NAMCOL** the **NULL** directive is globally applied to all queries in the GRG file and cannot be set on a per-query basis.

0.8.5 Miscellaneous

For the most part when you load a database from an SQL Query the maximum field width of 256 characters is relaxed and any length fields are supported. There may be some problems using equate processing with large text fields however.

0.9 Using GUILE

If your version of GURGLE has been compiled with support for GUILE (you will get a banner saying so when you start it up) then you can define equates in the scheme language rather than the GURGLE language. The resulting equates can be used in exactly the same way as those defined in the native language - they can be called from native language equates and from text bodies. They cannot be used where a reversed equate is required. A native language equate can pass any number of arguments to the GUILE equate (the types of which will be silently mapped) and the GUILE equate can return a numeric or string argument to the native language equate. From the GUILE side any native language equate can be directly called if it takes no arguments or only one argument and if it returns one argument this will be returned as the result. There are also some additional functions that can be called from the GUILE side defined below. This support is relatively untested.

A scheme equate is defined with the EQGUILE predefined macro. Arguments must currently be given as a white space separated list following the name of the equate (as seen from the native language side and the GUILE side). You should not include the `define` construct as this will be wrapped around automatically (to include the specified arguments). Any `define` commands used within the body of the scheme equate will only be visible from the GUILE side.

0.9.1 GURGLE Procedures

These are functions that can be called from the GUILE side code. They allow the GUILE side to read and manipulate GURGLE system variables and read database fields.

The `grg_getstrsysvar(s)` procedure takes a string which is the name of a string valued system variable and returns its current value, eg. `(display grg_getstrsysvar("_eq_extn"))` to display the value of the `_eq_extn` variable.

The `grg_getnumsysvar(s)` procedure takes a string which is the name of a numeric valued system variable and returns its current value, eg. `(display grg_getstrsysvar("_eq_verbose"))` to display the value of the `_eq_verbose` variable.

The `grg_putstrsysvar(s,v)` procedure takes a string which is the name of a string valued system variable and a string which is the value to assign to that variable. It returns the assigned value, eg. `(display grg_putstrsysvar("_eq_extn",".html"))` to change the value of the file extension and to display the change to the user.

The `grg_putnumsysvar(s,v)` procedure takes a string which is the name of a numeric valued system variable and a number which is the value to assign to that variable. It returns the assigned value, eg. `(display grg_putnumsysvar("_eq_verbose",0))` to reset the value of the verbose state.

The `grg_getfield(f,r,d)` procedure takes three arguments: the name of a field (string), a record (number) from 1 to the number of records, and the name of a database (string) to which the field is defined in. It returns the value of the given field of the given record in the given database. For example, `(display grg_getfield("NAME",2,"people"))` would display the value of the "Name" field of the 2nd record of the "people" database. This is the equivalent of the fully dereferenced field mechanism in the native language, ie. the above would be `%people->NAME[2]`.

0.10 Errors

Any error is considered fatal by the preprocessor program. This means it will abort immediately. The debug mode (see Section 0.2 [Running GURGLE], page 1) can be used to provide a lot of additional information to track down the error, if it is not immediately obvious. The following is a list of all the errors with pointers to any relevant help in alphabetical order.

grg: bad equate name (probably unescaped equate esc)
Either an undefined equate name, or the equate escape character has been used inadvertently without escaping in a text body (the `\#` character sequence should be used in this case).

grg: couldn't fill dynamic dbf file, '*filename*'
Covers a number of possible errors that stem from not being able to transfer a delimited text input file into a growing dBase3+ file structure (used internally to hold databases).

grg: couldn't open dbf file, '*filename*'
The named file declared via a GRGDBFFILE predefined macro could not be opened as a dBase3+ file (either doesn't exist or incorrect permissions).

grg: couldn't open dynamic SQL dbf file, '*filename*'
Always generated when a SQL Query declared via a DATABASE predefined macro failed for some reason (check the SQL statement is correct via another means, ensure the physical database **PHYSDB** is defined).

grg: couldn't open dynamic dbf file, '*filename*'
The named file declared via a DATABASE predefined macro could not be opened as a delimited text file (either doesn't exist or incorrect permissions).

grg: couldn't open file, '*filename*'
The given filename could not be opened for some reason. Is the path correct, filename correct, and filename readable?

grg: couldn't open include file, '*filename*'
The specified include file couldn't be opened for some reason. Is the path correct, filename correct, and filename readable?

grg: couldn't stop dynamic dbf file, '*filename*'
Covers a number of possible errors that stem from not being able to restructure a delimited text input file that has been loaded into a growing dBase3+ file structure (used internally to hold databases). Check pattern definitions.

grg: dfa.beg alloc failed
Low level out of memory error during DFA construction from pattern definitions.

grg: dfa.chr alloc failed
Low level out of memory error during DFA construction from pattern definitions.

grg: dfa.end alloc failed
Low level out of memory error during DFA construction from pattern definitions.

grg: dfa.end realloc failed
Low level out of memory error during DFA construction from pattern definitions.

grg: dfa.mid alloc failed
Low level out of memory error during DFA construction from pattern definitions.

grg: division by zero
An attempt was made to divide a number by zero in an equate expression.

grg: equate argument expected, '*string*'
An invalid argument was given to an equate macro definition. See Section 0.3.4 [Predefined Macros], page 8 on predefined macros in general and Section 0.3.4.8 [EQUATE], page 12 on equate macros in particular.

grg: equate definition too long, '*string*'
The equate definition after reversing (or if entered as reversed) is too long and must be shortened. Splitting up into more than one equate is usually the best way to do this.

grg: equate right operand expected, '*string*'
Any situation during equate processing where the operator expected a right operand (generally a variable name) immediately following the operator.

grg: equate stack overflow
When returning from evaluation of an equate back to the text body, filter definition or pattern definition from which it was called has more than one value left on the stack.

grg: equate stack underflow
When returning from evaluation of an equate back to the text body, filter definition or pattern definition from which it was called has less than zero values left on the stack. Or when an operator requires an operand from the stack but the stack is empty.

grg: equate var defined differently, '*string*'
Occurs when assigning a value to a system variable that does not match the data type the system variable was declared with. Does not occur for local variables since they just coerce the value to match their type (if possible).

grg: field argument expected, '*string*'
A field argument was expected for either the SORTON or BANNER predefined macros. See Section 0.3.4 [Predefined Macros], page 8 on field arguments and Section 0.3.4.5 [SORTON], page 11 and Section 0.3.4.12 [BANNER], page 14 on these macros in particular.

grg: field length overflow
A field value exceeded the defined length for a field. Only occurs from PostgreSQL driver

and indicates that the determined length of a field has probably not been properly derived and indicates a bug for handling the particular column type in a query result.

grg: field name too long
A field name is limited to ten characters (dBase3+ compatible).

grg: field right operand expected
A field operator has been used but the field name does not immediately follow the operator (or has not been given).

grg: filter equate - number/string/date expected
The data type returned by a filter definition based on an equate expression is not one of these.

grg: group text body too big
The definition of the text body of a group exceeded the maximum text body size.

grg: illegal ARG character
A character not allowed during the processing of the ARGS mode was seen.

grg: illegal TXT character
A character not allowed during the processing of the STANDARD mode was seen.

grg: illegal character, '*char*'
The preprocessor is very strict about characters that are not where they should be. Check that all predefined macros start at the beginning of a line and that they start with the double percent sequence. Check that comment starts at the beginning of a line with a double percent space sequence. Check that the arguments given to the predefined macros are correct (make sure that user macro definitions and equate definitions have been given a name and that string arguments start with a double character). Also check that there are not any spurious control characters in the file. See Section 0.3 [GURGLE File Format], page 3 for an overview of the structure, syntax, and semantics of GRG files.

grg: illegal character or operator in equate, '*string*'
A character that is not valid in its current position in an equate definition has been identified.

grg: index overflow
Occurs when using the stroke operator and the index given to access a character of a string was greater than the length of the string.

grg: index underflow
Occurs when using the stroke operator and the index given to access a character of a string was less than zero.

grg: inputs left bracket expected
The `inputs` keyword requires brackets to immediately follow it (although they can have no content).

grg: inputs right bracket expected
The `inputs` keyword requires brackets to immediately follow it (although they can have no content).

grg: invalid operand type
The data type of the operand did not meet the requirements of the operation being applied at the time.

grg: left token without right in pattern
An error in a pattern definition, the token is enclosed within `<...>` bracket pairs, the closing one is missing.

grg: local variable stack overflow
The number of local variables on the stack (those of the current equate all equates back up the call chain to the first one) have overrun the general equate processing stack.

grg: macro argument expected, '*string*'
Something other than a macro argument was used with either the DEFINE or the EQUATE predefined macros. See section Section 0.3.4 [Predefined Macros], page 8 on macro arguments and Section 0.3.4.2 [DEFINE], page 9 and Section 0.3.4.8 [EQUATE], page 12 on these particular macros.

grg: maximum TeX block exceeded
A text body definition has exceeded the maximum size given in Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor. There is no way to work around this, other than to reduce the size of the text body. This can be done by converting parts of the text body into user defined macros, user text bodies and equates.

grg: mismatch in comment brackets
One comment bracket (either `{ }` or /* */ pairs) opening or closing is missing.

grg: mismatch in index brackets
One indexing bracket ([ ] pairs) opening or closing is missing.

grg: mismatch in loop brackets
One looping bracket ([ ] pairs) opening or closing is missing.

grg: mismatch in loop/call brackets
One looping/call bracket (( ) pairs) opening or closing is missing.

grg: missing loop separator
The loop separator `";"` or `"do"` is missing.

grg: no database defined
Any GRG must declare at least one database file using the %%DATABASE predefined macro.

grg: no such database
Occurs when accessing a database with the `->` mechanism or the `:` mechanism on a roll/through loop and the named database has not been declared via the DATABASE predefined macro.

grg: numeric argument expected, '*string*'
One of the PAGE01 or PAGENN predefined macros expected a numeric argument. See Section 0.3.4 [Predefined Macros], page 8 on numeric arguments and sections Section 0.3.4.13 [PAGE01], page 15 and Section 0.3.4.14 [PAGENN], page 15 on these two macros.

grg: numeric operand expected
An operation required a numeric data type.

grg: numeric operands expected
An operation required two numeric data types.

grg: numeric or string operand expected
An operation required a numeric or string data type.

grg: operands types differ
An operation requires two operands of any type but they must be of the same type.

grg: out of place argument, '*string*'
This should never happen. If it does don't panic, just call for help.

grg: pattern parse error
There was a syntax error in the pattern definitions.

grg: premature end of pattern after escape sequence
There was a syntax error in the pattern definitions.

grg: pushback buffer exceeded
The internal pushback buffer has overflowed. This is used to pushback and cause this resource to break. See section Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor. Break down include files into separate files and reduce the size of user macros to work around this problem.

grg: query statement buffer overflow
The size of the query block in each SQL Section 0.3.4.3 [DATABASE], page 10 directive is statically set at a maximum of 16384 bytes. If exceeded this error will occur.

grg: record index out of range
The given index in the `x[n]` or `x->y[n]` syntax was less than 1 or greater than the total number of records in the associated database.

grg: record loop right operand expected
The record loop has been given a `":"` syntax to identify a loop through a named database but the database name has not been given or is not immediately adjacent to the colon character.

grg: reg. exp. syntax error "%<FLD>=<RE>", '*string*'
The syntax of a filter condition used with the FILTER predefined macro is wrong. The error message shows the correct syntax. See section Section 0.3.4.7 [FILTER], page 11 on the construction of filter conditions.

grg: stack overflow
The stack overflowed during equate processing as there were too many items on it (note that the stack size is reduced both by operations pushing values onto it and by local variables).

grg: stack underflow
A value was required from the stack by an operation during equate processing but the stack was empty.

grg: stack value out of range
Similar to a stack overflow except that the overflow occured during the creation of a local variable rather than an operation pusing a value onto the stack.

grg: string argument expected, '*string*'
One of the INCLUDE, DATABASE, or FILTER predefined macros expected a string argument. See Section 0.3.4 [Predefined Macros], page 8 on string arguments and Section 0.3.4.1 [INCLUDE], page 9, Section 0.3.4.3 [DATABASE], page 10, and Section 0.3.4.7 [FILTER], page 11 on these three macros.

grg: text body too big
The text body maximum size has been exceeded by the definition given for one of the predefined macros (such as a header, footer, record, equate etc).

grg: too many arguments for texpage01, '*string*'
This has a maximum of two numeric arguments.

grg: too many arguments for texpagenn, '*string*'
This has a maximum of two numeric arguments.

grg: too many arguments for texpattern, '*string*'
This has a maximum of 5 string arguments.

grg: too many banner args, '*string*'
The maximum number of field arguments to the BANNER predefined macro has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.12 [BANNER], page 14 on this particular macro.

grg: too many blocks, '*string*'
The maximum number of user defined text blocks has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.16 [BLOCK], page 16 on this particular macro.

grg: too many equate defs, '*string*'
The maximum number of equate definitions given by using the EQUATE predefined macro has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.8 [EQUATE], page 12 on this particular macro.

grg: too many equate vars
The maximum number of local variables for an equate expression has been exceeded.

grg: too many file args, '*string*'
The maximum number of file arguments given with the DATABASE predefined macro has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.3 [DATABASE], page 10 on this particular macro.

grg: too many filter args, '*string*'
The maximum number of filters specified by using the FILTER predefined macro has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.7 [FILTER], page 11 on this particular macro.

grg: too many group args, '*string*'
Not implemented.

grg: too many local vars
The maximum number of equate local variables has been exceeded. The maximum size is determined by the local variable table not the equate stack size.

grg: too many macro defs, '*string*'
The maximum number of user macro definitions given by using the DEFINE predefined macro has been exceeded. See Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.2 [DEFINE], page 9 on this particular macro.

grg: too many sort args, '*string*'
The maximum sort depth specified by using the SORTON predefined macro has been exceeded. See sections Section 0.6 [Hard Limits], page 36 on the hard limits of the preprocessor and Section 0.3.4.5 [SORTON], page 11 on this particular macro.

grg: too many unique pattern characters
The DFA limits a number of internal resources, this is one. Simplify the pattern definitions where possible.

grg: too many unique pattern modes
The DFA limits a number of internal resources, this is one. Simplify the pattern definitions where possible.

grg: undefined equate variable, '*string*'
A system variable is being used before it has been declared (by writing a value to it).

grg: undefined local variable, '*string*'
A local variable is being used before it has been declared (by writing a value to it).

grg: undefined macro or equate, '*string*'
An undefined user macro or equate definition has been used in a text body or in an equate (not user macros). See Section 0.3.4 [Predefined Macros], page 8 on macro definitions and Section 0.3.4.2 [DEFINE], page 9 and Section 0.3.4.8 [EQUATE], page 12 on user macros and equate definitions respectively.

grg: unknown banner field, '*string*'
An unknown database field name argument has been given to the BANNER predefined macro. See Section 0.3.4 [Predefined Macros], page 8 on field arguments and Section 0.3.4.12 [BANNER], page 14 on this particular macro.

grg: unknown character token, '*token*'
A pattern definition has been given a character set token which is not one of the predefined ones and has not been defined by the user.

grg: unknown dfa mode
A jump has been made to a DFA mode which has not been defined (this may be a result of not correctly setting the initial value with **DFAMODE**).

grg: unknown equate, '*string*'
An undefined equate definition has been used in a text body or in an equate. See Section 0.3.4 [Predefined Macros], page 8 on macro definitions and Section 0.3.4.8 [EQUATE], page 12 on equate definitions.

grg: unknown field name, '*string*'
An unknown database field name has been used in a text body or in an equate. See Section 0.3.4 [Predefined Macros], page 8 on text bodies and Section 0.3.4.8 [EQUATE], page 12 on equate definitions.

grg: unknown field type, '*string*'
A field type defined in a dBase3+ database file has not been recognised as The valid recognised types are C, N, D and L.

grg: unknown mode
This should never happen. If it does don't panic, just call for help.

grg: unknown predefined GURGLE macro, '*string*'
A predefined macro name has not been recognised. All predefined macros start with the GRG or TEX sequence, followed by the rest of the name which must match one of the valid names. Same error as below.

grg: unknown predefined tex macro, '*string*'
An invalid predefined macro has been used. Check for spelling mistakes. See Section 0.3.4 [Predefined Macros], page 8 for a list of predefined macros. Note that predefined macros are not case sensitive.

grg: unknown sort field, '*string*'
An unknown database field name argument has been given to the SORTON predefined macro. See Section 0.3.4 [Predefined Macros], page 8 on field arguments and Section 0.3.4.5 [SORTON], page 11 on this particular macro.

grg: unknown token
An error in a pattern definition, the resulting token should be one of the valid start of record, end of record, start of field, end of field, error or discard token values.

grg: unseparated condition
A condition construct has been used in an equate expression which is missing the "else" clause (or the "endif" clause if no "else" clause is given). Note that reversed expressions always require an "else" clause (the colon character) whereas unreversed expressions do not.

grg: unterminated condition
A condition construct has been used in an equate expression which is missing the "endif" or ";" terminator. All conditions require this terminator.

grg: unterminated string, '*string*'
A string has been given with a missing end quote.

0.11 Examples

Included below is a complete example GRG file of a reasonable complexity. This demonstrates how moderately complex output can be generated without doing any very complex equate work (ie. just by using basic conditions on fields).

```
%% DAI Grants Awarded Report Form Template
%%DEFINE    TITLE Grants Awarded
%%DATABASE  "../research.dbf"
%%FILTER    "%GRANT_STAT=ACTIVE" "%GRANT_STAT=AWARDED"
%%SORTON    %AI_AREA %PI_SNAME
%%EQUATE    GRANT_HOLDER ?CO_INVEST"; %CO_INVEST"
%%EQUATE    DURATION ?START_DATE"%START_DATE--"%END_DATE
%%EQUATE    ACCOUNT_NO ?ACCOUNT_NO"RR%ACCOUNT_NO"
%%EQUATE    RAS_PER_YR ?RAS_PER_YR
%%EQUATE    STAFF_AD ?STAFF_AD"%STAFF_AD "?AD_PERCENT"%AD_PERCENT\%; "
%%EQUATE    STAFF_TG ?STAFF_TG"%STAFF_TG "?TG_PERCENT"%TG_PERCENT\%; "
%%EQUATE    STAFF_CN ?STAFF_CN"%STAFF_CN "?CN_PERCENT"%CN_PERCENT\%"
%%EQUATE    STAFF_AR ?STAFF_AR"%STAFF_AR "?AR_PERCENT"%AR_PERCENT\%"
%%EQUATE    STAFF1 ?STAFF_NM1
%%EQUATE    STAFF2 ?STAFF_NM2"; %STAFF_NM2"
%%EQUATE    STAFF3 ?STAFF_NM3"; %STAFF_NM3"
%%EQUATE    STAFF4 ?STAFF_NM4"; %STAFF_NM4"
%%EQUATE    STAFF5 ?STAFF_NM5"; %STAFF_NM5"
%%HEADER
  \documentstyle[grants]{article}
  \pagestyle{headings}
  \markright{\noindent Department of Artificial Intelligence
  \hfill %%TITLE as at \today{} \hfill}
  \sloppy
  \begin{document}
  \begin{titlepage}
  \mbox{}
  \end{titlepage}
```

```
%%PAGE01 3
  \vspace*{1ex}
  \centerline{{\bf DEPARTMENT OF ARTIFICIAL INTELLIGENCE}}
  \centerline{{\bf %%TITLE as at \today}}
  \vspace*{1ex}
%%PAGENN 4
  \vspace*{1ex}
%%BANNER    %AI_AREA
  \vspace*{1ex}
  \flushleft{\underline{\bf %AI_AREA}}
%%RECORD
  \begin{list}{}{\leftmargin 2.15in
  \renewcommand{\makelabel}[1]{\hfil \#1}\labelwidth 2.1in
  \itemsep 0ex \parsep 0ex}
  \item[{\bf Title:}] %GRANT_TITL
  \item[{\bf Grantholder:}] %PI_SNAME, %PI_INITS%%GRANT_HOLDER
  \item[{\bf Grant No:}] %GRANT_NO
  \item[{\bf A/C No:}] %%ACCOUNT_NO
  \item[{\bf Duration:}] %%DURATION
  \item[{\bf Value:}] \pounds%TOTAL\\{\bf Staff} \pounds%STAFF_COST;
  {\bf Travel} \pounds%TRAVEL; {\bf Consumables} \pounds%CONSUMABLE;
  \\{\bf Equipment} \pounds%EQUIPMENT; {\bf Indirect} \pounds%INDIRECT
  \item[{\bf RA Man Yrs/Ac Yr:}] %%RAS_PER_YR
  \item[{\bf Research Posts:}] %%STAFF_AR
  \item[{\bf Other Posts:}] %%STAFF_AD%%STAFF_TG%%STAFF_CN
  \item[{\bf Staff:}] %%STAFF1%%STAFF2%%STAFF3%%STAFF4%%STAFF5
  \item[{\bf Source Of Funding:}] %FUNDING
  \end{list}
%%FOOTER
  \end{document}
```

The example below is one of the simplest files that could be created. This makes an SQL query to a RDBMS and just dumps the output in delimited form using the default header and record text bodies.

```
%%DATABASE "cat.sql"
%%DEFINE PHYSDB animals
%%DEFINE NAMCOL
%%DEFINE DELIM |
```

The example below shows a more complex SQL query which is then used to construct HTML for a web page in a fairly simple way. The file includes some other files for default setup which are not shown below.

```
#!/usr/bin/gurgle
%%INCLUDE   "../PTC/html.pt"
%%INCLUDE   "../PTC/basic.grg"
%%INCLUDE   "banner.html"
%%EQUATE    eq_init "../testing/absences.html" >>_eq_outfile
%%DEFINE    PHYSDB daidb
```

```
%%DEFINE    NAMCOL

%%DATABASE "absence.sql"
  select
    lastname, firstname, reason, reason_oth, start_date as sdate,
    return_date as rdate,
    date_part('year',start_date) as year,
    date_trunc('month',start_date) as month,
    return_date - start_date as days
  from
    absence, person
  where
    person.person# = absence.person# and
    ( ( start_date >= date('today') and
        start_date < date('today') + '28 days' ) or
      ( start_date < date('today') and
        return_date >= date('today') ) )
  order by
    lastname

%%HEADER
  #BEGIN_INSERT
  <TABLE BORDER=0>
  <TR ALIGN=LEFT>
    <TH COLSPAN=2>Name
    <TH COLSPAN=2>Date(s)
    <TH COLSPAN=2>Reason

%%RECORD
  <TR>
    <TD><B>%FIRSTNAME %LASTNAME</B><TD>
    <TD>%SDATE #EVAL(%RDATE<>""?" to "+%RDATE:;) (%DAYS)<TD>
    <TD>#EVAL(%REASON="Other"?%REASON_OTH:%REASON;)

%%FOOTER
  </TABLE>
  #END_INSERT

%%END
```

The example below just contains some useful equate definitions for commonly required functions (particularly string handling).

```
%%EQUATE traceon
  /* turn on equate tracing */
  1 >> _eq_trace
%%EQUATE traceoff
  /* turn off equate tracing */
```

```
  0 >> _eq_trace
%%EQUATE datey
  /* returns string with century and year part of date */
  inputs(s)
  outputs(substr(s,0,3))
%%EQUATE datem
  /* returns string with month part of date */
  inputs(s)
  outputs(substr(s,4,5))
%%EQUATE dated
  /* returns string with day part of date */
  inputs(s)
  outputs(substr(s,6,7))
%%EQUATE bdate
  /* returns string with date formatted to "dd/mm/ccyy" */
  inputs(s)
  outputs(dated(s)+"/"+datem(s)+"/"+datey(s))
%%EQUATE ltou
  /* converts lower case string s to upper case */
  inputs(s)
  if not (s = "") then
    0 >> i
    while s'i <> 0 do
      if s'i >= 97 and s'i <= 122 then
        s,(s'i - 32)`i >> s
      endif
      ++i >> i
    endwhile
  endif
  outputs(s)
%%EQUATE substr
  /* returns substring of s1 starting at s and ending at e */
  inputs(s1,s,e)
  "" >> s2
  0 >> x
  while (s <= e) do
    s2,(s1's)`x >> s2
    ++s >> s
    ++x >> x
  endwhile
  outputs(s2,0`x)
%%EQUATE strlen
  /* returns length of string s */
  inputs(s)
  0 >> x while s'x <> 0 do ++x >> x endwhile
  outputs(x)
%%EQUATE nrecs
```

```
/* returns size (n records) of given database db */
/* note that for a quoted string keywords are not expanded within so
the raw reversed operators must be used instead */
inputs(db)
0 >> n
if db = "" then "" >> s else ":" + db >> s endif
"(n + 1 >> n)" + s >> s
exec(s)
outputs(n)
```

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

   F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

   G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

   H. Include an unaltered copy of this License.

   I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

   J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

   K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Command Index

Variable Index

_

C

D

E

F

M

N

P

T

Concept Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W