GNU recutils

for version 1.9, 19 September 2024

by Jose E. Marchesi and John Darrington

Table of Contents

1 Introduction

1.1 Purpose

GNU recutils is a set of tools and libraries to access human-editable, text-based databases called *recfiles*. The data is stored as a sequence of records, each record containing an arbitrary number of named fields. Advanced capabilities usually found in other data storage systems are supported: data types, data integrity (keys, mandatory fields, *etc.*) as well as the ability of records to refer to other records (sort of foreign keys). Despite its simplicity, recfiles can be used to store medium-sized databases.

So, yet another data storage system? The mere existence of this package deserves an explanation. There is a rich set of already available free data storage systems, covering a broad range of requirements. Big systems having complex data storage requirements will probably make use of some full-fledged relational system such as MySQL or PostgreSQL. Less demanding applications, or applications with special deployment requirements, may find it more convenient to use a simpler system such as SQLite, where the data is stored in a single binary file. XML files are often used to store configuration settings for programs, and to encode data for transmission through networks.

So it looks like all the needs are covered by the existing solutions ... but consider the following characteristics of the data storage systems mentioned in the previous paragraph:

– The stored data is not directly human readable.
– The stored data is definitely not directly writable by humans.
– They are program dependent.
– They are not easily managed by version control systems.

Regarding the first point (human readability), while it is clearly true for the binary files, some may argue XML files are indeed human readable... well... `<bar><foo tag="val">try</foo> to r&iamp;ead <p>this</p></bar>`. YAML[1] is an example of a hierarchical data storage format which is much more readable than XML. The problem with YAML is that it was designed as a “data serialization language” and thus to map the data constructs usually found in programming languages. That makes it too complex for the simple task of storing plain lists of items.

Recfiles are human-readable, human-writable and still easy to parse and to manipulate automatically. Obviously they are not suitable for any task (for example, it can be difficult to manage hierarchies in recfiles) and performance is somewhat sacrificed in favor of readability. But they are quite handy to store small to medium simple databases.

The GNU recutils suite comprises:

– This Texinfo manual, describing the Rec format and the accompanying software.
– A C library (librec) that provides a rich set of functions to manipulate rec data.
– A set utilities that can be used in shell scripts and in the command line to operate on rec files.
– An emacs mode, `rec-mode`.

1.2 A Little Example

Everyone loves to grow a nice book collection at home. Unfortunately, in most cases the management of our private books gets uncontrolled: some books get lost, some of them may be

[1] Yet Another Markup Language

loaned to some friend, there are some duplicated (or even triplicated!) titles because we forgot about the existence of the previous copy, and many more details.

In order to improve the management of our little book collection we could make use of a complex data storage system such as a relational database. The problem with that approach, as explained in the previous section, is that the tool is too complicated for the simple task: we do not need the full power of a relational database system to maintain a simple collection of books.

With GNU recutils it is possible to maintain such a little database in a text file. Let's call it '`books.rec`'. The following table resumes the information items that we want to store for each title, along with some common-sense restrictions.

– Every book has a title, even if it is "No Title".
– A book can have several titles.
– A book can have more than one author.
– For some books the author is not known.
– Sometimes we don't care about who the author of a book is.
– We usually store our books at home.
– Sometimes we loan books to friends.
– On occasions we lose track of the physical location of a book. Did we loan it to anyone? Was it lost in the last move? Is it in some hidden place at home?

The contents of the rec file follows:

```
# -*- mode: rec -*-

%rec: Book
%mandatory: Title
%type: Location enum loaned home unknown
%doc:
+ A book in my personal collection.

Title: GNU Emacs Manual
Author: Richard M. Stallman
Publisher: FSF
Location: home

Title: The Colour of Magic
Author: Terry Pratchett
Location: loaned

Title: Mio Cid
Author: Anonymous
Location: home

Title: chapters.gnu.org administration guide
Author: Nacho Gonzalez
Author: Jose E. Marchesi
Location: unknown

Title: Yeelong User Manual
Location: home

# End of books.rec
```

Simple. The file contains a set of records separated by blank lines. Each record comprises a set of fields with a name and a value.

The GNU recutils can then be used to access the contents of the file. For example, we could get a list of the names of loaned books by invoking `recsel` in the following way:

```
$ recsel -e "Location = 'loaned'" -P Title books.rec
The Colour of Magic
```

2 The Rec Format

A recfile is nothing but a text file which conforms to a few simple rules. This chapter shows you how, by observing these rules, recfiles of arbitrary complexity can be written.

2.1 Fields

A *field* is the written form of an association between a label and a value. For example, if we wanted to associate the label `Name` with the value `Ada Lovelace` we would write:

```
Name: Ada Lovelace
```

The separator between the field name and the field value is a colon followed by a blank character (space and tabs, but not newlines). The name of the field shall begin in the first column of the line.

A *field name* is a sequence of alphanumeric characters plus underscores (`_`), starting with a letter or the character `%`. The regular expression denoting a field name is:

```
[a-zA-Z%][a-zA-Z0-9_]*
```

Field names are case-sensitive. `Foo` and `foo` are different field names.

The following list contains valid field names (the final colon is not part of the names):

```
Foo:
foo:
A23:
ab1:
A_Field:
```

The *value of a field* is a sequence of characters terminated by a single newline character (`\n`).

Sometimes a value is too long to fit in the usual width of terminals and screens. In that case, depending on the specific tool used to access the file, the readability of the data would not be that good. It is therefore possible to physically split a logical line by escaping a newline with a backslash character, as in:

```
LongLine: This is a quite long value \
comprising a single unique logical line \
split in several physical lines.
```

The sequence `\n` (newline) `+` (PLUS) and an optional `_` (SPACE) is interpreted as a newline when found in a field value. For example, the C string `"bar1\nbar2\n bar3"` would be encoded in the following way in a field value:

```
Foo: bar1
+ bar2
+  bar3
```

2.2 Records

A *record* is a group of one or more fields written one after the other:

```
Name1: Value1
Name2: Value2
Name2: Value3
```

It is possible for several fields in a record to share the same name or/and the field value. The following is a valid record containing three fields:

```
Name: John Smith
Email: john.smith@foomail.com
Email: john@smith.name
```

The *size of a record* is defined as the number of fields that it contains. A record cannot be empty, so the minimum size for a record is 1. The maximum number of fields for a record is only limited by the available physical resources. The size of the previous record is 3.

Records are separated by one or more blank lines. For instance, the following example shows a file named '`personalities.rec`' featuring three records:

```
Name: Ada Lovelace
Age: 36

Name: Peter the Great
Age: 53

Name: Matusalem
Age: 969
```

2.3 Comments

Any line having an `#` (ASCII 0x23) character in the first column is a comment line.

Comments may be used to insert information that is not part of the database but useful in other ways. They are completely ignored by processing tools and can only be seen by looking at the recfile itself.

It is also quite convenient to comment-out information from the recfile without having to remove it in a definitive way: you may want to recover the data into the database later! Comment lines can be used to comment-out both full registers and single fields:

```
Name: Jose E. Marchesi
# Occupation: Software Engineer
# Severe lack of brain capacity
# Fired on 02/01/2009 (without compensation)
Occupation: Unoccupied
```

Comments are also useful for headers, footers, comment blocks and all kind of markers:

```
# -*- mode: rec -*-
#
# TODO
#
# This file contains the Bugs database of GNU recutils.
#
# Blah blah...

...

# End of TODO
```

Unlike some file formats, comments in recfiles must be complete lines. You cannot start a comment in the middle of a line. For example, in the following record, the `#` does *not* start a comment:

```
Name: Peter the Great # Russian Tsar
Age: 53
```

2.4 Record Descriptors

Certain properties of a set of records can be specified by preceding them with a *record descriptor*. A record descriptor is itself a record, and uses fields with some predefined names to store properties.

2.4.1 Record Sets

The most basic property that can be specified for a set of records is their *type*. The special field name `%rec` is used for that purpose:

```
%rec: Entry

Id: 1
Name: Entry 1

Id: 2
Name: Entry 2
```

The records following the descriptors are then identified as having its type. So in the example above we would say there are two records of type “Entry”. Or in a more colloquial way we would say there are two “Entries” in the database.

The effect of a record descriptor ends when another descriptor is found in the stream of records. This allows you to store different kinds of records in the same database. For example, suppose you are maintaining a depot. You will need to keep track of both what items are available and when they are sold or restocked.

The following example shows the usage of two record descriptors to store both kind of records: articles and stock.

```
%rec: Article

Id: 1
Title: Article 1

Id: 2
Title: Article 2

%rec: Stock

Id: 1
Type: sell
Date: 20 April 2011

Id: 2
Type: stock
Date: 21 April 2011
```

The collection of records having same types in recfiles are known as *record sets* in recutils jargon. In the example above two record sets are defined: one containing articles and the other containing stock movements.

Nothing prevents having empty record sets in databases. This is in fact usually the case when a new recfile is written but no data exists yet. In our depot example we could write a first version of the database containing just the record descriptors:

```
%rec: Article

%rec: Stock
```

Special records are not required, and many recfiles do not have them. This is because all the records contained in the file are of the same type, and their nature can usually be inferred from both the file name and their contents. For example, ‘`contacts.rec`’ could simply contain records representing contacts without an explicit `%rec: Contact` record descriptor. In this case we say that the type of the anonymous records stored in the file is the *default record type*.

Another possible situation, although not usual, is to have a recfile containing both non-typed (default) and typed record types:

```
Id: 1
Title: Blah

Id: 2
Title: Bleh

%rec: Movement

Date: 13-Aug-2012
Concept: 20

Date: 24-Sept-2012
Concept: 12
```

In this case the records preceding the movements are of the “default” type, whereas the records following the record descriptor are of type `Movement`. Even though it is supported by the format and the utilities, it is generally not recommended to mix non-typed and typed records in a recfile.

2.4.2 Naming Record Types

It is up to you how to name your record sets. Any string comprising only alphanumeric characters or underscores, and that starts with a letter will be a legal name. However, it is recommended to use the singular form of a noun in order to describe the “type” of the records in the records set. Examples are `Article`, `Contributor`, `Employee` and `Movement`.

The used noun should be specific enough in order to characterize the property of the records which matters. For example, in a contributor’s database it would be better to have a record set named `Contributor` than `Person`.

The reason of using singular nouns instead of their plural forms is that it works better with the utilities: it is more natural to read `recsel -t Contributor` (`-t` is for “type”) than `recsel -t Contributors`.

2.4.3 Documenting Records

As well as a name, it is a good idea to provide a description of the record set. This is sometimes called the record set’s *documentation* and is specified using the `%doc` field.

Whereas the name is usually short and can contain only alphanumeric characters and underscores, no such restriction applies to the documentation. The documentation is typically more verbose than the name provided by the `%rec` field and may contain arbitrary characters such as punctuation and parentheses. It is somewhat similar to a comment (see Section 2.3 [Comments], page 5), but it can be managed more easily in a programmatic way. Unlike a comment, the `%doc` field is recognized by tools such as `recinf` (see Section 17.1 [Invoking recinf], page 62) which processes record descriptors. For example, you might have two record sets with `%rec` and `%doc` fields as follows:

```
%rec: Contact
%doc: Family, friends and acquaintances (other than business).

Name: Granny
Phone: +12 23456677

Name: Edwina
```

```
    Phone: +55 0923 8765


    %rec: Associate
    %doc: Colleagues and other business contacts

    Name: Karl Schmidt
    Phone: +49 88234566

    Name: Genevieve Curie
    Phone: +33 34 87 65
```

2.4.4 Record Sets Properties

Besides determining the type of record that follows in the stream, record descriptors can be used to describe other properties of those records. This can be done by using *special fields*, which have special names from a predefined set. Consider for example the following database, where record descriptors are used to specify a (optional) numeric 'Id' and a mandatory 'Title' field:

```
    %rec: Item
    %type: Id int
    %mandatory: Title

    Id: 10
    Title: Notebook (big)

    Id: 11
    Title: Fountain Pen
```

Note that the names of special fields always start with the character `%`. Also note that it is also possible to use non-special fields in a record descriptor, but such fields will have no effect on the described record set.

Every record set must contain one, and only one, field named `%rec`. It is not mandated that that field must occupy the first position in the record. However, it is considered a good style to place it as the first field in the record set, in order for the casual reader to easily identify the type of the records.

The following list briefly describes the special fields defined in the recutils format, along with references to the sections of this manual describing their usage in depth.

`%rec`
: Naming record types. Also, they allow using external and remote descriptors. See Chapter 9 [Remote Descriptors], page 39.

`%mandatory`, `%allowed` and `%prohibit`
: Requiring or forbidding specific fields. See Section 7.1 [Mandatory Fields], page 34. See Section 7.2 [Prohibited Fields], page 34. See Section 7.3 [Allowed Fields], page 35.

`%unique` and `%key`
: Working with keys. See Section 7.4 [Keys and Unique Fields], page 35.

`%doc`
: Documenting your database. See Section 2.4.3 [Documenting Records], page 7.

`%typedef` and `%type`
: Field types. See Chapter 6 [Field Types], page 29.

`%auto`
: Auto-counters and time-stamps. See Chapter 12 [Auto-Generated Fields], page 49.

`%sort`
: Keeping your record sets sorted. See Section 3.7 [Sorted Output], page 20.

`%size` Restricting the size of your database. See Section 7.6 [Size Constraints], page 36.

`%constraint`
Enforcing arbitrary constraints. See Section 7.7 [Arbitrary Constraints], page 37.

`%confidential`
Storing confidential information. See Chapter 13 [Encryption], page 52.

`%singular`
Fields without repeating values.

3 Querying Recfiles

Since recfiles are always human readable, you could lookup data simply by opening an editor and searching for the desired information. Or you could use a standard tool such as `grep` to extract strings matching a pattern. However, recutils provides a more powerful and flexible way to lookup data. The following sections explore how the recutils can be used in order to extract data from recfiles, from very basic and simple queries to quite complex examples.

3.1 Simple Selections

`recsel` is an utility whose primary purpose is to select records from a recfile and print them on standard output. Consider the following example record set, which we shall assume is saved in a recfile called '`acquaintances.rec`':

```
# This database contains a list of both real and fictional people
# along with their age.

Name: Ada Lovelace
Age: 36

Name: Peter the Great
Age: 53

# Name: Matusalem
# Age: 969

Name: Bart Simpson
Age: 10

Name: Adrian Mole
Age: 13.75
```

If we invoke `recsel acquaintances.rec` we will get a list of all the records stored in the file in the terminal:

```
$ recsel acquaintances.rec
Name: Ada Lovelace
Age: 36

Name: Peter the Great
Age: 53

Name: Bart Simpson
Age: 10

Name: Adrian Mole
Age: 13.75
```

Note that the commented out parts of the file, in this case the explanatory header and the record corresponding to Matusalem, are not part of the output produced by `recsel`. This is because `recsel` is concerned only with the data.

`recsel` will also "pack" the records so any extra empty lines that may be between records are not echoed in the output:

acquaintances.rec:

```
Name: Peter the Great
Age: 53

# Note the extra empty lines.


Name: Bart Simpson
Age: 10
```

```
$ recsel acquaintances.rec
Name: Peter the Great
Age: 53

Name: Bart Simpson
Age: 10
```

It is common to store data gathered in several recfiles. For example, we could have a '`contacts.rec`' file containing general contact records, and also a '`work-contacts.rec`' file containing business contacts:

contacts.rec:

```
Name: Granny
Phone: +12 23456677

Name: Doctor
Phone: +12 58999222
```

work-contacts.rec:

```
Name: Yoyodyne Corp.
Email: sales@yoyod.com
Phone: +98 43434433

Name: Robert Harris
Email: robert.harris@yoyod.com
Note: Sales Department.
```

Both files can be passed to `recsel` in the command line. In that case `recsel` will simply process them and output their records in the same order they were specified:

```
$ recsel contacts.rec work-contacts.rec
Name: Granny
Phone: +12 23456677

Name: Doctor
Phone: +12 58999222

Name: Yoyodyne Corp.
Email: sales@yoyod.com
Phone: +98 43434433

Name: Robert Harris
Email: robert.harris@yoyod.com
Note: Sales Department.
```

As mentioned above, the output follows the ordering on the command line, so `recsel work-contacts.rec contacts.rec` would output the records of '`work-contacts.rec`' first and then the ones from '`contacts.rec`'.

Note however that `recsel` will merge records from several files specified in the command line only if they are anonymous. If the contacts in our files were typed:

contacts.rec:

```
%rec: Contact

Name: Granny
Phone: +12 23456677

Name: Doctor
Phone: +12 58999222
```

work-contacts.rec:

```
%rec: Contact

Name: Yoyodyne Corp.
Email: sales@yoyod.com
Phone: +98 43434433

Name: Robert Harris
Email: robert.harris@yoyod.com
Note: Sales Department.
```

Then we would get the following error message:

```
$ recsel contacts.rec work-contacts.rec
recsel: error: duplicated record set 'Contact' from work-contacts.rec.
```

3.2 Selecting by Type

As we saw in the section discussing record descriptors, it is possible to have several different types of records in a single recfile. Consider for example a '`gnu.rec`' file containing information about maintainers and packages in the GNU Project:

```
%rec: Maintainer

Name: Jose E. Marchesi
Email: jemarch@gnu.org

Name: Luca Saiu
Email: positron@gnu.org

%rec: Package

Name: GNU recutils
LastRelease: 12 February 2014

Name: GNU epsilon
LastRelease: 10 March 2013
```

If `recsel` is invoked in that file it will complain:

```
$ recsel gnu.rec
recsel: error: several record types found.  Please use -t to specify one.
```

This is because `recsel` does not know which records to output: the maintainers or the packages. This can be resolved by using the `-t` command line option:

```
$ recsel -t Package gnu.rec
Name: GNU recutils
LastRelease: 12 February 2014

Name: GNU epsilon
LastRelease: 10 March 2013
```

By default `recsel` never outputs record descriptors. This is because most of the time the user is only interested in the data. However, with the `-d` command line option, the record descriptor of the selected type is printed preceding the data records:

```
$ recsel -d -t Maintainer gnu.rec
%rec: Maintainer

Name: Jose E. Marchesi
Email: jemarch@gnu.org

Name: Luca Saiu
Email: positron@gnu.org
```

Note that at the moment it is not possible to select non-typed (default) records when other record sets are stored in the same file. This is one of the reasons why mixing non-typed records and typed records in a single recfile is not recommended.

Note also that if a nonexistent record type is specified in `-t` then `recsel` does nothing.

3.3 Selecting by Position

As was explained in the previous sections, `recsel` outputs all the records of some record set. The records are echoed in the same order they are written in the recfile. However, often it is desirable to select a subset of the records, determined by the position they occupy in their record set.

The `-n` command line option to `recsel` supports doing this in a natural way. This is how we would retrieve the first contact listed in a contacts database using `recsel`:

```
$ recsel -n 0 contacts.rec
Name: Granny
Phone: +12 23456677
```

Note that the index is zero-based. If we want to retrieve more records we can specify several indexes to `-n` separated by commas. If a given index is too big, it is simply ignored:

```
$ recsel -n 0,1,999 contacts.rec
Name: Granny
Phone: +12 23456677

Name: Doctor
Phone: +12 58999222
```

With `-n`, the order in which the records are echoed does not depend on the order of the indexes passed to `-n`. For example, the output of `recsel -n 0,1` will be identical to the output of `recsel -n 1,0`.

Ranges of indexes can also be used to select a subset of the records. For example, the following call would also select the first three contacts of the database:

```
$ recsel -n 0-2 contacts.rec
Name: Granny
Phone: +12 23456677

Name: Doctor
Phone: +12 58999222

Name: Dad
Phone: +12 88229900
```

It is possible to mix single indexes and index ranges in the same call. For example, `recsel -n 0,5-6` would select the first, sixth and seventh records.

3.4 Random Records

Consider a database in which each record is a cooking recipe. It is always difficult to decide what to cook each day, so it would be nice if we could ask `recsel` to pick up a random recipe. This can be achieved using the `-m` (`--random`) command line option of `recsel`:

```
$ recsel -m 1 recipes.rec
Title: Curry chicken
Ingredient: A whole chicken
Ingredient: Curry
Preparation: ...
```

If we need two recipes, because we will be cooking at both lunch and dinner, we can pass a different number to `-m`:

```
$ recsel -m 2 recipes.rec
Title: Fabada Asturiana
Ingredient: 300 gr of fabes.
Ingredient: Chorizo
Ingredient: Morcilla
Preparation: ...

Title: Pasta with ragu
Ingredient: 500 gr of spaghetti.
Ingredient: 2 tomatoes.
Ingredient: Minced meat.
Preparation: ...
```

The algorithm used to implement `-m` guarantees that you will never get multiple instances of the same record. This means that if a record set has *n* records and you ask for *n* random records, you will get all the records in a random order.

3.5 Selection Expressions

Selection expressions, also known as “sexes” in recutils jargon, are infix expressions that can be applied to a record. A “sex” is a predicate which selects a subset of records within a recfile. They can be simple expressions involving just one operator and a pair of operands, or complex compound expressions with parenthetical sub-expressions and many operators and operands. One of their most common uses is to examine records matching a particular set of conditions.

3.5.1 Selecting by predicate

Consider the example recfile ‘`acquaintances.rec`’ introduced earlier. It contains names of people along with their respective ages. Suppose we want to get a list of the names of all the children. It would not be easy to do this using `grep`. Neither would it, for any reasonably large recfile, be feasible to search manually for the children. Fortunately the `recsel` command provides an easy way to do such a lookup:

```
$ recsel -e "Age < 18" -P Name acquaintances.rec
Bart Simpson
Adrian Mole
```

Let us look at each of the arguments to `recsel` in turn. Firstly we have `-e` which tells `recsel` to lookup records matching the expression `Age < 18` — in other words all those people whose ages are less than 18. This is an example of a *selection expression*. In this case it is a simple test, but it can be as complex as needed.

Next, there is `-P` which tells `recsel` to print out the value of the `Name` field — because we want just the name, not the entire record. The final argument is the name of the file from whence the records are to come: ‘`acquaintances.rec`’.

Rather than explicitly storing ages in the recfile, a more realistic example might have the date of birth instead (otherwise it would be necessary to update the people's ages in the recfile on every birthday).

```
# Date of Birth
%type: Dob date

Name: Alfred Nebel
Dob: 20 April 2010
Email: alf@example.com

Name: Bertram Worcester
Dob: 3 January 1966
Email: bert@example.com

Name: Charles Spencer
Dob: 4 July 1997
Email: charlie@example.com

Name: Dirk Hogart
Dob: 29 June 1945
Email: dirk@example.com

Name: Ernest Wright
Dob: 26 April 1978
Email: ernie@example.com
```

Now we can achieve a similar result as before, by looking up the names of all those people who were born after a particular date:

```
$ recfix acquaintances.rec
$ recsel -e "Dob >> '31 July 1994'" -p Name acquaintances.rec
Name: Alfred Nebel
Name: Charles Spencer
```

The `>>` operator means "later than", and is used here to select a date of birth after 31st July 1994. Note also that this example uses a lower case `-p` whereas the preceding example used the upper case `-P`. The difference is that `-p` prints the field name and field value, whereas `-P` prints just the value.

`recsel` accepts more than one `-e` argument, each introducing a selection expression, in which case the records which satisfy all expressions are selected. You can provide more than one field label to `-P` or `-p` in order to select additional fields to be displayed. For example, if you wanted to send an email to all children 14 to 18 years of age, and today's date were 1st August 2012, then you could use the following command to get the name and email address of all such children:

```
$ recfix acquaintances.rec
$ recsel -e "Dob >> '31 July 1994' && Dob << '01 August 1998'" \
  -p Name,Email acquaintances.rec
Name: Charles Spencer
Email: charlie@example.com
```

As you can see, there is only one such child in our record set.

Note that the example command shown above contains both double quotes `"` and single quotes `'`. The double quotes are interpreted by the shell (*e.g.* `bash`) and the single quotes are interpreted by `recsel`, defining a string. (And the backslash is interpreted by the shell, the usual continuation character so that this manual doesn't have a too-long line.)

3.5.2 SEX Operands

The supported operands are: numbers, strings, field names and parenthesized expressions.

3.5.2.1 Numeric Literals

The supported numeric literals are integer numbers and real numbers. The usual sign character '`-`' is used to denote negative values. Integer values can be denoted in base 10, base 16 using the `0x` prefix, and base 8 using the `0` prefix. Examples are:

```
10000
0
0xFF
-0xa
012
-07
-1342
.12
-3.14
```

3.5.2.2 String Literals

String values are delimited by either the `'` character or the `"` character. Whichever delimiter is used, the delimiter closing the literal must be the same as the delimiter used to open it.

Newlines and tabs can be part of a string literal.

Examples are:

```
'Hello.'
'The following example is the empty string.'
''
```

The `'` and `"` characters can be part of a string if they are escaped with a backslash, as in:

```
'This string contains an apostrophe: \'.'
"This one a double quote: \"."
```

3.5.2.3 Field Values

The value of a field value can be included in a selection expression by writing its name. The field name is replaced by a string containing the field value, to handle the possibility of records with more than one field by that name. Examples:

```
Name
Email
long_field_name
```

It is possible to use the role part of a field if it is not empty. So, for example, if we are searching for the issues opened by '`John Smith`' in a database of issues we could write:

```
$ recsel -e "OpenedBy = 'John Smith'"
```

instead of using a full field name:

```
$ recsel -e "Hacker:Name:OpenedBy = 'John Smith'"
```

When the name of a field appears in an expression, the expression is applied to all the fields in the record featuring that name. So, for example, the expression:

```
Email ~ "\\.org"
```

matches any record in which there is a field named '`Email`' whose value terminates in (the literal string) '`.org`'. If we are interested in the value of some specific email, we can specify its relative position in the containing record by using *subscripts*. Consider, for example:

```
    Email[0] ~ "\\.org"
```

Will match for:

```
    Name: Mr. Foo
    Email: foo@foo.org
    Email: mr.foo@foo.com
```

But not for:

```
    Name: Mr. Foo
    Email: mr.foo@foo.com
    Email: foo@foo.org
```

The regexp syntax supported in selection expressions is POSIX EREs, with several GNU extensions. See Chapter 19 [Regular Expressions], page 73.

3.5.2.4 Parenthesized Expressions

Parenthesis characters (`(` and `)`) can be used to group sub expressions in the usual way.

3.5.3 Operators

The supported operators are arithmetic operators (addition, subtraction, multiplication, division and modulus), logical operators, string operators and field operators.

3.5.3.1 Arithmetic Operators

Arithmetic operators for addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`) and modulus (`%`) are supported with their usual meanings.

These operators require either numeric operands or string operands whose value can be interpreted as numbers (integer or real).

3.5.3.2 Boolean Operators

The boolean operators **and** (`&&`), **or** (`||`) and **not** (`!`) are supported with the same semantics as their C counterparts.

A compound boolean operator `=>` is also supported in order to ease the elaboration of constraints in records: `A => B`, which can be read as “A implies B”, translates into `!A || (A && B)`.

The boolean operators expect integer operands, and will try to convert any string operand to an integer value.

3.5.3.3 Comparison Operators

The compare operators **less than** (`<`), **greater than** (`>`), **less than or equal** (`<=`), **greater than or equal** (`>=`), **equal** (`=`) and **unequal** (`!=`) are supported with their usual meaning.

Strings can be compared with the equality operator (`=`).

The match operator (`~`) can be used to match a string with a given regular expression (see Chapter 19 [Regular Expressions], page 73).

3.5.3.4 Date Comparison Operators

The compare operators **before** (`<<`), **after** (`>>`) and **same time** (`==`) can be used with fields and strings containing parseable dates.

See Chapter 20 [Date input formats], page 74.

3.5.3.5 Field Operators

Field counters are replaced by the number of occurrences of a field with the given name in the record. For example:

```
#Email
```

The previous expression is replaced with the number of fields named `Email` in the record. It can be zero if the record does not have a field with that name.

3.5.3.6 String Operators

The string concatenation operator (`&`) can be used to concatenate any number of strings and field values.

```
'foo' & Name & 'bar'
```

3.5.3.7 Conditional Operator

The ternary conditional operator can be used to select alternatives based on the value of some expression:

```
expr1 ? expr2 : expr3
```

If `expr1` evaluates to true (*i.e.* it is an integer or the string representation of an integer and its value is not zero) then the operator yields `expr2`. Otherwise it yields `expr3`.

3.5.4 Evaluation of Selection Expressions

Given that:

- It is possible to refer to fields by name in selection expressions.
- Records can have several fields with the same name.

It is clear that some backtracking mechanism is needed in the evaluation of the selection expressions. For example, consider the following expression that is deciding whether a “registration” in a webpage should be rejected:

```
((Email ~ "foomail\.com") || (Age <= 18)) && !#Fixed
```

The previous expression will be evaluated for every possible permutation of the fields “Email”, “Age” and “Fixed” present in the record, until one of the combinations succeeds. At that point the computation is interrupted.

When used to decide whether a record matches some criteria, the goal of a selection expression is to act as a boolean expression. In that case the final value of the expression depends on both the type and the value of the result launched by the top-most subexpression:

- If the result is an **integer**, the expression is true if its value is not zero.
- If the result is a **real**, or a **string**, the expression evaluates to false.

Sometimes a selection expression is used to compute a result instead of a boolean. In that case the returned value is converted to a string. This is used when replacing the slots in templates (see Section 14.1 [Templates], page 56).

3.6 Field Expressions

Field expressions (also known as “fexes”) are a way to select fields of a record. They also allow you to do certain transformations on the selected fields, such as changing their names.

A FEX comprises a sequence of *elements* separated by commas:

```
ELEM_1,ELEM_2,...,ELEM_N
```

Each element makes a reference to one or more fields in a record identified by a given name and an optional subscript:

```
Field_Name[min-max]
```

min and *max* are zero-based indexes. It is possible to refer to a field occupying a given position. For example, consider the following record:

```
Name: Mr. Foo
Email: foo@foo.com
Email: foo@foo.org
Email: mr.foo@foo.org
```

We would select all the emails of the record with:

```
Email
```

The first email with:

```
Email[0]
```

The third email with:

```
Email[2]
```

The second and the third email with:

```
Email[1-2]
```

And so on. It is possible to select the same field (or range of fields) more than once just by repeating them in a field expression. Thus, the field expression:

```
Email[0],Name,Email
```

will print the first email, the name, and then all the email fields including the first one.

It is possible to include a *rewrite rule* in an element of a field expression, which specifies an alias for the selected fields:

```
Field_Name[min-max]:Alias
```

For example, the following field expression specifies an alias for the fields named `Email` in a record:

```
Name,Email:ElectronicMail
```

Since the rewrite rules only affect the fields selected in a single element of the field expression, it is possible to define different aliases to several fields having the same name but occupying different positions:

```
Name,Email[0]:PrimaryEmail,Email[1]:SecondaryEmail
```

When that field expression is applied to the following record:

```
Name: Mr. Foo
Email: primary@email.com
Email: secondary@email.com
Email: other@email.com
```

the result will be:

```
Name: Mr. Foo
PrimaryEmail: primary@email.com
SecondaryEmail: secondary@email.com
Email: other@email.com
```

It is possible to use the dot notation in order to refer to field and sub-fields. This is mainly used in the context of joins, where new fields are created having compound names such as `Foo_Bar`. A reference to such a field can be done in the fex using dot notation as follows:

```
Foo.Bar
```

3.7 Sorted Output

This special field sets sorting criteria for the records contained in a record set. Its usage is:

```
%sort: field1 field2 ...
```

Meaning that the desired order for the records will be determined by the contents of the fields named in the `%sort` value. The sorting is always done in ascending order, and there may be records that lack the involved fields, *i.e.* the sorting fields need not be mandatory.

It is an error to have more than one `%sort` field in the same record descriptor, as only one field list can be used as sorting criteria.

Consider for example that we want to keep the records in our inventory system ordered by entry date. We could achieve that by using the following record descriptor in the database:

```
%rec: Item
%type: Date date
%sort: Date

Id: 1
Title: Staplers
Date: 10 February 2011

Id: 2
Title: Ruler Pack 20
Date: 2 March 2009

...
```

As you can see in the example above, the fact we use `%sort` in a database does not mean that the database will be always physically ordered. Unsorted record sets are not a data integrity problem, and thus the diagnosis tools must not declare a recfile as +invalid because of this. The utility `recfix` provides a way +to physically order the fields in the file (see Section 17.6 [Invoking recfix], page 68).

On the other hand any program listing, presenting or processing data extracted from the recfile must honor the `%sort` entry. For example, when using the following `recsel` program in the database above we would get the output sorted by date:

```
$ recsel inventory.rec
Id: 2
Title: Ruler Pack 20
Date: 2 March 2009

Id: 1
Title: Staplers
Date: 10 February 2011
```

The sorting of the selected field depends on its type:

– Numeric fields (integers, ranges, reals) are numerically ordered.
– Boolean fields are ordered considering that “false” values come first.
– Dates are ordered chronologically.
– Any other kind of field is ordered using a lexicographic order.

It is possible to specify several fields as the sorting criteria. In that case the records are sorted using a lexicographic order. Consider for example the following unsorted database containing marks for several students:

```
%rec: Marks
%type: Class enum A B C
%type: Score real

Name: Mr. One
Class: C
Score: 6.8

Name: Mr. Two
Class: A
Score: 6.8

Name: Mr. Three
Class: B
Score: 9.2

Name: Mr. Four
Class: A
Score: 2.1

Name: Mr. Five
Class: C
Score: 4
```

If we wanted to sort it by `Class` and by `Score` we would insert a `%sort` special field in the descriptor, having:

```
%rec: Marks
%type: Class enum A B C
%type: Score real
%sort: Class Score

Name: Mr. Four
Class: A
Score: 2.1

Name: Mr. Two
Class: A
Score: 6.8

Name: Mr. Three
Class: B
Score: 9.2

Name: Mr. Five
Class: C
Score: 4

Name: Mr. One
Class: C
Score: 6.8
```

The order of the fields in the `%sort` field is significant. If we reverse the order in the example above then we get a different sorted set:

```
%rec: Marks
%type: Class enum A B C
%type: Score real
%sort: Score Class

Name: Mr. Four
Class: A
Score: 2.1

Name: Mr. Five
Class: C
Score: 4

Name: Mr. Two
Class: A
Score: 6.8

Name: Mr. One
Class: C
Score: 6.8

Name: Mr. Three
Class: B
Score: 9.2
```

In this last case, `Mr. One` comes after `Mr. Two` because the class `A` comes before the class `B` even though the score is the same (`6.8`).

4 Editing Records

The simplest way of editing a recfile is to start your favourite text editor and hack the contents of the file as desired. However, the rec format is structured enough so recfiles can be updated automatically by programs. This is useful for writing shell scripts or when there are complex data integrity rules stored in the file that we want to be sure to preserve.

The following sections discuss the usage of the recutils for altering recfiles in the level of record: adding new records, deleting or commenting them out, sorting them, *etc.*

4.1 Inserting Records

Adding new records to a recfile is pretty trivial: open it with your text editor and just write down the fields comprising the records. This is really the best way to add contents to a recfile containing simple data. However, complex databases may introduce some difficulties:

Multi-line values.
It can be tedious to manually encode the several lines.

Data integrity.
It is difficult to manually maintain the integrity of data stored in the data base.

Counters and timestamps.
Some record sets feature auto-generated fields, which are commonly used to implement counters and time-stamps. See Chapter 12 [Auto-Generated Fields], page 49.

Thus, to facilitate the insertion of new data a command line utility called `recins` is included in the recutils. The usage of `recins` is very simple, and can be used both in the command line or called from another program. The following subsections discuss several aspects of using this utility.

4.1.1 Adding Records With recins

Each invocation of `recins` adds one record to the targeted database. The fields comprising the records are specified using pairs of `-f` and `-v` command line arguments. For example, this is how we would add the first entry to a previously empty contacts database:

```
$ recins -f Name -v "Mr Foo" -f Email -v foo@bar.baz contacts.rec
$ cat contacts.rec
Name: Mr. Foo
Email: foo@bar.baz
```

If we invoke `recins` again on the same database we will be adding a second record:

```
$ recins -f Name -v "Mr Bar" -f Email -v bar@gnu.org contacts.rec
$ cat contacts.rec
Name: Mr. Foo
Email: foo@bar.baz

name: Mr. Bar
Email: bar@gnu.org
```

There is no limit on the number of `-f -v` pairs that can be specified to `recins`, other than any limit on command line arguments which may be imposed by the shell.

The field values provided using `-v` are encoded to follow the rec format conventions, including multi-line field values. Consider the following example:

```
$ recins -f Name -v "Mr. Foo" -f Address -v ’
Foostrs. 19
Frankfurt am Oder
```

```
Germany’ contacts.rec
$ cat contacts.rec
Name: Mr. Foo
Address:
+ Foostrs. 19
+ Frankfurt am Oder
+ Germany
```

It is also possible to provide fields already encoded as rec data for their addition, using the `-r` command line argument. This argument can be intermixed with `-f -v`.

```
$ recins -f Name -v "Mr. Foo" -r "Email: foo@bar.baz" contacts.rec
$ cat contacts.rec
Name: Mr. Foo
Email: foo@bar.baz
```

If the string passed to `-r` is not valid rec data then `recins` will complain with an error and the operation will be aborted.

At this time, it is not possible to add new records containing comments.

4.1.2 Replacing Records With recins

`recins` can also be used to replace existing records in a database with a provided record. This is done by specifying some criteria selecting the record (or records) to be replaced.

Consider for example the following command applied to our contacts database:

```
$ recins -e "Email = ’foo@bar.baz’" -f Name -v "Mr. Foo" \
    -f Email -v "new@bar.baz" contacts.rec
```

The contact featuring an email `foo@bar.baz` gets replaced with the following record:

```
Name: Mr. Foo
Email: new@bar.baz
```

The records to be replaced can also be specified by index, or a range of indexes. For example, the following command replaces the first, second and third records in a database with dummy records:

```
$ recins -n 0,1-2 -f Dummy -v XXX foo.rec
$ cat foo.rec
Dummy: XXX

Dummy: XXX

Dummy: XXX

... Other records ...
```

4.1.3 Adding Anonymous Records

In a previous chapter we noted that `recsel` interprets the absence of a `-t` argument depending on the actual contents of the file. If the recfile contains records of just one type the command assumes that the user is referring to these records.

`recins` does not follow this convention, and the absence of an explicit type always means to insert (or replace) an anonymous record. Consider for example the following database:

```
%rec: Marks
%type: Class enum A B C

Name: Alfred
```

```
Class: A

Name: Bertram
Class: B
```

If we want to insert a new mark we have to specify the type explicitly using `-t`:

```
$ cat marks.rec | recins -t Marks -f Name -v Xavier -f Class -v C
%rec: Marks
%type: Class enum A B C

Name: Alfred
Class: A

Name: Bertram
Class: B

Name: Xavier
Class: C
```

If we forget to specify the type then an anonymous record is created instead:

```
$ cat marks.rec | recins -f Name -v Xavier -f Class -v C
Name: Xavier
Class: C

%rec: Marks
%type: Class enum A B C

Name: Alfred
Class: A

Name: Bertram
Class: B
```

4.2 Deleting Records

Just as `recins` inserts records, the utility `recdel` deletes them. Consider the following recfile ‘`stock.rec`’:

```
%rec: Item
%type: Expiry date
%sort: Title

Title: First Aid Kit
Expiry: 2 May 2009

Title: Emergency Rations
Expiry: 10 August 2009

Title: Life raft
Expiry: 2 March 2009
```

Suppose we wanted to delete all items with an `Expiry` value before a certain date, we could do this with the following command:

```
$ recdel -t Item -e 'Expiry << "5/12/2009"' stock.rec
```

After running this command, only one record will remain in the file (*viz:* the one titled 'Emergency Rations') because all the others have expiry dates prior to 12 May 2009.[1] The `-t` option can be omitted if, and only if, there is no `%rec` field in the recfile.

`recdel` tries to warn you if you attempt to perform a delete operation which it deems to be too pervasive. In such cases, it will refuse to run, unless you give the `--force` flag. However, you should not rely upon `recdel` to protect you, because it cannot always correctly guess that you might be deleting more records than intended. For this reason, it may be wise to use the `-c` flag, which causes the relevant records to be commented out, rather than deleted. (And of course backups are always wise.)

The complete options available to the `recdel` command are explained later. See Section 17.4 [Invoking recdel], page 66.

4.3 Sorting Records

In the example above, note the existence of the `%sort: Title` line. This field was discussed previously (see Section 3.7 [Sorted Output], page 20) and, as mentioned, does not imply that the records need to be stored in the recfile in any particular order.

However, if desired, you can automatically arrange the recfile in that order using `recfix` with the `--sort` flag. After running the command

```
$ recfix --sort stock.rec
```

the file '`stock.rec`' will have its records sorted in alphabetical order of the `Title` fields, thus:

```
%rec: Item
%type: Expiry date
%sort: Title

Title: Emergency Rations
Expiry: 10 August 2009

Title: First Aid Kit
Expiry: 2 May 2009

Title: Liferaft
Expiry: 2 March 2009
```

[1] '5/12/2009' means the 12th day of May 2009, *not* the fifth day of December, even if your `LC_TIME` environment variable has been set to suggest otherwise.

5 Editing Fields

Fields of a recfile can, of course, be edited manually using an editor and this is often the easiest way when only a few fields need to be changed or when the nature of the changes do not follow any particular pattern. If, however, a large number of similar changes to several records are required,the `recset` command can make the job easier.

The formal description of `recset` is presented later (see Section 17.5 [Invoking recset], page 67). In this chapter some typical usage examples are discussed. As with `recdel`, `recset` if used erroneously has the potential to make very pervasive changes, which could result in a large loss of data. It is prudent therefore to take a copy of a recfile before running such commands.

5.1 Adding Fields

As mentioned above, the command `recins` adds new records to a recfile, but it cannot add fields to an existing record. This task can be achieved automatically using `recset` with its `-a` flag.

Suppose that (after a stock inspection) you wanted to add an 'Inspected' field to all the items in the recfile. The following command could be used.

```
$ recset -t Item -f Inspected -a 'Yes' stock.rec
```

Here, because no record selection flag was provided, the command affected *all* the records of type 'Item'. We could limit the effect of the command using the `-e`, `-q`, `-n` or `-m` flags. For example to add the 'Inspected' field to only the first item the following command would work:

```
$ recset -t Item -n 0 -f Inspected -a 'Yes' stock.rec
```

Similarly, a selection expression could have been used with the `-e` flag in order to add the field only to records which satisfy the expression.

If you use `recset` with the `-a` flag on a field that already exists, a new field (in addition to those already present) will be appended with the given value.

5.2 Setting Fields

It is also possible to update the value of a field. This is done using `recset` with its `-s` flag. In the previous example, an 'Inspected' flag was added to certain records, with the value 'yes'. After reflection, one might want to record the date of inspection, rather than a simple yes/no flag. Records which have no such field will remain unchanged.

```
$ recset -t Item -f Inspected -s '30 October 2006' stock.rec
```

Although the above command does not have any selection criteria, it will only affect those records for which a 'Inspected' field exists. This is because the `-s` flag only sets values of existing fields. It will not create any fields.

If instead the `-S` flag is used, this will create the field (if it does not already exist) *and* set its value.

```
$ recset -t Item -f Inspected -S '30 October 2006' stock.rec
```

5.3 Deleting Fields

You can delete fields using `recset`'s `-d` flag. For example, if we wanted to delete the `Inspected` field which we introduced above, we could do so as follows:

```
$ recset -t Item -f Inspected -d stock.rec
```

This would delete *all* fields named `Inspected` from *all* records of type `Item`. It may be that, we only wanted to delete the `Inspected` fields from records which satisfy a certain condition. The following would delete the fields only from items whose `Expiry` date was before 2 January 2010:

```
$ recset -t Item -e 'Expiry << "2 January 2010"' -f Inspected -d stock.rec
```

5.4 Renaming Fields

Another use of `recset` is to rename existing fields. This is achieved using the `-r` flag. To rename all instances of the `Expiry` field occurring in any record of type `Item` to `UseBy`, the following command suffices:

```
$ recset -t Item -f Expiry -r 'UseBy' stock.rec
```

As with most operations, this could be done selectively, using the `-e` flag and a selection expression.

6 Field Types

Field values are, by default, unrestricted text strings. However, it is often useful to impose some restrictions on the values of certain fields. For example, consider the following record:

```
Id: 111
Name: Jose E. Marchesi
Age: 30
MaritalStatus: single
Phone: +49 666 666 66
```

The values of the fields must clearly follow some structure in order to make sense. `Id` is a numeric identifier for a person. `Name` will never use several lines. `Age` will typically be in the range `0..120`, and there are only a few valid values for `MaritalStatus`: single, married, divorced, and widow(er). Phones may be restricted to some standard format as well to be valid. All these restrictions (and many others) can be enforced by using *field types*.

There are two kind of field types: *anonymous* and *named*. Those are described in the following subsections.

6.1 Declaring Types

A type can be declared in a record descriptor by using the `%typedef` special field. The syntax is:

```
%typedef: type_name type_description
```

Where *type_name* is the name of the new type, and *type_description* a description which varies depending of the kind of type. For example, this is how a type `Age_t` could be defined as numbers in the range `0..120`:

```
%typedef: Age_t range 0 120
```

Type names are identifiers having the following syntax:

```
[a-zA-Z][a-zA-Z0-9_]*
```

Even though any identifier with that syntax could be used for types, it is a good idea to consistently follow some convention to help distinguishing type names from field names. For example, the `_t` suffix could be used for types.

A type can be declared to be an alias for another type. The syntax is:

```
%typedef: type_name other_type_name
```

Where *type_name* is declared to be a synonym of *other_type_name*. This is useful to avoid duplicated type descriptions. For example, consider the following example:

```
%typedef: Id_t          int
%typedef: Item_t        Id_t
%typedef: Transaction_t Id_t
```

Both `Item_t` and `Transaction_t` are aliases for the type `Id_t`. Which is in turn an alias for the type `int`. So, they are both numeric identifiers.

The order of the `%typedef` fields is not relevant. In particular, a type definition can forward-reference another type that is defined subsequently. The previous example could have been written as:

```
%typedef: Item_t        Id_t
%typedef: Transaction_t Id_t
%typedef: Id_t          int
```

Integrity check will complain if undefined types are referenced. As well as when any aliases up referencing back (looping back directly or indirectly) in type declarations. For example, the following set of declarations contains a loop. Thus, it's invalid:

```
%typedef: A_t B_t
%typedef: B_t C_t
%typedef: C_t A_t
```

The scope of a type is the record descriptor where it is defined.

6.2 Types and Fields

Fields can be declared to have a given type by using the `%type` special field in a record descriptor. The synopsis is:

```
%type: field_list type_name_or_description
```

Where *field_list* is a list of field names separated by commas. *type_name_or_description* can be either a type name which has been previously declared using `%typedef`, or a type description. Type names are useful when several fields are declared to be of the same type:

```
%typedef: Id_t    int
%type:    Id      Id_t
%type:    Product Id_t
```

Anonymous types can be specified by writing a type description instead of a type name. They help to avoid superfluous type declarations in the common case where a type is used by just one field. A record containing a single `Id` field, for example, can be defined without having to use a `%typedef` in the following way:

```
%rec: Task
%type: Id int
```

6.3 Scalar Field Types

The rec format supports the declaration of fields of the following scalar types: integer numbers, ranges and real numbers.

Signed *integers* are supported by using the `int` declaration:

```
%typedef: Id_t int
```

Given the declaration above, fields of type `Id_t` must contain integers, and they may be negative. Hexadecimal values can be written using the `0x` prefix, and octal values using an extra `0`. Valid examples are:

```
%type: Id Id_t

Id: 100
Id: -23
Id: -0xFF
Id: 020
```

Sometimes it is desirable to reduce the *range* of integers allowed in a field. This can be achieved by using a range type declaration:

```
%typedef: Interrupt_t range 0 15
```

Note that it is possible to omit the minimum index in ranges. In that case it is implicitly zero:

```
%typedef: Interrupt_t range 15
```

It is possible to use the keywords `MIN` and `MAX` instead of a numeral literal in one or both of the points conforming the range. They mean the minimum and the maximum integer value supported by the implementation respectively. See the following examples:

```
%typedef: Negative range MIN -1
%typedef: Positive range 0 MAX
%typedef: AnyInt range MIN MAX
```

```
%typedef: Impossible range MAX MIN
```

Hexadecimal and octal numbers can be used to specify the limits in a range. This helps to define scalar types whose natural base is not ten, like for example:

```
%typedef: Address_t range 0x0000 0xFFFF
%typedef: Perms_t range 755
```

Real number fields can be declared with the `real` type specifier. A wide range of real numbers can be represented this way, only limited by the underlying floating point representation. The decimal separator is always the dot (`.`) character regardless of the locale setting. For example:

```
%typedef: Longitude_t real
```

Examples of fields of type real:

```
%rec: Rectangle
%typedef: Longitude_t real
%type: Width  Longitude_t
%type: Height Longitude_t

Width: 25.01
Height: 10
```

6.4 String Field Types

The `line` field type specifier can be used to restrict the value of a field to a single line, *i.e.* no newline characters are allowed. For example, a type for proper names could be declared as:

```
%typedef: Name_t line
```

Examples of fields of type line:

```
Name: Mr. Foo Bar
Name: Mrs. Bar Baz
Name: This is
+ invalid
```

Sometimes it is the maximum size of the field value that shall be restricted. The `size` field type specifier can be used to define the maximum number of characters a field value can have. For example, if we were collecting input that will get written in a paper-based forms system allowing up to 25 characters width entries, we could declare the entries as:

```
%typedef: Address_t size 25
```

Note that hexadecimal and octal integer constants can also be used to specify field sizes:

```
%typedef: Address_t size 0x18
```

Arbitrary restrictions can be defined by using regular expressions. The *regexp* field type specifier introduces an ERE (extended regular expression) that will be matched against fields having that name. The synopsis is:

```
%typedef: type_name regexp /re/
```

where *re* is the regular expression to match.

For example, consider the `Id_t` type designed to represent the encoding of the identifier of ID cards in some country:

```
%typedef: Id_t regexp /[0-9]{9}[a-zA-Z]/
```

Examples of fields of type `Id_t` are:

```
IDCard: 123456789Z
IDCard: invalid id card
```

Note that the slashes delimiting the *re* can be replaced with any other character that is not itself used as part of the regexp. That is useful in some cases such as:

```
%typedef: Path_t regexp |(/[^/]/?)+|
```

The regexp flavor supported in recfiles are the POSIX EREs plus several GNU extensions. See Chapter 19 [Regular Expressions], page 73.

6.5 Enumerated Field Types

Fields of this type contain symbols taken from an enumeration.

The type is described by writing the sequence of symbols comprising the enumeration. Enumeration symbols are strings described by the following regexp:

```
[a-zA-Z0-9][a-zA-Z0-9_-]*
```

The symbols are separated by blank characters (including newlines). For example:

```
%typedef: Status_t enum NEW STARTED DONE CLOSED
%typedef: Day_t enum Monday Tuesday Wednesday Thursday Friday
+                    Saturday Sunday
```

It is possible to insert comments when describing an enum type. The comments are delimited by parenthesis pairs. The contents of the comments can be any character but parentheses. For example:

```
%typedef: TaskStatus_t enum
+ NEW         (The task was just created)
+ IN_PROGRESS (Task started)
+ CLOSED      (Task closed)
```

Boolean fields, declared with the type specifier `bool`, can be seen as special enumerations holding the binary values true and false.

```
%typedef: Yesno_t bool
```

The literals allowed in boolean fields are `yes/no`, `0/1` and `true/false`. Examples are:

```
SwitchedOn: 1
SwitchedOn: yes
SwitchedOn: false
```

6.6 Date and Time Types

The *date* field type specifier can be used to declare dates and times. The synopsis is:

```
%typedef: type_name date
```

There are many permitted date formats, described in detail later in this manual (see Chapter 20 [Date input formats], page 74). Of particular note are the following:

- Dates and times read from recfiles are not affected by the locale or the timezone. This means that the `LC_TIME` and the `TZ` environment variables are ignored. If you wish, for example, to specify a time which must be interpreted as UTC, you must explicitly append the time zone correction: *e.g.* '`2001-1-10 12:09Z`'.
- The field value '1/10/2001' means January 10, 2001, **not** October 1, 2001.
- Relative times and dates (such as '1 day ago') are permitted but are not particularly useful.

6.7 Other Field Types

The *Email* field type specifier is used to declare electronic addresses. The synopsis is:

```
%typedef: Email_t email
```

Sometimes it is useful to make fields to store field names. For that purpose the *Field* field type specifier is supported. The synopsis is:

```
%typedef: Field_t field
```

Universally Unique Identifiers (also known as UUIDs) are a way to assign a globally unique label to some object. The *uuid* field type specifier serves that purpose. The synopsis is:

```
%typedef: Id_t uuid
```

The format of the uuids is specified as 32 hexadecimal digits, displayed in five groups separated by hyphens. For example:

```
550e8400-e29b-41d4-a716-446655440000
```

There is one other possible field type, *viz:* a foreign key. The following example defines the type `Maintainer_t` to be of type "record `Hacker`"; in other words, a foreign key referring to a record in the `Hacker` record set.

```
%typedef: Maintainer_t rec Hacker
```

This essentially means that the values to be stored in fields of type `Maintainer_t` are of whatever type is defined for the primary key of the `Hacker` record set. Why this is useful is discussed later. See Chapter 11 [Queries which Join Records], page 45.

7 Constraints on Record Sets

The records in a recfile are by default not restricted to any particular structure except that they must contain one or more fields and optional comments. This provides the format with huge expressive power; but in many cases, it is also desirable to impose some restrictions in order to reflect some of the properties of the data stored in the database. It is also useful in order to preserve data integrity and thus avoid data corruption.

The following sections describe the usage of some predefined special fields whose purpose is to impose this kind of restriction in the structure of the records.

7.1 Mandatory Fields

Sometimes, you want to make sure that *every* record of a particular type contains certain fields. To do this, use the special field `%mandatory`. The usage is:

```
%mandatory: field1 field2 ... fieldN
```

The field names are separated by one or more blank characters.

The fields listed in a `%mandatory` entry are non-optional; *i.e.* at least one field with this name shall be present in any record of this kind. Records violating this restriction are invalid and a checking tool will report the situation as a data integrity failure.

Consider for example an "address book" database where each record stores the information associated with a contact. The records will be heterogeneous, in the sense they won't all contain exactly the same fields: the contact of an Internet shop will probably have a `URL` field, while the entry for our grandmother probably won't. We still want to make sure that every entry has a field with the name of the contact. In this case, we could use `%mandatory` as follows:

```
%rec: Contact
%mandatory: Name

Name: Granny
Phone: +12 23456677

Name: Yoyodyne Corp.
Email: sales@yoyod.com
Phone: +98 43434433
```

A word of caution, however: In many situations, especially in day to day social interaction, it is common to find that certain information is simply unavailable. For example, although every person has a date of birth, some people will refuse to provide that information.

It is probably wise therefore to avoid stipulating a field as mandatory, unless it is essential to the enterprise. Otherwise, a data entry clerk faced with this situation will have to make the choice between dropping the entry entirely or entering some fake data to keep the system happy.

7.2 Prohibited Fields

The inverse of `%mandatory` is `%prohibit`. Prohibited fields may not occur in *any* record of the given type. The usage is:

```
%prohibit: field1 field2 ... fieldN
```

The field names are separated by one or more blank characters.

Fields listed in a `%prohibit` entry are forbidden; *i.e.* no field with this name should be present in any record of this kind. Again, records violating this restriction are invalid.

Several `%prohibit` fields can appear in the same record descriptor. The set of prohibited fields is the union of all the entries. For example, in the following database both `Id` and `id` are prohibited:

```
%rec: Entry
%prohibit: Id
%prohibit: id
```

One possible use case for prohibited fields arises when some field name is reserved for some future use. For example, if we were organizing a sports competition, we would want competitors to register before the event. However a competitor's `result` should not and cannot be entered before the competition takes place. Initially then, we would change the record descriptor as follows:

```
%rec: Contact
%mandatory: Name
%prohibit: result
```

At the start of the event, the `%prohibit` line can be deleted, to allow results to be entered.

7.3 Allowed Fields

In some cases we know the set of fields that may appear in the records of a given type, even if they are not mandatory. The `%allowed` special field is used to specify this restriction. The usage is:

```
%allowed: field1 field2 ... fieldN
```

The field names are separated by one or more blank chracters.

If there are more or one `%allowed` fields in a record descriptor, all fields of all the records in the record set must be in the union of `%allowed`, `%mandatory` and `%key`. Otherwise an integrity error is raised.

Several `%allowed` fields can appear in the same record descriptor. The set of allowed fields is the union of all the entries.

7.4 Keys and Unique Fields

The `%unique` and `%key` special fields are used to avoid several instances of the same field in a record, and to implement keys in record sets. Their usage is:

```
%unique: field1 field2 ... fieldN
%key: field
```

The field names are separated by one or more blank characters.

Normally it is permitted for a record to contain two or more fields of the same name. The `%unique` special field revokes this permissiveness. A field declared "unique" cannot appear more than once in a single record.

For example, an entry in an address book database could contain an `Age` field. It does not make sense for a single person to be of several ages. So, a field could be declared as "unique" in the corresponding record descriptor as follows:

```
%rec: Contact
%mandatory: Name
%unique: Age
```

Several `%unique` fields can appear in the same record descriptor. The set of unique fields is the union of all the entries.

`%key` makes the referenced field the primary key of the record set. The primary key behaves as if both `%unique` and `%mandatory` had been specified for that field. Additionally, there is further restriction, *viz:* a given value of a primary key field may appear no more than once within a record set.

Consider for example a database of items in stock. Each item is identified by a numerical `Id` field. No item may have more than one `Id`, and no items may exist without an associated `Id`.

Additionally, no two items may share the same `Id`. This common situation can be implementing by declaring `Id` as the key in the record descriptor:

```
%rec: Item
%key: Id
%mandatory: Title

Id: 1
Title: Box

Id: 2
Title: Sticker big
```

It would not make sense to have several primary keys in a record set. Thus, it is not allowed to have several `%key` fields in the same record descriptor. It is also forbidden for two items to share the same 'Id' value. Both of these situations would be data integrity violations, and will be reported by a checking tool.

Elsewhere, we discuss how primary keys can be used to link one record set to another using primary keys together with foreign keys. See Chapter 11 [Queries which Join Records], page 45.

7.5 Singular Fields

Sometimes we require certain fields with a given name to not appear in a record set featuring the same contents, but we don't want (or we can't) declare such fields as the key of the record set.

In these circumstances we can use *singular fields*, which are declared as such in the record descriptor using the `%singular` special field:

```
%singular: field
```

7.6 Size Constraints

Sometimes it is desirable to place constraints on entire records. This can be done with the `%size` special field which is used to limit the number of records in a record set. Its usage is:

```
%size: [relational_operator] number
```

If no operator is specified then *number* is interpreted as the exact number of records of this type. The number can be any integer literal, including hexadecimal and octal constants. For example:

```
%rec: Day
%size: 7
%type: Name enum
+ Monday Tuesday Wednesday Thursday Friday
+ Saturday Sunday
%doc: There should be exactly 7 days.
```

The optional *relational_operator* shall be one of `<`, `<=`, `>` and `>=`. For example:

```
%rec: Item
%key: Id
%size: <= 100
%doc: We have at most 100 different articles.
```

It is valid to specify a size of `0`, meaning that no records of this type shall exist in the file.

Only one `%size` field shall appear in a record descriptor.

7.7 Arbitrary Constraints

Occasionally, `%mandatory`, `%prohibit` and `%size` are just not flexible enough. We might, for instance, want to ensure that *if* a field is present, then it must have a certain relationship to other fields. Or we might want to stipulate that under certain conditions only, a record contains a particular field.

To this end, recutils provides a way for arbitrary field constraints to be defined. These permit restrictions on the presence and/or value of fields, based upon the value or presence of other fields within that record. This is done using the `%constraint` special field. Its usage is:

```
%constraint: expr
```

where *expr* is a selection expression (see Section 3.5 [Selection Expressions], page 14). When a constraint is present in a record set it means that all the records of that type must satisfy the selection expression, *i.e.* the evaluation of the expression with the record returns 1. Otherwise an integrity error is raised.

Consider for example a record type `Task` featuring two fields of type date called `Start` and `End`. We can use a constraint in the record set to specify that the task cannot start after it finishes:

```
%rec: Task
%type: Start,End date
%constraint: Start << End
```

The "implies" operator `=>` is especially useful when defining constraints, since it can be used to specify conditional constraints, *i.e.* constraints applying only in certain records. For example, we could specify that if a task is closed then it must have an `End` date in the following way:

```
%rec: Task
%type: Start,End date
%constraint: Start << End
%constraint: Status = 'CLOSED' => #End
```

It is acceptable to declare several constraints in the same record set.

8 Checking Recfiles

Sometimes, when creating a recfile by hand, typographical errors or other mistakes will occur. If a recfile contains such mistakes, then one cannot rely upon the results of queries or other operations. Fortunately there is a tool called `recfix` which can find these errors. It is a good idea to get into the habit of running `recfix` on a file after editing it, and before trying other commands.

8.1 Syntactical Errors

One easy mistake is to forget the colon separating the field name from its value.

```
%rec: Article
%key  Id

Name: Thing
Id:   0
```

Running `recfix` on this file will immediately tell us that there is a problem:

```
$ recfix --check inventory.rec
inventory.rec: 2: error: expected a record
```

Here, `recfix` has diagnosed a problem in the file ‘`inventory.rec`’ and the problem lies at line 2. If, as in this case, `recfix` shows there is a problem with the recfile, you should attend to that problem before trying to use any other recutils program on that file, otherwise strange things could happen. The `--check` flag is optional but in normal execution not required because that is the default operation.

8.2 Semantic Errors

However `recfix` checks more than the syntactical integrity of the recfile. It also checks certain semantics and that the data is self-consistent. To do this, it uses the special fields of the record, some of which were introduced above (see Chapter 7 [Constraints on Record Sets], page 34). It is a good idea to use the special fields to stipulate the “enterprise rules” of the data.

Errors will be reported if any of the following special keywords are present and the data does not match the stipulated conditions

`%mandatory`
: The mandated fields are missing from a record.

`%prohibit`
: The prohibited fields are present in a record.

`%unique`
: There is more than one field in a single record of the given name.

`%key`
: Two or more records share the same value of the field which is the key field.

`%typedef` and `%type`
: A field has a value which does not conform to the specified type.

`%size`
: The number of records does not conform to the specified restriction.

`%constraint`
: A field does not conform to the specified constraint.

`%confidential`
: An unencrypted value exists for a confidential field.

9 Remote Descriptors

The `%rec` special field is used for two main purposes: to identify a record as a record descriptor, and to provide a name for the described record set. The synopsis of the usage of the field is the following:

```
%rec: type [url_or_file]
```

type is the name of the kind of records described by the descriptor. It is mandatory to specify it, and it follows the same lexical conventions used by field names. See Section 2.1 [Fields], page 4. There is a non-enforced convention to use singular nouns, because the name makes reference to the type of a single entity, even if it applies to all the records contained in the record set. For example, the following record set contains transactions, and the type specified in the record descriptor is `Transaction`.

```
%rec: Transaction

Id: 10
Title: House rent

Id: 11
Title: Loan
```

Only one `%rec` field should be in a record descriptor. If there are more it is an integrity violation. It is highly recommended (but not enforced) to place this field in the first position of the record descriptor.

Sometimes it is convenient to store records of the same type in different files. The duplication of record descriptors in this case would surely lead to consistency problems. A possible solution would be to keep the record descriptor in a separated file and then include it in any operation by using pipes. For example:

```
$ cat descriptor.rec data.rec | recsel ...
```

For those cases it is more convenient to use a *external descriptor*. External descriptors can be built appending a file path to the `%rec` field value, like:

```
%rec: FSD_Entry /path/to/file.rec
```

The previous example indicates that a record descriptor describing the `FSD_Entry` records shall be read from the file '`/path/to/file.rec`'. A record descriptor for `FSD_Entry` may not exist in the external file. Both relative and absolute paths can be specified there.

URLs can be used as sources for external descriptors as well. In that case we talk about *remote descriptors*. For example:

```
%rec: Department http://www.myorg.com/Org.rec
```

The URL shall point to a text file containing rec data. If there is a record descriptor in the remote file documenting the `Department` type, it will be used.

Note that the local record descriptor can provide additional fields to "expand" the record type. For example:

```
%rec: FSD_Entry http://www.jemarch.net/downloads/FSD.rec
%mandatory: Rating
```

The record descriptor above is including the contents of the `FSD_Entry` record descriptor from the URL, and adding them to the local record descriptor, that in this case contains just the `%mandatory` field.

If you are using GNU recutils (see Chapter 17 [Invoking the Utilities], page 62) to process your recfiles, any URL schema supported by `libcurl` will work.

10 Grouping and Aggregates

Grouping and aggregate functions are two related features which are useful to extract statistics from a record set, or a subset of that record set.

10.1 Grouping Records

Consider a recfile containing a list of items in a shop inventory. For each item it is stored its type, its category, its price, the date of the last selling operation of an item of that type, and the amount of items currently available in stock. A sample of such a database could be:

```
Type: EC Car
Category: Toy
Price: 12.2
LastSell: 20-April-2012
Available: 623

Type: Terria
Category: Food
Price: 0.60
LastSell: 22-April-2012
Available: 8239

Type: Typex
Category: Office
Price: 1.20
LastSell: 22-April-2012
Available: 10878

Type: Notebook
Category: Office
Price: 1.00
LastSell: 21-April-2012
Available: 77455

Type: Sexy Puzzle
Category: Toy
Price: 6.20
LastSell: 6.20
Available: 12
```

Now imagine we are interested in grouping the contents of the `Items` record set in groups of items of the same category. We can do it using the `-G` command line argument for `recsel`. This argument accepts a list of fields separated by commas. The argument can be read as “group by”.

In this case we want to group by `Category`, so we would do:

```
$ recsel -G Category
Type: Terria
Category: Food
Price: 0.60
LastSell: 22-April-2012
Available: 8239
```

```
Type: Typex
Category: Office
Price: 1.20
LastSell: 22-April-2012
Available: 10878
Type: Notebook
Price: 1.00
LastSell: 21-April-2012
Available: 77455

Type: EC Car
Category: Toy
Price: 12.2
LastSell: 20-April-2012
Available: 623
Type: Sexy Puzzle
Price: 6.20
LastSell: 6.20
Available: 12
```

We can see that the output is three records, corresponding to the three different categories of items present in the database. However, we are only interested in the types of products in each category, so we can remove unwanted information using `-p`:

```
$ recsel -G Category -p Category,Type items.rec
Category: Food
Type: Terria

Category: Office
Type: Typex
Type: Notebook

Category: Toy
Type: EC Car
Type: Sexy Puzzle
```

It is also possible to group by several fields. We could group by both `Category` and `LastSell`:

```
$ recsel -G Category,LastSell -p Category,LastSell,Type items.rec
Category: Food
LastSell: 22-April-2012
Type: Terria

Category: Office
LastSell: 21-April-2012
Type: Notebook

Category: Office
LastSell: 22-April-2012
Type: Typex

Category: Toy
LastSell: 20-April-2012
Type: EC Car
```

```
Category: Toy
LastSell: 6.20
Type: Sexy Puzzle
```

10.2 Aggregate Functions

recutils supports *aggregate functions*. These are so called because they accept a record set and a field name as inputs and generate a single result. Usually this result is numerical.

The supported aggregate functions are the following:

`Count(FIELD)`
: Counts the number of occurrences of a field.

`Avg(FIELD)`
: Calculates the average (mean) of the numerical values of a field.

`Sum(FIELD)`
: Calculates the sum of the numerical values of a field.

`Min(FIELD)`
: Calculates the minimum numerical value of a field.

`Max(FIELD)`
: Calculates the maximum numerical value of a field.

The aggregate functions are to be invoked in the field expressions in `recsel`. By default they are applied to the totality of the records in a record set. For example, using the items database from the previous section, we can do calculations as in the following examples.

The SQL aggregate functions can be applied to the totality of the tuples in the relation. For example, using the `Count` aggregate function we can calculate the number of fields named `Category` present in the record set as follows:

```
$ recsel -p "Count(Category)" items.rec
Count_Category: 5
```

The result is a field whose name is derived from the function name and the field passed as its parameter, separated by an underline. This name scheme probably suffices for most purposes, but it is always possible to use a rewrite rule to obtain something different:

```
$ recsel -p "Count(Category):NumCategories" items.rec
NumCategories: 5
```

You can use different letter case in writing the name of the aggregate, and this will be reflected in the field name:

```
$ recsel -p "CoUnT(Category)" items.rec
CoUnT_Category: 5
```

It is possible to use more than one aggregate function in the field expression. Suppose we are also interested in the average price of the items we sell. We can use the `Avg` aggregate:

```
$ recsel -p "Count(Category),Avg(Price)" items.rec
Count_Category: 5
Avg_Price: 4.240000
```

Now let's add a field along with an aggregate function to the field expression and see what we get:

```
$ recsel -p "Type,Avg(Price)" items.rec
Type: EC Car
Avg_Price: 12.200000
```

```
Type: Terria
Avg_Price: 0.600000

Type: Typex
Avg_Price: 1.200000

Type: Notebook
Avg_Price: 1

Type: Sexy Puzzle
Avg_Price: 6.200000
```

We get five records! The reason is that when *only* aggregate functions are part of the field expression, they are applied to the single record that would result from concatenating all the records in the record set together. However, when a regular field appears in the field expression the aggregate functions are applied to the individual records. This is still useful in some cases, such as a database of maintainers:

```
Name: Jose E. Marchesi
Email: jemarch@gnu.org
Email: jemarch@es.gnu.org

Name: Luca Saiu
Email: positron@gnu.org
```

Lets see how many emails each maintainer has:

```
$ recsel -p "Name,Count(Email)" maintainers.rec
Name: Jose E. Marchesi
Count_Email: 2

Name: Luca Saiu
Count_Email: 1
```

Aggregate functions are most useful when we combine them with grouping. This is when we are interested in some property of a subset of the records in the database. For example, the average prices of each item category stored in the database can be obtained by executing:

```
$ recsel -p "Category,Avg(Price)" -G Category items.rec
Category: Food
Avg_Price: 0.600000

Category: Office
Avg_Price: 1.100000

Category: Toy
Avg_Price: 9.200000
```

If we were interested in the actual prices that result in each average we can do:

```
$ recsel -p "Category,Price,Avg(Price)" -G Category items.rec
Category: Food
Price: 0.60
Avg_Price: 0.600000

Category: Office
Price: 1.20
Price: 1.00
```

```
Avg_Price: 1.100000

Category: Toy
Price: 12.2
Price: 6.20
Avg_Price: 9.200000
```

11 Queries which Join Records

Suppose you wanted to add the residential address of the people in the '`acquaintances.rec`' file from Section 3.1 [Simple Selections], page 10.

One way to do this is as follows:

```
%type: Dob date

Name: Alfred Nebel
Dob: 20 April 2010
Email: alf@example.com
Address: 42 Abbeter Way, Inprooving, WORCS
Telephone: 01234 5676789

Name: Mandy Nebel
Dob: 21 February 1972
Email: mandy@example.com
Address: 42 Abbeter Way, Inprooving, WORCS
Telephone: 01234 5676789

Name: Bertram Nebel
Dob: 3 January 1966
Email: bert@example.com
Address: 42 Abbeter Way, Inprooving, WORCS
Telephone: 01234 5676789

Name: Charles Spencer
Dob: 4 July 1997
Email: charlie@example.com
Address: 2 Serpe Rise, Little Worning, SURREY
Telephone: 09876 5432109

 Name: Dirk Spencer
Dob: 29 June 1945
Email: dirk@example.com
Address: 2 Serpe Rise, Little Worning, SURREY
Telephone: 09876 5432109

Name: Ernest Wright
Dob: 26 April 1978
Email: ernie@example.com
Address: 1 Wanter Rise, Greater Inncombe, BUCKS
```

This will work fine. However you will notice that there are two addresses where more than one person live (presumably they are members of the same family). This has a number of disadvantages:

- You have to type (or copy) the same information several times.
- Should a family move house, then you would have to update the addresses (and telephone number) of all the family members.
- A typing error in one of the addresses would lead an automatic query to erroneously suggest that the people lived at different addresses.
- It unnecessarily increases the size of the recfile.

11.1 Foreign Keys

A better way would be to separate the addresses and people into different record sets. The first record set might look like this:

```
%rec: Person
%type: Dob date
%type: Abode rec Residence


Name: Alfred Nebel
Dob: 20 April 2010
Email: alf@example.com
Abode: 42AbbeterWay

Name: Mandy Nebel
Dob: 21 February 1972
Email: mandy@example.com
Mobile: 0555 342123
Abode: 42AbbeterWay

Name: Bertram Nebel
Dob: 3 January 1966
Email: bert@example.com
Abode: 42AbbeterWay

Name: Charles Spencer
Dob: 4 July 1997
Email: charlie@example.com
Abode: 2SerpeRise

Name: Dirk Spencer
Dob: 29 June 1945
Email: dirk@example.com
Mobile: 0555 342123
Abode: 2SerpeRise

Name: Ernest Wright
Dob: 26 April 1978
Abode: ChezGrampa
```

and the second (following in the same file), like this:

```
%rec: Residence
%key: Id

Address: 42 Abbeter Way, Inprooving, WORCS
Telephone: 01234 5676789
Id: 42AbbeterWay

Address: 2 Serpe Rise, Little Worning, SURREY
Telephone: 09876 5432109
Id: 2SerpeRise
```

```
Address: 1 Wanter Rise, Greater Inncombe, BUCKS
Id: ChezGrampa
```

Here you can see that there are two record sets *viz:* `Person` and `Residence`. There are six people, but only three residences, because some residences accommodate more than one person. Note also that the `Residence` descriptor has the entry `%key: Id` whilst the `Person` descriptor has `%type: Abode rec Residence`. This is because `Abode` is the foreign key which identifies the residence where a person lives.

We could have declared the `Id` field as `%auto`. This would have had the advantage that we need not manually update it. However, we decided that the `Abode` field values in the `Person` records are better as alphanumeric fields, so that they can contain human readable values. In this way, it is self-evident by reading a `Person` record where that person lives. Yet since the `Id` field is declared using the `%key` special field name, you can be sure that you don't accidentally reuse an existing key.

11.2 Joining Records

The above example has also added a new field to the `Person` record set to contain that person's mobile phone number. Note that the `Telephone` field belongs to the `Residence` record set because that contains the telephone number of the home, whereas `Mobile` belongs to `Person` since mobile telephones are normally used exclusively by one individual.

If we want to look up the name and address of a person in our recfile, we can use `recsel` as before. Because we now have more than one record set in the '`acquaintances.rec`' file, we have to tell `recsel` in which record set we want to look up records. We do this with the `-t` flag as follows:

```
$ recsel -t Person -P Name,Abode acquaintances.rec
Alfred Nebel
42AbbeterWay

Mandy Nebel
42AbbeterWay

Bertram Nebel
42AbbeterWay

Charles Spencer
2SerpeRise

Dirk Spencer
2SerpeRise

Ernest Wright
ChezGrampa
```

This result tells us the names of all the people in the recfile, as well as giving a concise and hopefully effective reminder telling us where they live. However these results would not be useful to someone unacquainted with the individuals. They need a list of names and full addresses. We can use `recsel` to produce such a list:

```
$ recsel -t Person -j Abode acquaintances.rec
Name: Charles Spencer
Dob: 4 July 1997
Email: charlie@example.com
```

```
Abode_Address: 2 Serpe Rise, Little Worning, SURREY
Abode_Telephone: 09876 5432109
Abode_Id: 2SerpeRise

Name: Dirk Spencer
Dob: 29 June 1945
Email: dirk@example.com
Mobile: 0555 342123
Abode_Address: 2 Serpe Rise, Little Worning, SURREY
Abode_Telephone: 09876 5432109
Abode_Id: 2SerpeRise

Name: Ernest Wright
Dob: 26 April 1978
Abode_Address: 1 Wanter Rise, Greater Inncombe, BUCKS
Abode_Id: ChezGrampa
```

The `-t` flag we have seen before. It tells `recsel` that we want to extract records of type `Person`. The `-j` flag is new. It says that we want to perform a *join*. Specifically we want to join the `Person` records according to their `Abode` field.

In the above example, `recsel` displays several field names which do not appear anywhere in the input *e.g.* `Abode_Address`. This is the `Address` field in the record joined by the foreign key `Abode`. In this example probably only the name and address are of interest. The other information such as date of birth is incidental. The foreign key `Abode_Id` is certainly not wanted in the output since it is redundant. As usual, you can use the `-P` or `-p` options to limit the fields which will be displayed. However the full joined field name, if appropriate, must be specified. So the names and addresses without the other information can be retrieved thus:

```
$ recsel -t Person -j Abode -p Name,Abode_Address acquaintances.rec
Name: Charles Spencer
Abode_Address: 2 Serpe Rise, Little Worning, SURREY

Name: Dirk Spencer
Abode_Address: 2 Serpe Rise, Little Worning, SURREY

Name: Ernest Wright
Abode_Address: 1 Wanter Rise, Greater Inncombe, BUCKS
```

12 Auto-Generated Fields

Consider for example a list of articles in stock in a toy store:

```
%rec: Item
%key: Description

Description: 2cm metal soldier WWII
Amount: 2111

Description: Flying Helicopter Indoor Maxi
Amount: 8

...
```

It would be natural to identify the items by their descriptions, but it is also error prone: was it “Flying Helicopter Indoor Maxi” or “Flying Helicopter Maxi Indoor”? Was “Helicopter” in lower case or upper case?

Thus it is quite common in databases to use some kind of numeric “Id” to uniquely identify items like those ones, because numbers are easy to increment and manipulate. So we could add a new numeric `Id` field and use it as the primary key:

```
%rec: Item
%key: Id
%mandatory: Description

Id: 0
Description: 2cm metal soldier WWII
Amount: 2111

Id: 1
Description: Flying Helicopter Indoor Maxi
Amount: 8

...
```

A problem with this approach is that we must be careful to not assign already used ids when we introduce more articles in the database. Other than its uniqueness, it is not important which number is associated with which article.

To ease the management of those Ids database systems use to provide a facility called “auto-counters”. Auto-counters can be implemented in recfiles using the `%auto` directive in the record descriptor. Its usage is:

```
%auto: field1 field2 ... fieldN
```

The list of field names are separated by one or more blank characters. There can be several `%auto` fields in the same record descriptor, the effective list of auto-generated fields being the union of all the entries.

When `recins` inserts a new record in the recfile, it looks for any declared auto field. If any of these fields are not provided explicitly in the command line then `recins` generates them along with the user-provided fields. Such auto fields are generated at the beginning of the new records, in the same order they are found in the `%auto` directives.

For example, consider a ‘`items.rec`’ database with an empty record set:

```
%rec: Item
%key: Id
```

```
%auto: Id
%mandatory: Description
```

If we insert a new record and we do not specify an `Id` then it will be generated automatically by `recins`:

```
$ recins -t Item -f Description -v 'recutils t-shirts' \
         -f Amount -v 200 \
         items.rec
$ cat items.rec
%rec: Item
%key: Id
%auto: Id
%mandatory: Description

Id: 0
Description: recutils t-shirts
Amount: 200
```

The concrete effect of the `%auto` directive depends on the type of the affected field. The following sections document how.

12.1 Counters

If an auto field is of type `integer` or `range` then any newly generated field will use the "next biggest" unused number in the record set.

Consider the toy inventory database introduced above. We could declare the `Id` field to be generated automatically:

```
%rec: Item
%key: Id
%type: Id int
%mandatory: Description
%auto: Id

Id: 0
Description: 2cm metal soldier WWII
Amount: 2111
```

When the next new item is introduced in the database, `recins` will note the `%auto`, and create a new `Id` field for the new record with the next-biggest unused integer, since `Id` is declared to be of type `int`. In this example, the new record would have an Id of `1`. The database can still provide an explicit Id for the new record. In that case the field is not generated automatically.

Note that if no explicit type is defined for an auto generated field then it is assumed to be an integer.

12.2 Unique Identifiers

Universally Unique Identifiers, often abbreviated as UUIDs, can also be auto-generated using recutils. Suppose you maintain a database with events featuring the following record descriptor:

```
%rec: Event
%key: Id
%mandatory: Title Date
```

What would be appropriate to identify each event? We could use an integer and declare it as auto-generated. After adding two events the database would look like this:

```
%rec: Event
%key: Id
%mandatory: Title Date

Id: 0
Title: Team meeting
Date: 12-08-2013

Id: 1
Title: Dave's birthday
Date: 20-12-2013
```

However, suppose that we want to share our events with other people, *i.e.* to send them event records and to incorporate their records into our own database. In this case the `Ids` would collide. A good solution is to use `uuids` and declare them as `auto`:

```
%rec: Event
%key: Id
%type: Id uuid
%mandatory: Title Date

Id: f81d4fae-7dec-11d0-a765-00a0c91e6bf6
Title: Team meeting
Date: 12-08-2013

Id: f81d4fae-dc18-11d0-a765-a01328400a0c
Title: Dave's birthday
Date: 20-12-2013
```

12.3 Time-Stamps

Auto generated dates can be used to implement automatic timestamps. Consider for example a "Transfer" record set registering bank transfers. We want to save a timestamp every time a transfer is done, so we include an `%auto` for the date:

```
%rec: Transfer
%key: Id
%type: Id int
%type: Date date
%auto: Id Date
```

13 Encryption

For ethical or security reasons it is sometimes necessary that information in a recfile should not be readable by unauthorized people. One way to prevent a recfile from being read is to use the security features of the operating system. A more secure way would be to encrypt the entire recfile using a free strong encryption program such as GnuPG. The disadvantage of both these methods is that the entire recfile has to be secured when it may well be the case that only certain data need to be protected.

Recutils offers a way to encrypt specified fields in a record, whilst leaving the rest in clear text.

13.1 Confidential Fields

To specify that a field should be encrypted, use the `%confidential` special field. This special field declares a set of fields as *confidential*, meaning they contain secret data such as passwords or personal information. Its usage is:

```
%confidential: field1 field2 ... fieldN
```

The field names are separated by one or more blank characters. There can be several `%confidential` fields in the same record descriptor, the effective list of confidential fields being the union of all the entries.

Declaring a field as confidential indicates that its contents must not be stored in plain text, but encrypted with a password-based mechanism. When the information is retrieved from the database the confidential fields are unencrypted if the correct password is provided. Likewise, when information is inserted in the database the confidential fields are encrypted with some given password.

For example, consider a database of users of some service. For each user we want to store a name, a login name, an email address and a password. All this information is public with the obvious exception of the password. Thus we declare the `Password` field as confidential in the corresponding record descriptor:

```
%rec: Account
%type: Name line
%type: Login line
%type: Email email
%confidential: Password
```

The rec format does not impose the usage of a specific encryption algorithm, but requires that:

- The algorithm must be password-based.
- The value of any encrypted field shall begin with the string '`encrypted-`' followed by the encrypted data.
- The encrypted data must be encoded in some ASCII encoding such as base64.

The above rules assure that it is possible to determine whether a given field is encrypted. For example, the following is an excerpt from the account database described above. It contains an entry with the password encrypted and another with the password unencrypted:

```
Name: Mr. Foo
Login: foo
Email: foo@foo.com
Password: encrypted-AAABBBCCDDDEEEFFF

Name: Mr. Bar
```

```
Login: bar
Email: bar@bar.com
Password: secret
```

Unencrypted confidential fields are a data integrity error, and utilities like `recfix` will report it. The same utility can be used to “fix” the database by massively encrypting any unencrypted field.

Nothing prevents the usage of several passwords in the same database. This allows the establishment of several level of securities or security profiles. For example, we may want to store different passwords for different online services:

```
%rec: Account
%confidential: WebPassword ShellPassword
```

We could then encrypt WebPassword entries using a password shared among all the webmasters, and the ShellPassword entries with a more restricted password available only to the administrator of the machine.

Note that since the utilities only accept to specify one password at a time different passwords cannot be specified at decryption time. This means that in the example above the administrator would need to run `recsel` twice in order to decrypt all the encrypted data in the recfile.

The GNU recutils fully support encrypted fields. See the documentation for `recsel`, `recins` and `recfix` for details on how to operate on files containing confidential fields.

13.2 Encrypting Files

`recins` allows the insertion of encrypted fields in a database. When the ‘`-s`’ (‘`--password`’) command line option is specified in the command line any field declared as confidential in the record descriptor will get encrypted using the given passphrase. If the command is executed interactively and ‘`-s`’ is not used then the user is asked to provide a password using the terminal. For example, the invocation:

```
$ recins -t Account -s mypassword -f Login -v foo -f Password  \
  -v secret accounts.rec
```

will encrypt the value of the `Password` field with `mypassword` as long as the field is declared as confidential. (see Section 13.1 [Confidential Fields], page 52 for details on confidential fields).

`recins` will issue a warning if a confidential field is inserted in the database but no password was provided to encrypt it. This is to avoid having unencrypted sensitive data in the recfiles.

13.3 Decrypting Data

The contents of confidential fields can be read using the ‘`-s`’ (‘`--password`’) command line option to `recsel`. When used, any selected record containing encrypted fields will try to decrypt them with the given password. If the operation succeeds then the output will include the unencrypted data. Otherwise the ASCII-encoded encrypted data will be emitted.

If `recsel` is invoked interactively and no password is specified with ‘`-s`’, the user will be asked for a password in case one is needed. No echo of the password will appear in the screen. The provided password will be used to decrypt all confidential fields as if it was specified with ‘`-s`’.

For example, consider the following database storing information about the user accounts of some online service. Each entry stores a login, a full name, email and a password. The password is declared as confidential:

```
%rec: Account
%key: Login
%confidential: Password
```

```
Login: foo
Name: Mr. Foo
Email: foo@foo.com
Password: encrypted-AAABBBCCCDDD

Login: bar
Name: Ms. Bar
Email: bar@bar.org
Password: encrypted-XXXYYYZZZUUU
```

If we use `recsel` to get a list of records of type `Account` without specifying a password, or if the wrong password was specified in interactive mode, then we would get the following output with the encrypted values:

```
$ cat accounts.rec | recsel -t Account -p Login,Password
Login: foo
Password: encrypted-AAABBBCCCDDD

Login: bar
Password: encrypted-XXXYYYZZZUUU
```

If we specify a password and both entries were encrypted using that password, we would get the unencrypted values:

```
$ recsel -t Account -s secret -p Login,Password accounts.rec
Login: foo
Password: foosecret

Login: bar
Password: barsecret
```

As mentioned above, a confidential field may be encrypted with different passwords in different records (see Section 13.1 [Confidential Fields], page 52). For example, we may have an entry in our database with data about the account of the administrator of the online service. In that case we might want to store the password associated with that account using a different password than that for users. In that case the output of the last command would have been:

```
$ recsel -t Account -s secret -p Login,Password accounts.rec
Login: foo
Password: foosecret

Login: bar
Password: barsecret

Login: admin
Password: encrypted-TTTVVVBBBNNN
```

We would need to invoke `recsel` with the password used to encrypt the admin entry in order to read it back unencrypted.

14 Generating Reports

Having a list of names and addresses, one might want to use this list to address envelopes (say, to send annual greeting cards). Since addresses are normally written on several lines, it would be appropriate then to split the `Address` field values across multiple lines as described in Section 2.1 [Fields], page 4. Suitable text can now be obtained thus:

```
$ recsel -t Person -j Abode -P Name,Abode_Address acquaintances.rec
Charles Spencer
2 Serpe Rise,
Little Worning,
SURREY

Dirk Spencer
2 Serpe Rise,
Little Worning,
SURREY

Ernest Wright
1 Wanter Rise,
Greater Inncombe,
BUCKS
```

A business enterprise might want to go one step further and generate letters (such as an advertisement or a recall notice) to customers. Since `recsel` merely selects records and fields from record sets, on its own it cannot do this; so there is another command designed for this purpose, called `recfmt`. This command uses a *template* which defines the general form of the desired output. A letter template might look as follows:

```
{{Name}}
{{Abode_Address}}

Dear {{Name}},

     Re: Special offer for January

We are delighted to be able to offer you a 95% discount on all car and
truck hire contracts between 1 January and 2 February.  Please call us
to take advantage of this offer.

Yours sincerely,


Karen van Rental (CEO)
^L
```

It is best to place such a template into a file, so that you can edit it as you wish. Notice the instances of double braces enclosing a field name, *e.g.* `{{Name}}`. These are called *slots* and indicate places where the respective field's value should be placed. Let's assume this template is in a file called '`offer.templ`'. We can then pipe the output from `recsel` into `recfmt` in order as follows:

```
$ recsel -t Person -j Abode acquaintances.rec | recfmt -f offer.templ
Charles Spencer
2 Serpe Rise,
Little Worning,
SURREY

Dear Charles Spencer,

     Re: Special offer for January

We are delighted to be able to offer you a 95% discount on all car and
.
.
.
```

For each record that `recsel` selects, one copy of '`offer.templ`' will be generated. Each slot will be replaced with the field value corresponding to the field name in the slot.

14.1 Templates

A recfmt template is a text string that may contain *template slots*. Those slots are substituted in the template using the information of a given record. Any text that is not within a slot is copied literally to the output.

Slots are written surrounded by double curly braces, like:

```
{{...}}
```

Slots contain selection expressions, that are executed every time the template is applied to a record. The slot is then replaced by the string representation of the value returned by the expression.

For example, consider the following template:

```
Task {{Id}}: {{Summary}}
------------------------
{{Description}}
--
Created at {{CreatedAt}}
```

When applied to the following record:

```
Id: 123
Summary: Fix recfmt.
CreatedAt: 12 December 2010
Description:
+ The recfmt tool shall be fixed, because right
+ now it is leaking 200 megabytes per processed record.
```

The result is:

```
Task 123: Fix recfmt.
------------------------
The recfmt tool shall be fixed, because right
now it is leaking 200 megabytes per processed record.
--
Created at 12 December 2010
```

You can use any selection expression in the slots, including conditionals and string concatenation.

15 Interoperability

Included in the recutils package are a number of utilities to assist in the creation of recfiles using data which already exists in other formats, and for exporting data from recfiles so that it can be used in other applications.

15.1 CSV Files

Many applications are able to read and write files containing so-called “comma separated values”. Such files generally contain tabular data where the columns are separated by commas and the rows by line feed and/or carriage return characters. Although record sets are not tables, tables can be easily emulated using records having the same fields in the same order. For example:

```
a: value
b: value
c: value

a: value
b: value
c: value

...
```

In several respects records are more flexible than tables:

– Fields can appear in a different order in several records.
– There can be several fields with the same name in a single record.
– Records can differ in the number of fields.

It is evident that records, such as those in recfiles, are a more general structure than comma separated values. This means that when converting from csv files to recfiles, certain decisions need to be made. The `rec2csv` utility (see Section 17.9 [Invoking rec2csv], page 70) implements an algorithm to deal with this problem and generate a table that the user expects.

The algorithm works as follows:

1. The utility first scans the specified record set, building a list with the names that will become the table header.
2. For each field, a header is added with the form:

   ```
   FIELDNAME[_n]
   ```

   where *n* is a number in the range `2..inf` and is the “index” of the field in its containing record plus one. For example, consider the following record set:

   ```
   a: a1
   b: b11
   b: b12
   c: c1

   a: a2
   b: b2
   d: d2
   ```

   The corresponding list of headers being:

   ```
   a b b_2 c a b d
   ```

3. Then duplicates are removed:

   ```
   a b b_2 c d
   ```

4. The resulting list of headers is then used to build the table in the generated csv file.

   In the above example the result would be

   ```
   "a","b","b_2","c","d"
   "a1","b11","b12","c1",
   "a2","b2",,,"d2"
   ```

   As shown, missing fields are implemented as empty columns in the generated csv.

15.2 Importing MDB Files

Access files (*mdb files*) are collections of several relations, also known as tables. Tables can be either *user tables* storing user data, or *system tables* storing information such as forms, queries or the relationships between the tables.

It is possible to get a listing with the names of all tables stored in a mdb file by calling `mdb2rec` in the following way:

```
$ mdb2rec -l sales.mdb
Customers
Products
Orders
```

So '`sales.mdb`' stores user information in the tables Customers, Products and Orders. If we want to include system tables in the listing we can use the '`-s`' command line option:

```
$ mdb2rec -s -l sales.mdb
MSysObjects
MSysACEs
MSysQueries
MSysRelationships
Customers
Products
Orders
```

The tables with names starting with `MSys` are system tables. The data stored in those tables is either not relevant to the recutils user (used by the Access program to create forms and the like) or is used in an indirect way by `mdb2rec` (such as the information from MSysRelationships).

Let's read some data from the '`mdb`' file. We can get the relation of Products in rec format:

```
$ mdb2rec sales.mdb Products
%rec: Products
%type: ProductID int
%type: ProductName size 80
%type: Discontinued bool

ProductID: 1
ProductName: GNU generation T-shirt
Discontinued: 0

...
```

A *record descriptor* is created for the record set containing the generated records, called Products. As seen in the example, `mdb2rec` is able to generate type information for the fields. The list of customers is similar:

```
$ mdb2rec sales.mdb Customers
%rec: Customers
%type: CustomerID size 4
```

```
%type: CompanyName size 80
%type: ContactName size 60

CustomerID: GSOFT
CompanyName: GNU Soft
ContactName: Jose E. Marchesi

...
```

If no table is specified in the invocation to `mdb2rec` all the tables in the file are processed, with the exception of the system tables, which requires '`-s`' to be used:

```
$ mdb2rec sales.mdb
%rec: Products
...

%rec: Customers
...

%rec: Orders
...
```

16 Bash Builtins

The command-line utilities described in Chapter 17 [Invoking the Utilities], page 62 are designed to be used interactively in the shell. Together, and often combined with the standard shell utilities, they provide a quite complete user interface. However, the user's experience can be greatly improved by a closer integration between the recutils and the shell. The following sections describe several extensions for `bash`, the GNU shell (see Section "Top" in *The GNU Bourne-Again SHell*). These extensions make the shell "aware" of the recutils.

As with any bash built-in, help is available in the command line using the `help` command. For example:

```
$ help readrec
```

If you installed recutils using a binary package in a GNU/Linux distribution, odds are that the built-in commands described in this chapter are already available to you. Otherwise (you get a "command not found" or similar error) you may have to register the built-in commands with your bash. This is very easy using the `enable` bash command. The registering command for readrec would be:

```
$ enable -f readrec.so readrec
```

Note however that some systems require the full path to '`readrec.so`' in order for this command to work.

16.1 readrec

The bash built-in `read`, when invoked with no options, consumes one line from standard input and makes it available in the predefined `REPLY` environment variable, or any other variable whose name is passed as an argument. This allows processing data structured in lines in a quite natural way. For example, the following program prints the third field of each line, with fields separated by commas, until standard input is exhausted:

```
# Process one line at a time.
while read
do
  echo "The third field is " `echo $REPLY | cut -d, -f 2`
done
```

However, `read` is not very useful when it comes to processing recutils records in the shell. Even though it is possible to customize the character used by `read` to split the input into records, we would need to ignore the empty records in the likely case of more than one empty line separating records. Also, we would need to use `recsel` to access to the record fields. Too complicated!

Thus, the `readrec` bash built-in is similar to `read` with the difference that it reads records instead of lines. It also "exports" the contents of the record to the user as the values of several environment variables:

- `REPLY_REC` is set to the record read from standard input.
- A set of variables `FIELD` named after each field found in the record are set to the (decoded) value of the fields found in the input record. When several fields with the same name are found in the input record then a bash array is created.

Consider for example the following simple database containing contacts information:

```
Name: Mr. Foo
Email: foo@bar.com
Email: bar@baz.net
Checked: no
```

```
Name: Mr. Bar
Email: bar@foo.com
Telephone: 999666000
Checked: yes
```

We would like to write some shell code to send an email to all the contacts, but only if the contact has not been checked before, *i.e.* the `Checked` field contains `no`. The following code snippet would do the job nicely using `readrec`:

```
recsel contacts.rec | while readrec
do
   if [ $Checked = "no" ]
   then
      mail -s "You are being checked." ${Email[0]} < email.txt
      recset -e "Email = '$Email'" -f Checked -S yes contacts.rec
      sleep 1
   fi
done
```

Note the usage of the bash array when accessing the primary email address of each contact. Note also that we update each contact to figure as “checked”, using `recset`, so she won’t get pestered again the next time the script is run.

17 Invoking the Utilities

Certain options are available in all of these programs. Rather than writing identical descriptions for each of the programs, they are listed here.

'--version'
: Print the version number, then exit successfully.

'--help'
: Print a help message, then exit successfully.

'--'
: Delimit the option list. Later arguments, if any, are treated as operands even if they begin with '-'. For example, `recsel -- -p` reads from the file named '-p'.

17.1 Invoking recinf

`recinf` reads the given rec files (or the data from standard input if no file is specified) and prints a summary of the record types contained in the input.

Synopsis:

```
recinf [option]... [file]...
```

The default behavior is to emit a line per record type in the input containing its name and the number of records of that type:

```
$ recinf hackers.rec tasks.rec
25 Hacker
102 Task
```

If the input contains anonymous records, *i.e.* records that are before the first record descriptor, the corresponding output line won't have a type name:

```
$ recinf data.rec
10
```

In addition to the common options described earlier the program accepts the following options.

'-t *type*'
'--type=*type*'
: Select records of a given type only.

'-d'
'--descriptor'
: Print all the record descriptors present in the file.

'-n'
'--names-only'
: Output just the names of the record types found in the input. If the input contains only anonymous records then output nothing.

'-S'
'--print-sexps'
: Print the data in the form of sexps (Lisp expressions) instead of rec format. This option can be useful for, of course, Lisp programs.

17.2 Invoking recsel

`recsel` reads the given rec files (or the data in the standard input if no file is specified) and prints out records (or part of records) based upon some criteria specified by the user.

`recsel` searches rec files for records satisfying certain criteria. Synopsis:

```
     recsel [option]... \
            [-n indexes | -e record_expr | -q str | -m num] \
            [-c | (-p|-P|-R) field_expr] \
            [file]...
```

If no *file* is specified then the command acts like a filter, getting the data from standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

The following *global options* are available.

'`-i`'
'`--case-insensitive`'
: Make string matching case-insensitive in selection expressions.

'`-C`'

'`--collapse`'
: Do not section the result in records with newlines.

'`-d`'
'`--include-descriptors`'
: Print record descriptors along with the matched records.

'`-s` *secret*'
'`--password=`*secret*'
: Try to decrypt confidential fields with the given password.

'`-S`'
'`--sort=`*fields*'
: Sort the output by the comma-separated list of field names, *fields*. This option takes precedence over any sorting criteria specified in the corresponding record descriptor with `%sort`.

'`-U`'
'`--uniq`' Remove duplicated fields in the output records. Fields are duplicated if they have the same field name and the same value.

'`-G`'
'`--group-by=`*fields*'
: Group the output records by the provided comma-separated list of *fields*. Grouping is performed before sorting.

The *selection options* are used to select a subset of the records in the input.

'`-n` *indexes*'
'`--number=`*indexes*'
: Match the records occupying the given positions in its record set. *indexes* must be a comma-separated list of numbers or ranges, with ranges being two numbers separated with dashes. For example, the following list denotes the first, the third, the fourth and all records up to the tenth: '`-n 0,2,4-9`'.

'`-e` *expr*'
'`--expression=`*expr*'
: A record selection expression (see Section 3.5 [Selection Expressions], page 14). Only the records matched by the expression will be taken into account to compute the output.

'`-q` *str*'
'`--quick=`*str*'
: Select records having a field whose value contains the substring *str*.

'`-m` *num*'
'`--random=`*num*'
: Select *num* random records. If *num* is zero then select all the records.

'`-t` *type*'
'`--type=`*type*'
: Select records of a given type only.

'`-j` *field*'
'`--join=`*field*'
: Perform an inner join of the record set selected by '`-t`' and the record set for which *field* is a foreign key. *field* must be a field declared with type `rec` and thus must be a foreign key. If a join is performed then any selection expression and field expression operate on the joined record sets.

The *output options* are used to determine what information about the selected records to display to the user, and how to display it.

'`-p` *name_list*'
'`--print=`*name_list*'
: List of fields to print for each record. *name_list* is a list of field names separated by commas. For example:

```
-p Name,Email
```

means to print the Name and the Email of every matching record, both the field names and values.

If this option is not specified then all the fields of the matching records are printed to standard output.

'`-P` *name_list*'
'`--print-values=`*name_list*'
: Same as '`-p`', but print only the values of the selected fields.

'`-R` *name_list*'
'`--print-row=`*name_list*'
: Same as '`-P`', but print the values separated by single spaces instead of newlines.

'`-c`'
'`--count`' If this option is specified then `recsel` will print the number of matching records instead of the records themselves. This option is incompatible with '`-p`', '`-P`' and '`-R`'.

This *special option* is available to ease the communication between the recutils and other programs, namely Lisp interpreters. This option is not intended to be used by human operators.

'`--print-sexps`'
: Print the data using sexps instead of rec format.

17.3 Invoking recins

`recins` adds new records to a rec file or to rec data read from standard input. Synopsis:

```
recins [option]... [-t type] \
       [-n indexes | -e record_expr | -q str | -m num] \
       [( -f str -v str]|[-r recdata )]... \
               [file]
```

The new record to be inserted by the command is constructed by using pairs of '`-f`' and '`-v`' options, or '`-r`'. Each pair defines a field. The order of the parameters is significant.

If no *file* is specified then the command acts like a filter, getting the data from standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'`-t`'
'`--type=`*expr*'
: The type of the new record. If there is a record set in the input data matching this type then the new record is added there. Otherwise a new record set is created. If this parameter is not specified then the new record is anonymous.

'`-f`'
'`--field=`*name*'
: Declares the name of a field. This option must be followed by a '`-v`'.

'`-v`'
'`--value=`*value*'
: The value of the field being defined.

'`-r`'
'`--record=`*value*'
: Add the fields of the record in *value*. This option can be intermixed with '`-f ... -v`' pairs.

'`-s`'
'`--password`'
: Encrypt confidential fields with the given password.

'`--no-external`'
: Don't use external record descriptors.

'`--verbose`'
: Be verbose when reporting integrity problems.

'`--no-auto`'
: Don't generate *auto* fields. See Chapter 12 [Auto-Generated Fields], page 49.

Record selection arguments are supported too. If they are used then `recins` uses "replacement mode": instead of appending the new record, matched records are replaced by copies of the provided record. The selection arguments are the same as in `recsel`:

'`-n` *indexes*'
'`--number=`*indexes*'
: Match the records occupying the given positions in its record set. *indexes* must be a comma-separated list of numbers or ranges, the ranges being two numbers separated with dashes. For example, the following list denotes the first, the third, the fourth and all records up to the tenth: `-n 0,2,4-9`.

'`-e` *record_expr*'
'`--expression=`*expr*'
: A record selection expression (see Section 3.5 [Selection Expressions], page 14). Matching records will get replaced.

'`-q` *str*'
'`--quick=`*str*'
: Remove records having a field whose value contains the substring *str*.

'-m *num*'
'--random=*num*'
: Select *num* random records. If *num* is zero then all records are selected, *i.e.* no replace mode is activated.

'-i'
'--case-insensitive'
: Make strings case-insensitive in selection expressions.

'--force' Insert the requested record even in potentially dangerous situations, such as when the data integrity of the database is compromised.

17.4 Invoking recdel

`recdel` removes records from a rec file, or from rec data read from standard input. Synopsis:

```
recdel [OPTIONS]... [-t type] \
       [-n indexes | -e record_expr | -q str | -m num] \
       [file]
```

If no *file* is specified then the command acts like a filter, getting the data from standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'-t'
'--type=*expr*'
: Remove records of the given type. If this parameter is not specified then records of any type will be removed.

'-n *indexes*'
'--number=*indexes*'
: Match the records occupying the given positions in its record set. *indexes* must be a comma-separated list of numbers or ranges, the ranges being two numbers separated with dashes. For example, the following list denotes the first, the third, the fourth and all records up to the tenth: `-n 0,2,4-9`.

'-e *record_expr*'
'--expression=*expr*'
: A record selection expression (see Section 3.5 [Selection Expressions], page 14). Only the records matched by the expression will be removed from the file.

'-q *str*'
'--quick=*str*'
: Remove records having a field whose value contains the substring *str*.

'-m *num*'
'--random=*num*'
: Remove *num* random records. If *num* is zero then remove all the records.

'-c'
'--comment'
: Comment the matching records out instead of removing them.

'--force' Delete even in potentially dangerous situations, such as a request to delete all the records of some type.

'--no-external'
: Don't use external record descriptors.

'`-i`'
'`--case-insensitive`'
: Make strings case-insensitive in selection expressions.

'`--verbose`'
: Be verbose when reporting integrity problems.

17.5 Invoking recset

`recset` manipulates the fields of records in a rec file, or rec data read from standard input. Synopsis:

```
recset [option]... [file]...
```

If no *file* is specified then the command acts like a filter, getting the data from standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

Record selection options:

'`-i`'
'`--case-insensitive`'
: Make strings case-insensitive in selection expressions.

'`-t`'
'`--type=`*expr*'
: Operate on the records of the given type. If this parameter is not specified then records of any type will be affected.

'`-n` *indexes*'
'`--number=`*indexes*'
: Operate on the records occupying the given positions in its record set. *indexes* must be a comma-separated list of numbers or ranges, the ranges being two numbers separated with dashes. For example, the following list denotes the first, the third, the fourth and all records up to the tenth: `-n 0,2,4-9`.

'`-e` *expr*'
'`--expression=`*expr*'
: A record selection expression (see Section 3.5 [Selection Expressions], page 14). Only the records matched by the expression will be processed.

'`-q` *str*'
'`--quick=`*str*'
: Operate on records having a field whose value contains the substring *str*.

'`-m` *num*'
'`--random=`*num*'
: Operate on *num* random records. If *num* is zero then operate on all the records.

Field selection options:

'`-f`'
'`--fields=`*FEX*'
: Field selection expression (see Section 3.6 [Field Expressions], page 18) to select the fields to operate.

Actions:

'`-s`'
'`--set=`*value*'
: Set the value of the selected fields to *value*.

'`-a`'
'`--add=`*`value`*'
: Add a new field to the selected record with value *value*.

'`-S`'
'`--set-add=`*`value`*'
: Set the value of the selected fields to *value*. If some of the fields don't exist in a record, append it with the specified value.

'`-r`'
'`--rename=`*`value`*'
: Rename a field; *value* must be a valid field name. The field expression associated with this action must contain a single field name and an optional subscript. If an entire record set is selected then the field is renamed in the record descriptor as well.

'`-d`'
'`--delete`'
: Delete the selected fields in the selected records.

'`-c`'
'`--comment`'
: Comment out the selected fields in the selected records.

'`--no-external`'
: Don't use external record descriptors.

'`--verbose`'
: Be verbose when reporting integrity problems.

'`--force`'
: Perform the requested operation even in potentially dangerous situations, or when the integrity of the data stored in the file is affected.

17.6 Invoking recfix

`recfix` checks and fixes rec files. Synopsis:

```
recfix [option]... [operation] [op_option]... [file]
```

If no *file* is specified then the command acts like a filter, getting the data from standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following global options.

'`--no-external`'
: Don't use external record descriptors.

The effect of running `recfix` depends on the operation it performs. The operation mode is selected by using one of the following options.

'`--check`'
: Check the integrity of the database contained in the file, printing diagnostics messages in case something is not right. This is the default operation.

'`--sort`'
: Perform a physical sort of all the records contained in the file (or standard input) after checking for its integrity. The sorting criteria are provided by the `%sort` special field, if any. If there is an integrity failure the sorting is not performed.

  This is a destructive operation.

'`--decrypt`'
'`--encrypt`'
: Decrypt (encrypt) all the (non-)encrypted fields in the database which are marked as confidential. This operation requires a password. If no password is specified with

'-s' and the program is run in a terminal, a prompt is given to get the password from the user.

If encryption is performed on a file having encrypted fields, the operation will fail unless '--force' is used.

These are destructive operations.

'--auto'
: Insert auto-generated fields as appropriate in the records which are missing them.

  This is a destructive operation.

As described above, some operations make use of these additional options:

'-s *secret*'
'--password=*secret*'
: Password used to encrypt or decrypt fields.

'--force'
: Force potentially dangerous operations.

17.7 Invoking recfmt

`recfmt` formats records using templates. Synopsis:

```
recfmt [option]... [template]
```

This program always works as a filter, getting the data from the standard input and writing the result to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'-f'
'--filename=*PATH*'
: Read the template from the file in *PATH* instead of the command line.

17.8 Invoking csv2rec

`csv2rec` reads the given comma-separated-values file (or the data from standard input if no file is specified) and prints out the converted rec data, if possible. Synopsis:

```
csv2rec [option]... [csv_file]
```

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'-t *type*'
'--type=*type*'
: Type of the converted records. If no type is specified then no type is used.

'-s'
'--strict'
: Be strict parsing the csv file.

'-e'
'--omit-empty'
: Omit empty fields.

17.9 Invoking rec2csv

`rec2csv` reads the given rec files (or the data in the standard input if no file is specified) and prints out the converted comma-separated-values. Synopsis:

```
rec2csv [option]... [rec_file]...
```

The rec data can be read from files specified in the command line, or from standard input. The program writes the converted data to standard output.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'`-t` *type*'
'`--type=`*type*'
: Type of the records to convert. If no type is specified then the default records (with no name) are converted.

'`-S`'
'`--sort=`*fields*'
: Sort the output by the comma-separated list of field names *fields*. This option has precedence to whatever sorting criteria are specified in the corresponding record descriptor with `%sort`.

'`-d`'
'`--delim=`*char*'
: Use *char* as the delimiter character separating fields in the output. Defaults to `,`.

17.10 Invoking mdb2rec

`mdb2rec` reads the given mdb file and prints out the converted rec data, if possible. Synopsis:

```
mdb2rec [option]... mdb_file [table]
```

All the tables contained in the mdb file are exported unless a table is specified in the command line.

In addition to the common options described earlier (see [Common Options], page 62) the program accepts the following options.

'`-s`'
'`--system-tables`'
: Include system tables in the output.

'`-l`'
'`--list-tables`'
: Dump a list of the table names contained in the mdb file, one per line.

'`-e`'
'`--keep-empty-fields`'
: Don't prune empty fields in the rec output.

18 Using ob-rec.el

ob-rec.el allows you to use Recutils as a language in org-mode source blocks.

18.1 Setup

Recutils should install the necessary files where emacs can see them.

In your .emacs you may need to add:

```
(require 'ob-rec)
```

You will need to add "rec" to your list of 'org-babel-load-languages' like below:

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((rec . t)))
```

18.2 Usage

To your org file, add a src code block like:

```
#+BEGIN_SRC rec :data books.rec
  Location = 'loaned'
#+END_SRC
```

This performs the equivalent of the command:

```
 $ recsel -e "Location = 'loaned'" books.rec
```

It will produce a result like:

```
#+RESULTS:
| Title                | Author          | Date            | Location |
|----------------------+-----------------+-----------------+----------|
| The Colour of Magic  | Terry Pratchett | 4/20/01 11:15pm | loaned   |
```

18.3 Header Arguments

'`:data`' The recfile you would like to query. Can be a relative path. Spaces in the filename or path need to be escaped with a backslash (for example, file\ name.rec). This is the only required header argument.

'`:results`'
: If this list contains "scalar", "html", "code" or "verbatim" then the output will look the same as if called from the command line and it will not be put into an org table.

'`:type`' Only returns this type of record. Corresponds to the -t argument. Accepts only one argument.

'`:fields`' Comma-separated list of fields to print. Corresponds to the -p argument.

'`:sort`' Comma-separated list of fields by which to sort records. Corresponds to the -S argument.

'`:groupby`'
: Comma-separated list of fields by which to group records. If the records grouped together share fields in common, these will be in separate columns with a "_N" appended. Corresponds to the -G argument.

'`:join`' Field on which to join records from one record set to another. Please see blah for more on how joins work. Corresponds to the -j argument.

18.4 Warnings

1. Output may be unpredictable if fields contain newlines, as would be the case for a multi-line field. This appears to be a limitation in org-mode's 'org-table-convert-region' function.

19 Regular Expressions

The character '`.`' matches any single character except the null character.

'`+`'	match one or more occurrences of the previous atom or regexp.
'`?`'	match zero or one occurrences of the previous atom or regexp.
'`\+`'	matches a '`+`'
'`\?`'	matches a '`?`'.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example '`[z-a]`', are invalid. Within square brackets, '`\`' is taken literally. Character classes are supported; for example '`[[:digit:]]`' matches a single decimal digit.

GNU extensions are supported:

'`\w`'	matches a character within a word
'`\W`'	matches a character which is not within a word
'`\<`'	matches the beginning of a word
'`\>`'	matches the end of a word
'`\b`'	matches a word boundary
'`\B`'	matches characters which are not a word boundary
'``\` ``'	matches the beginning of the whole input
'`\'`'	matches the end of the whole input

Grouping is performed with parentheses '`()`'. An unmatched '`)`' matches just itself. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example, '`\2`' matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis '`(`'.

The alternation operator is '`|`'.

The characters '`^`' and '`$`' always represent the beginning and end of a string respectively, except within square brackets. Within brackets, an initial '`^`' inverts the character class being matched.

'`*`', '`+`' and '`?`' are special at any point in a regular expression except the following places, where they are not allowed:

1. At the beginning of a regular expression
2. After an open-group, '`(`'
3. After the alternation operator, '`|`'

Intervals are specified by '`{`' and '`}`'. Invalid intervals such as '`a{1z`' are not accepted.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to sub-expressions within groups.

20 Date input formats

First, a quote:

> Our units of temporal measurement, from seconds on up to months, are so complicated, asymmetrical and disjunctive so as to make coherent mental reckoning in time all but impossible. Indeed, had some tyrannical god contrived to enslave our minds to time, to make it all but impossible for us to escape subjection to sodden routines and unpleasant surprises, he could hardly have done better than handing down our present system. It is like a set of trapezoidal building blocks, with no vertical or horizontal surfaces, like a language in which the simplest thought demands ornate constructions, useless particles and lengthy circumlocutions. Unlike the more successful patterns of language and science, which enable us to face experience boldly or at least level-headedly, our system of temporal calculation silently and persistently encourages our terror of time.
>
> . . . It is as though architects had to measure length in feet, width in meters and height in ells; as though basic instruction manuals demanded a knowledge of five different languages. It is no wonder then that we often look into our own immediate past or future, last Tuesday or a week from Sunday, with feelings of helpless confusion. . . .
>
> —Robert Grudin, *Time and the Art of Living*.

This section describes the textual date representations that GNU programs accept. These are the strings you, as a user, can supply as arguments to the various programs. The C interface (via the `parse_datetime` function) is not described here.

20.1 General date syntax

A *date* is a string, possibly empty, containing many items separated by whitespace. The whitespace may be omitted when no ambiguity arises. The empty string means the beginning of today (i.e., midnight). Order of the items is immaterial. A date string may contain many flavors of items:

- calendar date items
- time of day items
- time zone items
- combined date and time of day items
- day of the week items
- relative items
- pure numbers.

We describe each of these item types in turn, below.

A few ordinal numbers may be written out in words in some contexts. This is most useful for specifying day of the week items or relative items (see below). Among the most commonly used ordinal numbers, the word '`last`' stands for -1, '`this`' stands for 0, and '`first`' and '`next`' both stand for 1. Because the word '`second`' stands for the unit of time there is no way to write the ordinal number 2, but for convenience '`third`' stands for 3, '`fourth`' for 4, '`fifth`' for 5, '`sixth`' for 6, '`seventh`' for 7, '`eighth`' for 8, '`ninth`' for 9, '`tenth`' for 10, '`eleventh`' for 11 and '`twelfth`' for 12.

When a month is written this way, it is still considered to be written numerically, instead of being "spelled in full"; this changes the allowed strings.

In the current implementation, only English is supported for words and abbreviations like '`AM`', '`DST`', '`EST`', '`first`', '`January`', '`Sunday`', '`tomorrow`', and '`year`'.

The output of the `date` command is not always acceptable as a date string, not only because of the language problem, but also because there is no standard meaning for time zone items like '`IST`'. When using `date` to generate a date string intended to be parsed later, specify a date format that is independent of language and that does not use time zone items other than '`UTC`' and '`Z`'. Here are some ways to do this:

```
$ LC_ALL=C TZ=UTC0 date
Tue Jul 21 23:00:37 UTC 2020
$ TZ=UTC0 date +'%Y-%m-%d %H:%M:%SZ'
2020-07-21 23:00:37Z
$ date --rfc-3339=ns  # --rfc-3339 is a GNU extension.
2020-07-21 19:00:37.692722128-04:00
$ date --rfc-2822  # a GNU extension
Tue, 21 Jul 2020 19:00:37 -0400
$ date +'%Y-%m-%d %H:%M:%S %z'  # %z is a GNU extension.
2020-07-21 19:00:37 -0400
$ date +'@%s.%N'  # %s and %N are GNU extensions.
@1595372437.692722128
```

Alphabetic case is completely ignored in dates. Comments may be introduced between round parentheses, as long as included parentheses are properly nested. Hyphens not followed by a digit are currently ignored. Leading zeros on numbers are ignored.

Invalid dates like '`2019-02-29`' or times like '`24:00`' are rejected. In the typical case of a host that does not support leap seconds, a time like '`23:59:60`' is rejected even if it corresponds to a valid leap second.

20.2 Calendar date items

A *calendar date item* specifies a day of the year. It is specified differently, depending on whether the month is specified numerically or literally. All these strings specify the same calendar date:

```
2020-07-20     # ISO 8601.
20-7-20        # Assume 19xx for 69 through 99,
               # 20xx for 00 through 68 (not recommended).
7/20/2020      # Common U.S. writing.
20 July 2020
20 Jul 2020    # Three-letter abbreviations always allowed.
Jul 20, 2020
20-jul-2020
20jul2020
```

The year can also be omitted. In this case, the last specified year is used, or the current year if none. For example:

```
7/20
jul 20
```

Here are the rules.

For numeric months, the ISO 8601 format '*`year-month-day`*' is allowed, where *year* is any positive number, *month* is a number between 01 and 12, and *day* is a number between 01 and 31. A leading zero must be present if a number is less than ten. If *year* is 68 or smaller, then 2000 is added to it; otherwise, if *year* is less than 100, then 1900 is added to it. The construct '*`month/day/year`*', popular in the United States, is accepted. Also '*`month/day`*', omitting the year.

Literal months may be spelled out in full: '`January`', '`February`', '`March`', '`April`', '`May`', '`June`', '`July`', '`August`', '`September`', '`October`', '`November`' or '`December`'. Literal months may be abbreviated to their first three letters, possibly followed by an abbreviating dot. It is also permitted to write '`Sept`' instead of '`September`'.

When months are written literally, the calendar date may be given as any of the following:

```
day month year
day month
month day year
day-month-year
```

Or, omitting the year:

```
month day
```

20.3 Time of day items

A *time of day item* in date strings specifies the time on a given day. Here are some examples, all of which represent the same time:

```
20:02:00.000000
20:02
8:02pm
20:02-0500      # In EST (U.S. Eastern Standard Time).
```

More generally, the time of day may be given as '`hour:minute:second`', where *hour* is a number between 0 and 23, *minute* is a number between 0 and 59, and *second* is a number between 0 and 59 possibly followed by '`.`' or '`,`' and a fraction containing one or more digits. Alternatively, '`:second`' can be omitted, in which case it is taken to be zero. On the rare hosts that support leap seconds, *second* may be 60.

If the time is followed by '`am`' or '`pm`' (or '`a.m.`' or '`p.m.`'), *hour* is restricted to run from 1 to 12, and '`:minute`' may be omitted (taken to be zero). '`am`' indicates the first half of the day, '`pm`' indicates the second half of the day. In this notation, 12 is the predecessor of 1: midnight is '`12am`' while noon is '`12pm`'. (This is the zero-oriented interpretation of '`12am`' and '`12pm`', as opposed to the old tradition derived from Latin which uses '`12m`' for noon and '`12pm`' for midnight.)

The time may alternatively be followed by a time zone correction, expressed as '`shhmm`', where *s* is '`+`' or '`-`', *hh* is a number of zone hours and *mm* is a number of zone minutes. The zone minutes term, *mm*, may be omitted, in which case the one- or two-digit correction is interpreted as a number of hours. You can also separate *hh* from *mm* with a colon. When a time zone correction is given this way, it forces interpretation of the time relative to Coordinated Universal Time (UTC), overriding any previous specification for the time zone or the local time zone. For example, '`+0530`' and '`+05:30`' both stand for the time zone 5.5 hours ahead of UTC (e.g., India). This is the best way to specify a time zone correction by fractional parts of an hour. The maximum zone correction is 24 hours.

Either '`am`'/'`pm`' or a time zone correction may be specified, but not both.

20.4 Time zone items

A *time zone item* specifies an international time zone, indicated by a small set of letters, e.g., '`UTC`' or '`Z`' for Coordinated Universal Time. Any included periods are ignored. By following a non-daylight-saving time zone by the string '`DST`' in a separate word (that is, separated by some white space), the corresponding daylight saving time zone may be specified. Alternatively, a non-daylight-saving time zone can be followed by a time zone correction, to add the two values. This is normally done only for '`UTC`'; for example, '`UTC+05:30`' is equivalent to '`+05:30`'.

Time zone items other than '`UTC`' and '`Z`' are obsolescent and are not recommended, because they are ambiguous; for example, '`EST`' has a different meaning in Australia than in the United

States, and '`A`' has different meaning as a military time zone than as an obsolescent RFC 822 time zone. Instead, it's better to use unambiguous numeric time zone corrections like '`-0500`', as described in the previous section.

If neither a time zone item nor a time zone correction is supplied, timestamps are interpreted using the rules of the default time zone (see Section 20.10 [Specifying time zone rules], page 79).

20.5 Combined date and time of day items

The ISO 8601 date and time of day extended format consists of an ISO 8601 date, a '`T`' character separator, and an ISO 8601 time of day. This format is also recognized if the '`T`' is replaced by a space.

In this format, the time of day should use 24-hour notation. Fractional seconds are allowed, with either comma or period preceding the fraction. ISO 8601 fractional minutes and hours are not supported. Typically, hosts support nanosecond timestamp resolution; excess precision is silently discarded.

Here are some examples:

```
2012-09-24T20:02:00.052-05:00
2012-12-31T23:59:59,999999999+11:00
1970-01-01 00:00Z
```

20.6 Day of week items

The explicit mention of a day of the week will forward the date (only if necessary) to reach that day of the week in the future.

Days of the week may be spelled out in full: '`Sunday`', '`Monday`', '`Tuesday`', '`Wednesday`', '`Thursday`', '`Friday`' or '`Saturday`'. Days may be abbreviated to their first three letters, optionally followed by a period. The special abbreviations '`Tues`' for '`Tuesday`', '`Wednes`' for '`Wednesday`' and '`Thur`' or '`Thurs`' for '`Thursday`' are also allowed.

A number may precede a day of the week item to move forward supplementary weeks. It is best used in expression like '`third monday`'. In this context, '`last` *`day`*' or '`next` *`day`*' is also acceptable; they move one week before or after the day that *day* by itself would represent.

A comma following a day of the week item is ignored.

20.7 Relative items in date strings

Relative items adjust a date (or the current date if none) forward or backward. The effects of relative items accumulate. Here are some examples:

```
1 year
1 year ago
3 years
2 days
```

The unit of time displacement may be selected by the string '`year`' or '`month`' for moving by whole years or months. These are fuzzy units, as years and months are not all of equal duration. More precise units are '`fortnight`' which is worth 14 days, '`week`' worth 7 days, '`day`' worth 24 hours, '`hour`' worth 60 minutes, '`minute`' or '`min`' worth 60 seconds, and '`second`' or '`sec`' worth one second. An '`s`' suffix on these units is accepted and ignored.

The unit of time may be preceded by a multiplier, given as an optionally signed number. Unsigned numbers are taken as positively signed. No number at all implies 1 for a multiplier. Following a relative item by the string '`ago`' is equivalent to preceding the unit by a multiplier with value -1.

The string '`tomorrow`' is worth one day in the future (equivalent to '`day`'), the string '`yesterday`' is worth one day in the past (equivalent to '`day ago`').

The strings '`now`' or '`today`' are relative items corresponding to zero-valued time displacement, these strings come from the fact a zero-valued time displacement represents the current time when not otherwise changed by previous items. They may be used to stress other items, like in '`12:00 today`'. The string '`this`' also has the meaning of a zero-valued time displacement, but is preferred in date strings like '`this thursday`'.

When a relative item causes the resulting date to cross a boundary where the clocks were adjusted, typically for daylight saving time, the resulting date and time are adjusted accordingly.

The fuzz in units can cause problems with relative items. For example, '`2020-07-31 -1 month`' might evaluate to 2020-07-01, because 2020-06-31 is an invalid date. To determine the previous month more reliably, you can ask for the month before the 15th of the current month. For example:

```
$ date -R
Thu, 31 Jul 2020 13:02:39 -0400
$ date --date='-1 month' +'Last month was %B?'
Last month was July?
$ date --date="$(date +%Y-%m-15) -1 month" +'Last month was %B!'
Last month was June!
```

Also, take care when manipulating dates around clock changes such as daylight saving leaps. In a few cases these have added or subtracted as much as 24 hours from the clock, so it is often wise to adopt universal time by setting the `TZ` environment variable to '`UTC0`' before embarking on calendrical calculations.

20.8 Pure numbers in date strings

The precise interpretation of a pure decimal number depends on the context in the date string.

If the decimal number is of the form *yyyymmdd* and no other calendar date item (see Section 20.2 [Calendar date items], page 75) appears before it in the date string, then *yyyy* is read as the year, *mm* as the month number and *dd* as the day of the month, for the specified calendar date.

If the decimal number is of the form *hhmm* and no other time of day item appears before it in the date string, then *hh* is read as the hour of the day and *mm* as the minute of the hour, for the specified time of day. *mm* can also be omitted.

If both a calendar date and a time of day appear to the left of a number in the date string, but no relative item, then the number overrides the year.

20.9 Seconds since the Epoch

If you precede a number with '`@`', it represents an internal timestamp as a count of seconds. The number can contain an internal decimal point (either '`.`' or '`,`'); any excess precision not supported by the internal representation is truncated toward minus infinity. Such a number cannot be combined with any other date item, as it specifies a complete timestamp.

Internally, computer times are represented as a count of seconds since an Epoch—a well-defined point of time. On GNU and POSIX systems, the Epoch is 1970-01-01 00:00:00 UTC, so '`@0`' represents this time, '`@1`' represents 1970-01-01 00:00:01 UTC, and so forth. GNU and most other POSIX-compliant systems support such times as an extension to POSIX, using negative counts, so that '`@-1`' represents 1969-12-31 23:59:59 UTC.

Most modern systems count seconds with 64-bit two's-complement integers of seconds with nanosecond subcounts, which is a range that includes the known lifetime of the universe with nanosecond resolution. Some obsolescent systems count seconds with 32-bit two's-complement integers and can represent times from 1901-12-13 20:45:52 through 2038-01-19 03:14:07 UTC. A few systems sport other time ranges.

On most hosts, these counts ignore the presence of leap seconds. For example, on most hosts '`@1483228799`' represents 2016-12-31 23:59:59 UTC, '`@1483228800`' represents 2017-01-01 00:00:00 UTC, and there is no way to represent the intervening leap second 2016-12-31 23:59:60 UTC.

20.10 Specifying time zone rules

Normally, dates are interpreted using the rules of the current time zone, which in turn are specified by the `TZ` environment variable, or by a system default if `TZ` is not set. To specify a different set of default time zone rules that apply just to one date, start the date with a string of the form '`TZ="`*`rule`*`"`'. The two quote characters ('`"`') must be present in the date, and any quotes or backslashes within *rule* must be escaped by a backslash.

For example, with the GNU `date` command you can answer the question "What time is it in New York when a Paris clock shows 6:30am on October 31, 2019?" by using a date beginning with '`TZ="Europe/Paris"`' as shown in the following shell transcript:

```
$ export TZ="America/New_York"
$ date --date='TZ="Europe/Paris" 2019-10-31 06:30'
Sun Oct 31 01:30:00 EDT 2019
```

In this example, the '`--date`' operand begins with its own `TZ` setting, so the rest of that operand is processed according to '`Europe/Paris`' rules, treating the string '`2019-10-31 06:30`' as if it were in Paris. However, since the output of the `date` command is processed according to the overall time zone rules, it uses New York time. (Paris was normally six hours ahead of New York in 2019, but this example refers to a brief Halloween period when the gap was five hours.)

A `TZ` value is a rule that typically names a location in the '`tz`' database. A recent catalog of location names appears in the TWiki Date and Time Gateway. A few non-GNU hosts require a colon before a location name in a `TZ` setting, e.g., '`TZ=":America/New_York"`'.

The '`tz`' database includes a wide variety of locations ranging from '`Arctic/Longyearbyen`' to '`Antarctica/South_Pole`', but if you are at sea and have your own private time zone, or if you are using a non-GNU host that does not support the '`tz`' database, you may need to use a POSIX rule instead. Simple POSIX rules like '`UTC0`' specify a time zone without daylight saving time; other rules can specify simple daylight saving regimes. See Section "Specifying the Time Zone with `TZ`" in *The GNU C Library*.

20.11 Authors of `parse_datetime`

`parse_datetime` started life as `getdate`, as originally implemented by Steven M. Bellovin (`smb@research.att.com`) while at the University of North Carolina at Chapel Hill. The code was later tweaked by a couple of people on Usenet, then completely overhauled by Rich $alz (`rsalz@bbn.com`) and Jim Berets (`jberets@bbn.com`) in August, 1990. Various revisions for the GNU system were made by David MacKenzie, Jim Meyering, Paul Eggert and others, including renaming it to `get_date` to avoid a conflict with the alternative Posix function `getdate`, and a later rename to `parse_datetime`. The Posix function `getdate` can parse more locale-specific dates using `strptime`, but relies on an environment variable and external file, and lacks the thread-safety of `parse_datetime`.

This chapter was originally produced by François Pinard (`pinard@iro.umontreal.ca`) from the '`parse_datetime.y`' source code, and then edited by K. Berry (`kb@cs.umb.edu`).

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

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.

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

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index