Forms Mode User's Manual

Forms-Mode version 1.2, patchlevel 9

July 1991

Johan Vromans
jv@mh.nl

31. Forms mode

Forms mode is an Emacs major mode for working with simple plain-text databases in a forms-oriented manner. In forms mode, the information in these files is presented in an Emacs window in a user-defined format, one record at a time. Forms can be inspected read-only (viewing) or modified and updated.

Forms mode is not a simple major mode, but requires two files to do its job: a control file and a data file. The data file holds the actual data which will be presented. The control file describes how it will be presented.

31.1 What is in a forms

Let's illustrate forms mode with an example. Suppose you are looking at your '`/etc/passwd`' file, and your screen looks as follows:

```
====== /etc/passwd ======

User : root   Uid: 0   Gid: 1

Name : Super User

Home : /

Shell: /bin/sh
```

As you can see, the familiar fields from the entry for the super user are all there, but instead of being colon-separated on one single line, they make up a forms.

The contents of the forms consists of the contents of the fields of the record (e.g. "root", "0", "1", "Super User") interspersed with normal text (e.g "User : ", "Uid: ").

You can define yourself how text and fields will be used to make up the forms.

When you modifiy the contents of the forms, it will be analyzed and the new contents of the fields of this record will be extracted. If possible, the file will be updated with the new contents.

31.2 Data file format

Files for use with forms mode are very simple – each record (line) forms the contents of one form. Each record is supposed to consist of a number of fields. These fields are separated by the value of the string `forms-field-sep`, which is a `TAB` by default.

Fields may contain text which shows up in the forms in multiple lines. These lines are separated in the field using a *pseudo-newline* character which is defined by the value of the string `forms-multi-line`. Its default value is a Control-K character. If it is set to `nil` multiple line fields are prohibited.

31.3 Control file format

The control file serves two purposes.

First, it defines the data file to use, and its properties.

Second, the Emacs buffer it occupies will be used by the forms mode to display the forms.

The contents of the control file are evaluated using the Emacs command `eval-current-buffer`, hence must contain valid Emacs-lisp expressions. These expressions must set the following lisp variables to a suitable value:

`forms-file`

This variable must be set to the name of the data file.

Example:

```
(setq forms-file "my/data-file")
```

`forms-format-list`

This variable describes the way the fields of the record are formatted on the screen. See the next section for details.

`forms-number-of-fields`

This variable holds the number of fields in each record of the data file.

Example:

```
(setq forms-number-of-fields 10)
```

An error will be given if one of the above values has not been set.

Other variables that may be set from the control file are optional. Most of them have suitable default values.

`forms-field-sep`
: This variable may be used to designate the string which separates the fields in the records of the data file. If not set, it defaults to a string containing a single `TAB` character.
Example:

```
(setq forms-field-sep "\t")
```

`forms-read-only`
: If set to a value other than `nil`, the data file is treated read-only. If the data file can not be written into, read-only mode is enforced. The default value for `forms-read-only` is derieved from the access permissions of the data file.
Example:

```
(set forms-read-only t)
```

`forms-multi-line`
: This variable may be set to allow multi-line text in fields. It should be set to a string of one character, which denotes the pseudo new-line character to be used to separate the text lines.
Its default value is Control-K (octal 013). If set to `nil`, multi-line text fields are prohibited.
It may not be a character contained in `forms-field-sep`.
Example:

```
(setq forms-multi-line "\C-k")
```

`forms-forms-scroll`
: See section 31.6 [Forms Mode Commands], page 6 for the description.

`forms-forms-jump`
: See section 31.6 [Forms Mode Commands], page 6 for the description.

`forms-new-record-filter`
: The control file may define a function `forms-new-record-filter`, or set `forms-new-record-filter` to such a function. If so, this function is called when a new record is created to supply default values for fields.

`forms-modified-record-filter`
: The control file may define a function `forms-modified-record-filter`, or set `forms-modified-record-filter` to such a function. If so, this function is called when a record is modified, just before writing the modified record back to the data file.

31.4 The forms format description

The value of the variable `forms-format-list` is used to specify the format of the forms. It must be a list of formatting elements, each of which can be a string, number, lisp list or a lisp symbol that evaluates to one of these. The formatting elements are processed in the order they appear in the list.

A `string` formatting element is inserted in the forms "as is".

A `number` element selects a field of the record. The contents of this field are inserted. The first field of the record has number 1 (one).

A `lisp list` specifies a function call. This function is called every time a record is displayed, and its result, that must be a string, is inserted in the forms. The function should do nothing but returning a string. The fields of the record being displayed are available to this function as the list `forms-fields` and can be accessed using `(nth` *FIELD NUMBER* `forms-fields)`. Fields are numbered starting from 1 (one).

A `lisp symbol` must evaluate to one of the above possibilities.

If a record does not contain the number of fields as specified in `forms-number-of-fields`, a warning message will be printed. Excess fields are ignored, missing fields are set to empty.

The control file which shows your '`/etc/passwd`' file as demonstrated in the beginning of this document might look as follows:

```
;; This demo visits /etc/passwd.

(setq forms-file "/etc/passwd")
(setq forms-number-of-fields 7)
(setq forms-read-only t)                ; to make sure
(setq forms-field-sep ":")
(setq forms-multi-line nil)             ; not allowed

(setq forms-format-list
      (list
       "====== /etc/passwd ======\n\n"
       "User : "    1
       "   Uid: "   3
       "   Gid: "   4
       "\n\n"
       "Name : "    5
```

```
            "\n\n"
            "Home : "    6
            "\n\n"
            "Shell: "    7
            "\n"))
```

When functions are to be used in `forms-format-list` they must be quoted to prevent them from being evaluated too early:

```
     (setq forms-format-list
           (list
            "====== " forms-file " ======\n\n"
            "User : "    1
            '(make-string 20 ?-)
            ...
           ))
```

Alternativily, instead of quoting the functions, the whole list may be quoted:

```
     (setq forms-format-list
          '(
            "====== " forms-file " ======\n\n"
            "User : "    1
            (make-string 20 ?-)
            ...
           ))
```

Upon startup, the contents of `forms-format-list` are validated. If errors are encountered, processing is aborted with an error message which includes a descriptive text. See section 31.9 [Error Messages], page 10, for a detailed list of error messages.

31.5 Modifying A Form-mode Buffer's Contents

If a forms is not read-only, its contents can be modified.

All normal editor commands may be used to change the forms. There is no distinction between the "fixed" text and the text from the fields of the records. However, upon completion, the forms is parsed to extract the new contents of the fields. The "fixed" portions of the forms are used to delimit the fields, these portions should therefore not be modified to avoid the risk that the field contents cannot be determined. Moreover, ambiguous field contents, which can not be discriminated from "fixed" text, must be avoided.

If the contents of the forms cannot be recognized properly, this is signaled using a descriptive text. See section 31.9 [Error Messages], page 10, for more info. The cursor will indicate the last part of the forms which was successfully parsed.

If `forms-modified-record-filter` has been set, this function is called before the new data is written to the data file. The function is called with one argument: a vector that contains the contents of the fields of a record. Fields can be referenced or modified using the lisp functions `aref` and `aset`. The first field has number 1 (one). The function must return the (possibly modified) vector to the calling environment.

```
(defun my-modified-record-filter (record)
  ;; modify second field
  (aset record 2 (current-time-string))
  record ; return it
)
(setq forms-modified-record-filter 'my-modified-record-filter)
```

31.6 Forms mode commands

`M-x forms-find-file` *file*

Visits *file*, runs `eval-current-buffer` on it, and puts the buffer into forms-mode. The first record of the data file will be loaded and shown.

The modeline will display the major mode `"Forms"` followed by the minor mode `"View"` if the file is visited read-only. The number of the current record (*n*) and the total number of records (*t*) in the file is shown in the modeline as `"n/t"`.
For example:

```
--%%-Emacs: forms-passwd          (Forms View 1/54)----All-------
```

`M-x forms-find-file-other-window` *file*

This command is similar to `forms-find-file`, but visits the file in another window.

If the buffer is not read-only, you may change the buffer to modify the fields in the record. When the current record is left, e.g. by switching to another record, the contents of the buffer are parsed using the specifications in `forms-format-list`, and a new record is constructed which replaces the current record in the data file. Fields of the record which are not shown in the forms are not modified; they retain their original contents.

Most forms mode commands are bound to keys, and are accessible with the conventional `C-c` prefix. In read-only mode this prefix is not used. See section 31.7 [Key Bindings], page 9, for the default key bindigs used by forms mode.

The following commands are available within forms mode.

`forms-next-record`
: shows the next record. With a prefix argument, show the n-th next record.

`forms-prev-record`
: shows the previous record. With a prefix argument, show the n-th previous record.

`forms-jump-record`
: jumps to a record by number. A prefix argument is used for the record number to jump to. If no prefix argument is supplied, a record number is asked for in the minibuffer. If an invalid record number is supplied, an error message is displayed reading the offending record number, and the allowable range of numbers.

`forms-first-record`
: jumps to the first record.

`forms-last-record`
: jumps to the last record. Also re-counts the number of records in the data file.

`forms-next-field`
: jumps to the next field in the forms. With a numeric argument: jumps that many fields, or to the first field if there are not that many fields left.

: Jumping to fields is implemented using markers, which are placed in front of the fields. If the contents of the forms are modified, the markers are adjusted. However, if text around a marker has been deleted from the screen and inserted again it is possible that this marker no longer points at its field correctly. See section Markers in *the GNU Emacs Lisp Manual*, for more information on markers.

`forms-view-mode`
: switches to read-only mode. Forms mode commands may no longer be prefixed with `C-c`.

`forms-edit-mode`
: switches to edit mode. Forms mode commands must be prefixed with `C-c`. Switching to edit mode is only possible if write access to the data file is allowed.

`forms-insert-record`
: create a new record that is inserted before the current record. An empty form is pre-

sented, which can be filled in using familiar editor commands. With a prefix argument (for example, `C-u`) the new record is created *after* the current one.

If a function `forms-new-record-filter` was defined in the control file, this function is called to fill in default values for fields. The function is passed a vector of empty strings, one for each field. For convenience, an additional element is added so the numbers of the elements are the same as the numbers used in the forms description. The function must return the (updated) vector.

Instead of defining the function, `forms-new-record-filter` may be set to a function name.

Example:

```
(defun my-new-record-filter (fields)
  (aset fields 5 (login-name))
  (aset fields 1 (current-time-string))
  ;; and return it
  fields)
(setq forms-new-record-filter 'my-new-record-filter)
```

`forms-delete-record`
: deletes the current record. You are prompted for confirmation before the record is deleted unless a prefix argument has been provided.

`forms-search` *regexp*
: searches for *regexp* in all records following this one. If found, this record is shown. The next time it is invoked, the previous regexp is the default, so you can do repeated searches by simply pressing `RET` in response to the prompt.

`revert-buffer`
: reverts a possibly modified forms to its original state. It only affects the record currently in the forms.

`forms-exit`
: terminates forms processing. The data file is saved if it has been modified.

`forms-exit-no-save`
: aborts forms processing. If the data file has been modified Emacs will ask questions.

`describe-mode`
: gives additional help.

`save-buffer`
: saves the changes in the data file, if modified.

If the variable `forms-forms-scrolls` is set to a value other than `nil` (which it is, by default), the Emacs functions `scroll-up` and `scroll-down` will perform a `forms-next-record` and `forms-prev-record` when in forms mode. So you can use your favourite page commands to page through the data file.

Likewise, if the variable `forms-forms-jump` is not `nil` (which it is, by default), Emacs functions `beginning-of-buffer` and `end-of-buffer` will perform `forms-first-record` and `forms-last-record` when in forms mode.

After forms mode is entered, functions contained in `forms-mode-hooks` are executed to perform user defined customization.

31.7 Key bindings

This section describes the key bindings as they are defined when invoking forms mode.
All commands must be prefixed with `C-c` when editing a forms. If a forms is read-only, `C-c` is not used. The only exception to this rule is `forms-next-field`, which is bound to `TAB` in all maps.

```
C-c TAB   forms-next-field
C-c SPC   forms-next-record
C-c <     forms-first-record
C-c >     forms-first-record
C-c d     forms-delete-record
C-c e     forms-edit-mode
C-c i     forms-insert-record
C-c j     forms-jump-record
C-c n     forms-next-record
C-c p     forms-prev-record
C-c q     forms-exit
C-c s regexp
          forms-search
C-c v     forms-view-mode
C-c x     forms-exit-no-save
C-c ?     describe-mode
C-c DEL   forms-prev-record
```

31.8 Miscellaneous

A global variable `forms-version` holds the version information of the current implementation of forms mode.

It is very convenient to use symbolic names for the fields in a record. The function `forms-enumerate` provides an elegant means to set a series of variables to consecutive numbers. The function returns the higest number used, so it can be used to set `forms-number-of-fields` also:

```
(setq forms-number-of-fields
      (forms-enumerate
       '(field1 field2 field3 ... )))
```

`field1` will be set to 1, `field2` to 2 and so on.

Care has been taken to localize the current information of the forms mode, so it is possible to visit multiple files in forms mode simultaneously, even if they have different properties.

Since buffer-local functions are not available in this version of GNU Emacs, the definitions of the filter functions `forms-new-record-filter` and `forms-modified-record-filter` are copied into internal, buffer local variables when forms-mode is initialized.

If a control file is visited using the standard `find-file` commands, forms mode can be enabled with the command `M-x forms-mode`.
Forms mode will be automatically enabled if the file contains the string `"-*- forms -*-"` somewhere in the first line. However, this makes it hard to edit the control file itself so you'd better think twice before using this.

The default format for the data file, using `TAB` to separate fields and `C-k` to separate multi-line fields, matches the file format of some popular Macintosh database programs, e.g. FileMaker. So `forms-mode` could decrease the need to use Apple computers.

31.9 Error Messages

This section describes all error messages which can be generated by forms mode.

`'forms-file' has not been set`
: The variable `forms-file` was not set by the control file.

`'forms-number-of-fields' has not been set`
: The variable `forms-number-of-fields` was not set by the control file.

`'forms-number-of-fields' must be > 0`
: The variable `forms-number-of-fields` did not contain a positive number.

`'forms-field-sep' is not a string`

`'forms-multi-line' must be nil or a one-character string`

The variable **`forms-multi-line`** was set to something other than **`nil` or** a single-character string.

`'forms-multi-line' is equal to 'forms-field-sep'`

The variable **`forms-multi-line`** may not be equal to **`forms-field-sep`** for this would make it impossible to distinguish fields and the lines in the fields.

`'forms-format-list' has not been set`

`'forms-format-list' is not a list`

The variable **`forms-format-list`** was not set to a lisp **`list`** by the control file.

`Forms error: field number` *XX* **`out of range 1..`***NN*

A field number was supplied with a value of *XX*, which was not greater than zero and smaller than or equal to the number of fields in the forms, *NN*.

`Forms error: not a function` *FUN*

The first element of the lisp list specified in **`forms-format-list`** did not have a function value.

`Invalid element in 'forms-format-list':` *XX*

A list element was supplied in **`forms-format-list`** which was not a **`string`**, **`number`** nor a **`lisp list`**.

`Parse error: not looking at "..."`

When re-parsing the contents of a forms, the text shown could not be found.

`Parse error: cannot find "..."`

When re-parsing the contents of a forms, the text shown, which separates two fields, could not be found.

`Parse error: cannot parse adjacent fields` *XX* **`and`** *YY*

Fields *XX* and *YY* were not separated by text, so could not be parsed again.

`Record has` *XX* **`fields instead of`** *YY*

The number of fields in this record in the data file did not match **`forms-number-of-fields`**. Missing fields will be set to empty.

`Multi-line fields in this record - update refused!`

The current record contains newline characters, hence can not be written back to the data file, for it would corrupt it.

probably a field was set to a multi-line value, while the setting of **`forms-multi-line`** prohibited this.

`Record number` *XX* **`out of range 1..`***YY*

A jump was made to non-existing record *XX*. *YY* denotes the number of records in the file.

`Stuck at record` *XX*

An internal error prevented a specific record from being retrieved.

31.10 Examples

The following example exploits most of the features of forms-mode. This example is included in the distribution as file '`forms-demo`'.

```
;; This sample forms exploit most of the features of forms mode.

;; Set the name of the data file.
(setq forms-file "forms-data")

;; Use 'forms-enumerate' to set field names and number thereof.
(setq forms-number-of-fields
      (forms-enumerate
       '(arch-newsgroup ; 1
 arch-volume ; 2
 arch-issue ; and ...
 arch-article ; ... so
 arch-shortname ; ... ... on
 arch-parts
 arch-from
 arch-longname
 arch-keywords
 arch-date
 arch-remarks)))

;; The following functions are used by this form for layout purposes.
;;
(defun arch-tocol (target &optional fill)
  "Produces a string to skip to column TARGET. Prepends newline if needed.
The optional FILL should be a character, used to fill to the column."
  (if (null fill)
      (setq fill ? ))
  (if (< target (current-column))
      (concat "\n" (make-string target fill))
    (make-string (- target (current-column)) fill)))
;;
(defun arch-rj (target field &optional fill)
  "Produces a string to skip to column TARGET minus the width of field FIELD.
Prepends newline if needed. The optional FILL should be a character,
used to fill to the column."
  (arch-tocol (- target (length (nth field forms-fields))) fill))

;; Record filters.
;; This example uses the (defun ...) method of defining.
;;
(defun forms-new-record-filter (the-record)
  "Form a new record with some defaults."
  (aset the-record arch-from (user-full-name))
  (aset the-record arch-date (current-time-string))
```

```
  the-record ; return it
)

;; The format list.
(setq forms-format-list
     (list
       "====== Public Domain Software Archive ======\n\n"
       arch-shortname
       " - " arch-longname
       "\n\n"
       "Article: " arch-newsgroup
       "/" arch-article
       " "
       '(arch-tocol 40)
       "Issue: " arch-issue
       " "
       '(arch-rj 73 10)
       "Date: " arch-date
       "\n\n"
       "Submitted by: " arch-from
       "\n"
       '(arch-tocol 79 ?-)
       "\n"
       "Keywords: " arch-keywords
       "\n\n"
       "Parts: " arch-parts
       "\n\n====== Remarks ======\n\n"
       arch-remarks
     ))

;; That's all, folks!
```

31.11 Credits

Forms mode is developed by Johan Vromans `<jv@mh.nl>` at Multihouse Reseach in the Netherlands.

Harald Hanche-Olsen `<hanche@imf.unit.no>` supplied the idea for the new record filter, and provided better replacements for some internal functions.

Bugfixes and other useful suggestions were supplied by cwitty@portia.stanford.edu, Jonathan I. Kamens, Ignatios Souvatzis and Harald Hanche-Olsen.

This documentation was slightly inspired by the documentation of "rolo mode" by Paul Davis

at Schlumberger Cambridge Research `<davis%scrsu1%sdr.slb.com@relay.cs.net>`.

None of this would have been possible without GNU Emacs of the Free Software Foundation. Thanks, Richard!

Table of Contents