

IBP banding data system: *emacs* customizations

Zoological
Data Processing

John W. Shipman

2009-01-11 21:35

Table of Contents

1. Introduction	2
1.1. How to get this publication	2
2. Survey of modules in this package	2
3. Top-level modules	3
3.1. <code>maps2004m.el</code> : Multiple-station sets, MAPS 2004/2006 protocol	3
3.2. <code>maps2004s.el</code> : Single-station sets, MAPS 2004/2006 protocol	4
3.3. <code>maps1998m.el</code> : Multi-station sets, MAPS 1998 protocol	4
3.4. <code>maps1998s.el</code> : Single-station sets, MAPS 1998 protocol	5
3.5. <code>maws2007m.el</code> : Multi-station sets, MAWS protocol	5
3.6. <code>maws2007s.el</code> : Single-station sets, MAWS protocol	6
3.7. <code>maws2004m.el</code> : Multi-station sets, MAWS protocol	7
3.8. <code>maws2004s.el</code> : Single-station sets, MAWS protocol	7
4. <code>ibp.el</code> : Common logic	8
4.1. Introduction to <code>ibp.el</code>	8
4.2. The pseudo-object convention	8
4.3. <code>ibp.el</code> : Prologue	9
4.4. <code>ibp-tab</code> : Tab function	9
4.5. <code>ibp-tab-once</code> : Single tab function	9
4.6. <code>analyze-line</code> : Where are the parts of the current line?	11
4.7. <code>classify-line</code> : What kind of line is this?	12
4.8. <code>line-head-length</code> : How long is the head of this line?	14
4.9. <code>bracket-field</code> : What field contains a given position?	14
4.10. <code>bracket-tail-field</code> : What tail field contains the cursor?	15
4.11. <code>field-fill</code> : Move to the end of a field	17
4.12. <code>ibp-ditto</code> : Field duplication function	19
4.13. <code>ibp-ditto-once</code> : Duplicate one field	19
4.14. <code>find-prev-transaction</code> : Find the last preceding line with a tail	22
4.15. <code>field-def-object</code> : Field definition object	24
4.16. <code>line-object</code> : Represents one line	24
4.17. <code>line-has-tail-p</code> : Does this line have a tail?	25
4.18. <code>field-object</code> : Location of an actual field	26

1. Introduction

As part of Zoological Data Processing's system for processing bird banding data for the Institute for Bird Populations (IBP), the author has created a set of customizations for the *emacs* text editor that speed up data entry.

Installation and operating instructions for these functions are described in the specification¹.

This document presents the actual code for the *emacs* extensions in lightweight literate programming form. For more information on this form of presentation, see the author's lightweight literate programming page².

The programming language used to extend *emacs* is called *emacs* LISP, also called *e-lisp*. Files in this language have names ending in “.el”. More information on *emacs* is available at the Free Software Foundation page for *emacs*³.

Recommended references for *emacs* and *e-lisp*:

- Glickstein, Bob. Writing GNU Emacs Extensions. O'Reilly, 1997, ISBN 0-56592-261-1.
- Stallman, Richard M. GNU Emacs Manual, For Version 21, 15th Edition. Free Software Foundation, ISBN 188211485X.

1.1. How to get this publication

This publication is available in Web form⁴ and also as a PDF document⁵. Please forward any comments to tcc-doc@nmt.edu.

2. Survey of modules in this package

Here are the modules in the package. Each module, except for *ibp.el*, sets up one particular format.

Each bullet below contains a link to the actual generated *e-lisp* file, followed by a link to the literate exposition of that file in this document.

- *maps2004m.el*: Section 3.1, “*maps2004m.el*: Multiple-station sets, MAPS 2004/2006 protocol” (p. 3)
- *maps2004s.el*: Section 3.2, “*maps2004s.el*: Single-station sets, MAPS 2004/2006 protocol” (p. 4)
- *maps1998m.el*: Section 3.3, “*maps1998m.el*: Multi-station sets, MAPS 1998 protocol” (p. 4)
- *maps1998s.el*: Section 3.4, “*maps1998s.el*: Single-station sets, MAPS 1998 protocol” (p. 5)
- *maws2007m.el*: Section 3.5, “*maws2007m.el*: Multi-station sets, MAWS protocol” (p. 5)
- *maws2007s.el*: Section 3.6, “*maws2007s.el*: Single-station sets, MAWS protocol” (p. 6)
- *maws2004m.el*: Section 3.7, “*maws2004m.el*: Multi-station sets, MAWS protocol” (p. 7)
- *maws2004s.el*: Section 3.8, “*maws2004s.el*: Single-station sets, MAWS protocol” (p. 7)
- *ibp.el*: Section 4, “*ibp.el*: Common logic” (p. 8)

¹ <http://infohost.nmt.edu/~shipman/z/ibp/doc/iband7/emacs.html>

² <http://www.nmt.edu/~shipman/soft/litprog/>

³ <http://directory.fsf.org/emacs.html>

⁴ <http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/>

⁵ <http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/ibpemacs.pdf>

3. Top-level modules

Each of these modules is loaded by the *emacs* user to set up one particular format.

3.1. maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol

This module sets up a field structure for a multi-station set in the current MAPS protocol.

```
;; maps2004m.el: Set up field values and keybindings for
;;   MAPS 2004 protocol, multiple stations.
;; To use, run emacs with this command line option:
;;   emacs -l maps2004m filename
;;
```

maps2004m.el

First we load the common functions used by all the top-level modules.

```
(load "ibp.el")
```

maps2004m.el

Next comes the table that defines the field layout of the tail portion in the MAPS 2004 protocol.

```
(setq field-def-list
  (vector
    (field-def-object 4 "????")      ;; Species code
    (field-def-object 3 "0  ")      ;; Age, how-aged
    (field-def-object 3 "u  ")      ;; Sex, how-sexed
    (field-def-object 4 nil )        ;; Skull through fat
    (field-def-object 4 nil )        ;; Four old molt codes
    (field-def-object 4 nil )        ;; Pri.cov/Sec.cov/Pri/Sec
    (field-def-object 4 nil )        ;; Tert/Rect/Body plum/Nonfeath
    (field-def-object 3 nil )        ;; Wing length
    (field-def-object 4 nil )        ;; Weight
    (field-def-object 5 "3????")    ;; Status, date
    (field-def-object 3 nil )        ;; Time
    (field-def-object 4 nil )        ;; Station code
    (field-def-object 2 nil ) ) )    ;; Net
```

maps2004m.el

Finally, we bind keys to the tab and duplicate functions. (Putting these lines in *ibp.el* doesn't work. Also, the editor needs to be put into fundamental mode with "M-x fundamental-mode" because other modes may not treat *tab* correctly.)

There is quite a lot of trickery packaged into these two lines. The `global-set-key` function takes two arguments: the description of a key, and the (quoted) name of the function to be bound to it.

Key descriptions may use either quoted strings or *emacs* vector objects.

For binding the *tab* key, we use the quote string *C-i*.

```
(global-set-key "\C-i" 'ibp-tab)
```

maps2004m.el

The name of the ditto key varies across platforms. We'll use vector objects here, which are a list of symbols enclosed in square brackets. Sometimes (as in a separate *emacs* window in Linux) it is considered *C-^*, but sometimes (as in an *emacs* session running in a shell window) it is treated as *C-6*. We bind both of these so that the module will work in either case.

```
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)
```

3.2. maps2004s.el: Single-station sets, MAPS 2004/2006 protocol

Similar to Section 3.1, “maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol” (p. 3), but for single-station sets where the station code is not present.

```
;; maps2004s.el: Set up field values and keybindings for
;;   MAPS 2004 protocol, single stations.
;; To use, run emacs with this command line option:
;;   emacs -l maps2004s filename
;;
(load "ibp.el")
(setq field-def-list
  (vector (field-def-object 4 "????")      ;; Species code
          (field-def-object 3 "0  ")      ;; Age, how-aged
          (field-def-object 3 "u  ")      ;; Sex, how-sexed
          (field-def-object 4 nil)        ;; Skull through fat
          (field-def-object 4 nil)        ;; Four old molt codes
          (field-def-object 4 nil)        ;; Pri.cov/Sec.cov/Pri/Sec
          (field-def-object 4 nil)        ;; Tert/Rect/Body
  plum/Nonfeath
  (field-def-object 3 nil)                ;; Wing length
  (field-def-object 4 nil)                ;; Weight
  (field-def-object 5 "3????")           ;; Status, date
  (field-def-object 3 nil)                ;; Time
  (field-def-object 2 nil)                ;; Net
)
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)
```

3.3. maps1998m.el: Multi-station sets, MAPS 1998 protocol

Like Section 3.1, “maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol” (p. 3), but uses the pre-2004 field layout, with ten micro-aging fields.

```
;; maps1998m.el: Set up field values and keybindings for
;;   MAPS 2004 protocol, single stations.
;; To use, run emacs with this command line option:
;;   emacs -l maps1998m filename
;;
(load "ibp.el")
(setq field-def-list
  (vector
    (field-def-object 4 "????")      ;; Species code
    (field-def-object 3 "0  ")      ;; Age, how-aged
    (field-def-object 3 "u  ")      ;; Sex, how-sexed
    (field-def-object 4 nil)        ;; Skull through fat
    (field-def-object 4 nil)        ;; Four old molt codes
```

```

      (field-def-object 6 nil )
      (field-def-object 4 nil )           ;; Ten new fields divided 6/4
      (field-def-object 3 nil )           ;; Wing length
      (field-def-object 4 nil )           ;; Weight
      (field-def-object 5 "3????")      ;; Status, date
      (field-def-object 3 nil )           ;; Time
      (field-def-object 4 nil )           ;; Station code
      (field-def-object 2 nil ) ) )      ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)

```

3.4. maps1998s.el: Single-station sets, MAPS 1998 protocol

Like Section 3.3, “maps1998m.el: Multi-station sets, MAPS 1998 protocol” (p. 4), but for single-station sets with no station codes.

maps1998s.el

```

;; maps1998s.el: Set up field values and keybindings for
;; MAPS 1998 protocol, single stations.
;; To use, run emacs with this command line option:
;; emacs -l maps1998s filename
;;
(load "ibp.el")
(setq field-def-list
  (vector
    (field-def-object 4 "????")           ;; Species code
    (field-def-object 3 "0  ")           ;; Age, how-aged
    (field-def-object 3 "u  ")           ;; Sex, how-sexed
    (field-def-object 4 nil )             ;; Skull through fat
    (field-def-object 4 nil )             ;; Four old molt codes
    (field-def-object 6 nil )
    (field-def-object 4 nil )             ;; Ten new fields divided 6/4
    (field-def-object 3 nil )             ;; Wing length
    (field-def-object 4 nil )             ;; Weight
    (field-def-object 5 "3????")         ;; Status, date
    (field-def-object 3 nil )             ;; Time
    (field-def-object 2 nil ) ) )         ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)

```

3.5. maws2007m.el: Multi-station sets, MAWS protocol

For the MAWS 2007 protocol, with the station code on encounter lines, and a four-character net field.

maws2007m.el

```

;; maws2007m.el: Set up field values and keybindings for
;; MAWS 2007 protocol, multiple stations.
;; To use, run emacs with this command line option:
;; emacs -l maws2007m filename
;;
(load "ibp.el")

```

```
(setq field-def-list
  (vector
    (field-def-object 6 "??????") ;; Species code
    (field-def-object 3 "0  ") ;; Age, how-aged
    (field-def-object 3 "u  ") ;; Sex, how-sexed
    (field-def-object 4 nil ) ;; Skull through fat
    (field-def-object 4 nil ) ;; Four old molt codes
    (field-def-object 4 nil ) ;; Micro-aging: 8 fields...
    (field-def-object 4 nil ) ;; ...(divided 2/4/2 on the sheet)

    (field-def-object 3 nil ) ;; Wing length
    (field-def-object 4 nil ) ;; Mass (used to be weight)
    (field-def-object 5 "3????") ;; Status, date
    (field-def-object 3 nil ) ;; Time
    (field-def-object 4 nil ) ;; Station code
    (field-def-object 4 nil ) ) ) ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)
```

3.6. maws2007s.el: Single-station sets, MAWS protocol

Same as maws2007m.el except that the station field is omitted.

maws2007s.el

```
;; maws2007s.el: Set up field values and keybindings for
;; MAWS 2007 protocol, single stations.
;; To use, run emacs with this command line option:
;; emacs -l maws2007s filename
;;
(load "ibp.el")
(setq field-def-list
  (vector
    (field-def-object 6 "??????") ;; Species code
    (field-def-object 3 "0  ") ;; Age, how-aged
    (field-def-object 3 "u  ") ;; Sex, how-sexed
    (field-def-object 4 nil ) ;; Skull through fat
    (field-def-object 4 nil ) ;; Four old molt codes
    (field-def-object 4 nil ) ;; Micro-aging: 8 fields...
    (field-def-object 4 nil ) ;; ...(divided 2/4/2 on the sheet)

    (field-def-object 3 nil ) ;; Wing length
    (field-def-object 4 nil ) ;; Mass (used to be weight)
    (field-def-object 5 "3????") ;; Status, date
    (field-def-object 3 nil ) ;; Time
    (field-def-object 4 nil ) ) ) ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)
```

3.7. maws2004m.el: Multi-station sets, MAWS protocol

MAWS 2004 multi-station protocol.

maws2004m.el

```
;; maws2004m.el: Set up field values and keybindings for
;; MAWS 2004 protocol, multiple stations.
;; To use, run emacs with this command line option:
;; emacs -l maws2004m filename
;;
(load "ibp.el")
(setq field-def-list
  (vector
    (field-def-object 6 "??????") ;; Species code
    (field-def-object 3 "0  ") ;; Age, how-aged
    (field-def-object 3 "u  ") ;; Sex, how-sexed
    (field-def-object 4 nil ) ;; Skull through fat
    (field-def-object 4 nil ) ;; Four old molt codes
    (field-def-object 4 nil ) ;; Micro-aging: 8 fields...
    (field-def-object 4 nil ) ;; ...(divided 2/4/2 on the sheet)

    (field-def-object 3 nil ) ;; Wing length
    (field-def-object 4 nil ) ;; Mass (used to be weight)
    (field-def-object 5 "3????") ;; Status, date
    (field-def-object 3 nil ) ;; Time
    (field-def-object 4 nil ) ;; Station code
    (field-def-object 2 nil ) ) ) ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)
```

3.8. maws2004s.el: Single-station sets, MAWS protocol

MAWS 2004 single-station protocol.

maws2004s.el

```
;; maws2004s.el: Set up field values and keybindings for
;; MAWS 2004 protocol, single stations.
;; To use, run emacs with this command line option:
;; emacs -l maws2004s filename
;;
(load "ibp.el")
(setq field-def-list
  (vector
    (field-def-object 6 "??????") ;; Species code
    (field-def-object 3 "0  ") ;; Age, how-aged
    (field-def-object 3 "u  ") ;; Sex, how-sexed
    (field-def-object 4 nil ) ;; Skull through fat
    (field-def-object 4 nil ) ;; Four old molt codes
    (field-def-object 4 nil ) ;; Micro-aging: 8 fields...
    (field-def-object 4 nil ) ;; ...(divided 2/4/2 on the sheet)

    (field-def-object 3 nil ) ;; Wing length
    (field-def-object 4 nil ) ;; Mass (used to be weight)
```

```

      (field-def-object 5 "3????" )    ;; Status, date
      (field-def-object 3 nil )        ;; Time
      (field-def-object 2 nil ) ) )    ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-^] 'ibp-ditto)
(global-set-key [?\C-6] 'ibp-ditto)

```

4. `ibp.el`: Common logic

The functionality of this package boils down to two *emacs* functions that are intended to bound to specific keys:

- `ibp-ditto` duplicates the corresponding field on the previous line. For example, in the time field, it finds the time field on the previous line (if any), and copies that time to the current field.

This function is typically bound to the *tab* key.

- `ibp-tab` moves to the end of the current field. If the current field is not full, it also supplies default content. For example, in the age/how-aged fields of a banding record, this function fills in the default content "0 ", which means unknown age and two blank how-codes.

This function is typically bound to `C-^` (control-shift-6).

This package assumes that a global vector named `field-def-list` to have been loaded. That vector describes the fields in the encounter line, so that the `tab` and `duplicate` function know which groups of characters on the line are considered a field. For the files that define this vector, see Section 3, “Top-level modules” (p. 3).

4.1. Introduction to `ibp.el`

Some definitions:

head

The part of the record up to, but not including, the species code. In general, this includes the encounter code and the band number. For recap lines, the encounter code may be omitted. The band number may be either all nine characters, or just the last two.

tail

The fixed-format part of the record following the head. For the purposes of this package, the net field is the last field in the record. Data after that may be missing, and `tab` and `duplicate` functions are of little or no use.

This terminology differs from that used in the internal specification⁶: in that document, the fixed-format part of the record is called the *body*, while the fields following the net field are called the tail.

4.2. The pseudo-object convention

The author prefers an object-oriented style in programming generally. In `e-lisp`, objects are simulated using lists in which each position has a fixed meaning.

Furthermore, each position in one of these lists has an *accessor function* that extracts the datum from that position. For example, each description of a field in the record layout is described by a sort of “field definition object” describing the length of the field and its default content.

⁶ <http://www.nmt.edu/~shipman/z/ibp/doc/iband7/ims/>

For example, if such an object is contained in some `e-lisp` variable F , then the *accessor function* (`field-def-len F`) returns the field length.

The pseudo-objects used in this application are defined in:

- Section 4.15, “field-def-object: Field definition object” (p. 24).
- Section 4.16, “line-object: Represents one line” (p. 24).
- Section 4.18, “field-object: Location of an actual field” (p. 26).

4.3. `ibp.el`: Prologue

The actual code for `ibp.el` starts with a comment that references this documentation. Semicolon (`;`) is the `e-lisp` comment character.

```
ibp.el
```

```
;; ibp.el: Common functions for emacs IBP banding data extensions
;;-----
;; For documentation, see:
;;   http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/
;;-----
```

4.4. `ibp-tab`: Tab function

This command has two different functions. If the cursor is sitting inside an incomplete field, it fills up the rest of the field with characters from the field's default content. If the cursor is sitting within a completed field, it just moves the cursor just past the end of that field.

```
ibp.el
```

```
;; - - - i b p - t a b - - -

(defun ibp-tab (count)
  "Tab function for IBP data entry; supports repeat count."
```

This command can be repeated using the usual *emacs* `C-u` mechanism. The convention for repeatable commands is that they take a count argument; the default value of this argument is one, but the user can use `C-u` to repeat it an arbitrary number of times. The `interactive` function sets up a function that takes a numeric argument; the “`p`” argument specifies that the argument must be a number, and it is passed to the function as an integer (rather than as a string).

```
ibp.el
```

```
(interactive "p")      ;; Argument is the repeat count, default 1
```

Using a `while` loop, we call the `ibp-tab-once` function `count` times.

```
ibp.el
```

```
(while (> count 0)
  (ibp-tab-once)
  (setq count (1- count))))
```

4.5. `ibp-tab-once`: Single tab function

This function executes the tab function once. Here is its prologue, and its Cleanroom intended function. “Point” means the cursor's position within the *emacs* buffer.

```
(defun ibp-tab-once ()
  "Tab function for IBP data entry.

  [ if (line containing point does not have a tail)
    or (point is beyond the end of the tail) ->
      signal an error and terminate
    else if both point and the end of the line are within the
    same field ->
      buffer := buffer with filler appended from the element of
              field-def-list corresponding to that field, so as
              to bring the end of line to the end of that field
      point  := point advanced past that field
    else ->
      point := point advanced past the field containing point ]"
```

The “buffer” referred to above is the current editing buffer.

The next step is to start a `let` function, which creates a local scope in which we can manipulate two local variables:

- `line` will hold a `line-object` representing the current line.
- `field` will hold a `field-object` representing the last field currently existing on the current line.

```
(let (line      ;; line-object for the line containing point
      field)   ;; field-object for the field containing end of line
```

We call the `analyze-line` function to look at the line and return a `line-object` that tells us the position of the major parts of the current line.

```
;; [ line := line-object for the line containing point ]
(setq line (analyze-line))
```

The tab function works only within the tail portion of the line. If the current line has no tail, we must report an error. For the function that tests whether a line has a tail, see Section 4.17, “`line-has-tail-p`: Does this line have a tail?” (p. 25).

```
;; [ if line has a tail -> I
;;   else -> error/exit ]
(if (not (line-has-tail-p line))
    (error "Tab is valid only on transaction lines with a tail."))
```

Next we need to figure out which field contains the cursor.

- If the cursor is before the tail, we set `field` to a `field-object` representing the head of the line.
- If the cursor is in some field within the tail, we set `field` to a `field-object` describing that field.
- If the cursor is beyond the tail, that’s an error.

The work of locating the field is done by Section 4.9, “`bracket-field`: What field contains a given position?” (p. 14), which returns `null` if the cursor is past the tail, or a `field-object` otherwise.

```

;; [ if line has a tail ->
;;     if point is in the head of line ->
;;         field := a field-object representing the head, with nil
filler
;;     if point is in a tail field ->
;;         field := a field-object representing that field and its
;;                 filler from field-def-list
;;     if point is beyond the tail fields ->
;;         error/exit ]
(setq field (bracket-field line (point)))
(if (null field)
    (error "You are beyond the fields we know.))

```

If the end of the line is inside `field`, we have to fill out the field from the default content. In any case, we leave the cursor at the end of the field. This logic is handled by Section 4.11, “`field-fill`: Move to the end of a field” (p. 17).

4.6. analyze-line: Where are the parts of the current line?

This function locates the major parts of a line, and returns a `line-object` describing those parts. It looks at the first character of the line to determine its format; for the interpretation of that character, see the specification's section on encounter lines⁷.

ibp.el

```

;; - - -   a n a l y z e - l i n e   - - -

(defun analyze-line ()
  "Finds the type and cardinal points of the line containing point.

  [ return the line-object for the line containing point ]
"

```

Inside the scope of this `let` are four local variables:

line-kind

A symbol describing what kind of line this is. For possible values, see Section 4.7, “`classify-line`: What kind of line is this?” (p. 12).

line-beg

Position of the beginning of the line.

line-end

Position of the end of the line.

tail-beg

Position of the beginning of the tail portion, if any, or `nil` if the line has no tail.

ibp.el

```

(let (line-kind      ;; Line type code
      line-beg      ;; Line beginning position
      line-end      ;; Line end position
      tail-beg      ;; Line tail beginning position, if any

```

⁷ <http://infohost.nmt.edu/~shipman/z/ibp/doc/iband7/encounter-lines.html>

First we find the beginning and end of the current line. The `save-excursion` function creates a section in which we can move the cursor around freely, but its original position is restored after that section. The `beginning-of-line` function moves the cursor to the beginning of the line, and then we use `setq` to store the cursor position in `line-beg`. A similar jump with the `end-of-line` function is used to set `line-end` to the end position.

ibp.el

```
;; [ line-beg := starting position of the line containing point
;;   line-end := end position of the line containing point ]
(save-excursion          ;; Saves point while executing this block:
  (beginning-of-line)   ;; Move to start of line containing point
  (setq line-beg (point))
  (end-of-line)
  (setq line-end (point)))
```

Next we call a function to look at the first character of the line and see what kind it is; see Section 4.7, “`classify-line`: What kind of line is this?” (p. 12).

ibp.el

```
;; [ line-kind := a symbol for the type of the line
;;   containing point ]
(setq line-kind (classify-line line-beg line-end))
```

Now we have to set the value of `tail-beg`. If the current line isn't an encounter line, `classify-line` returns the symbol `'non-trans`; in this case, we set `tail-beg` to `nil` to signify that the line has no tail. For encounter lines, we set `tail-beg` to `line-beg` (the position of the start of the line) plus the length of the head section for a line of this kind, which is computed by Section 4.8, “`line-head-length`: How long is the head of this line?” (p. 14).

ibp.el

```
;; [ if line-kind is 'non-trans ->
;;   tail-beg := nil
;;   else ->
;;   tail-beg := line-beg + (head length for lines of
;;                           type line-kind) ]
(setq tail-beg
  (if (eq line-kind 'non-trans)
      nil
      (+ line-beg (line-head-length line-kind))))
```

Now we have everything we need to build a `line-object` and return it to the caller.

ibp.el

```
;; [ return a line-object with .kind=line-kind, .beg=line-beg,
;;   .end=line-end, and .tail=tail-beg ]
(line-object line-kind line-beg line-end tail-beg))
```

4.7. `classify-line`: What kind of line is this?

This function looks at the first character of a line and returns a symbol describing what kind of line it is. The return value will be one of:

- `'non-trans`: Not a transaction record; empty, a page header, or other non-encounter line.
- `'lost-destroyed`: Lost or destroyed band line.
- `'short-head`: An encounter line with a three-character head: unbanded or new band.

- 'short-recap: A recap without the r encounter code; the head is nine characters long.
- 'long-head: A recap or long-new line; the head is ten characters.

Here's the prologue:

ibp.el

```
;; - - - c l a s s i f y - l i n e - - -

(defun classify-line (beg end)
  "Examines the given line and returns a symbol describing its type.

Return values:
  'non-trans           Not a transaction record (empty, @, or #)
  'lost-destroyed     Lost or destroyed band transaction
  'short-head         Transaction with 3-character head: u or n
  'short-recap        Recap with no prefix, 9-character head
  'long-head          Transaction with 10-character head: r or g
  -----"
```

First we start a `let` scope, and within that scope, define a variable `line-text` containing the text of the line.

ibp.el

```
(let ((line-text (buffer-substring beg end)))
```

If the line is empty, it's a non-transaction line, and we are done.

ibp.el

```
(if (string= line-text "")
    'non-trans
```

Now that we know there's at least one character on the line, look at it, start another `let` scope, and set variable `prefix` to that first character.

ibp.el

```
(let ((prefix (string-to-char (substring line-text 0 1))))
```

The classification of the line type is done in a `cond`, which contains a series of tests, each with a corresponding result. For example, the second line here returns the `'short-head` symbol if `prefix` is equal to the character `n`. The expression `"?n"` means the character `"n"`.

ibp.el

```
(cond
  ((= prefix ?n) 'short-head)           ;; New band (short form)
  ((= prefix ?u) 'short-head)           ;; Unbanded
```

If the first character is a space, this is a recap line without the encounter code.

ibp.el

```
((= prefix ? ) 'short-recap)           ;; Short recaps start
w/space...
```

The next case is somewhat more complex. A short recap line can start with any character that can legally occur in a band number. This includes digits and question mark. (Letters can also occur in band numbers, but never as the first character.)

ibp.el

```
((and (<= ?0 prefix)                   ;; ...or a digit
      (<= prefix ?9)) 'short-recap)
```

```
((= prefix ??)                ;; ...or a question mark
 'short-recap)
```

The remaining line types are straightforwardly related to encounter codes.

ibp.el

```
((= prefix ?r) 'long-head)      ;; Recap with 'r' prefix
((= prefix ?c) 'long-head)      ;; Recap, changed band
((= prefix ?a) 'long-head)      ;; Recap, double-banded
((= prefix ?g) 'long-head)      ;; New band (long form)
((= prefix ?l) 'lost-destroyed) ;; Lost band
((= prefix ?d) 'lost-destroyed) ;; Destroyed band
```

If all the above conditions fail, we'll just call it a non-transaction line.

ibp.el

```
(t 'non-trans))))))
```

4.8. line-head-length: How long is the head of this line?

This function returns the length of the head portion of a line, given a line-type symbol such as 'long-head.

ibp.el

```
;; - - -   l i n e - h e a d - l e n g t h   - - -

(defun line-head-length (kind)
  "Determines the length of the head part of a line of a given type.

  [ if (line is a line-object) ->
    return the head length in characters for line's type ]
  "
  (cond ((eq kind 'lost-destroyed) 3)
        ((eq kind 'short-head) 3)
        ((eq kind 'short-recap) 9)
        ((eq kind 'long-head) 10)
        (t nil)))      ;; Not defined except for transaction lines
```

4.9. bracket-field: What field contains a given position?

Given a `line-object` and a position within that line, this function tries to find the field containing that position.

There are three cases:

- If the position is before the line's tail, we return a `field-object` describing the head.
- If the position is within the tail, we return a `field-object` describing whichever field contains that position.
- If the position is past the tail, we return `nil`.

This function just checks for the first case. If the position isn't before the tail, we call Section 4.10, "bracket-tail-field: What tail field contains the cursor?" (p. 15).

```
;; - - - b r a c k e t - f i e l d - - -

(defun bracket-field (line p)
  "Find the field containing a position p within a line-object line

  [ if (line is a line-object for a line with a tail)
    and (p is a position within line) ->
    if p is in the head part of line ->
      return a field-object representing the head part, with nil filler

    else if p is in a tail field of line ->
      return a field-object representing that field
    else ->
      return nil ]

  -----"
  (if (< p (line-tail line))
      (field-object (line-beg line) (line-tail line) nil)
      (bracket-tail-field line)))
```

4.10. bracket-tail-field: What tail field contains the cursor?

This function tries to find which tail field contains the cursor, and returns a `field-object` if successful. If the cursor is not within any tail field, it returns `nil`.

This function assumes that `field-def-list` is globally defined as a list of `field-def` objects that describe the format of the tail fields.

```
;; - - - b r a c k e t - t a i l - f i e l d - - -

(defun bracket-tail-field (line)
  "Finds the tail field containing point, if any.

  [ if (line is a line-object for the line containing point)
    and (line is a type that has a tail) ->
    if point is within a tail field ->
      return a field-object describing that field
    else ->
      return nil ]

  -----"
```

First we open a `let` scope, and define some local variables:

flag

Set initially to `'scanning`, this variable is changed to `'found` if we found the correct field, or `'not-found` if we go past the end of the line.

fieldx

Points to the current element of `field-def-list`.

n-fields

Points to the last+1 element of `field-def-list`.

field-def

Holds the current `field-def` object from `field-def-list`.

f-beg

Holds the position of the beginning of the current field.

f-end

Holds the last+1 position of the current field.

f-len

Holds the length of the current field.

ibp.el

```
(let (flag                ;; Changing this value exits the while loop
      fieldx              ;; Indexes field-def-list
      n-fields            ;; Index of last element of field-def-list
      field-def           ;; Holds each field-def object in turn
      f-beg               ;; Walks the start columns of each field
      f-end               ;; End of the current field
      f-len)              ;; Length of the current field
```

We set up initial values before walking the record.

ibp.el

```
;; [ flag      := 'scanning
  ;;   fieldx   := 0
  ;;   n-fields := index of last element of field-def-list
  ;;   f-beg    := location of tail of line ]
(setq flag 'scanning)
(setq fieldx 0)
(setq n-fields (length field-def-list))
(setq f-beg (line-tail line))
```

The while loop runs until the value of `flag` is changed to something other than `'scanning`.

ibp.el

```
;; [ if point is within a field whose length is given in
  ;;   elements fieldx through (n-fields - 1) of field-def-list ->
  ;;   flag := 'found
  ;;   f-beg := position of the start of that field
  ;;   f-end := position of the end of that field
  ;;   else ->
  ;;     flag := 'not-found
  ;;     f-beg := anything
  ;;     f-end := anything ]
(while (eq flag 'scanning)
```

Here is the intended function for the body of this loop:

ibp.el

```
;; [ if fieldx >= n-fields ->
  ;;   flag := 'not-found
  ;;   else if point is within a field starting at f-beg and having
  ;;   length field-lengths[fieldx] ->
  ;;     flag := 'found
  ;;   else ->
  ;;     f-beg := f-beg + field-def-list[fieldx].len
  ;;     fieldx := fieldx + 1 ]
```

If `fieldx` has exceeded the number of fields in `field-def-list`, set `flag` to `'not-found`, and we are done.

ibp.el

```
(if (>= fieldx n-fields)
    (setq flag 'not-found))
```

The `progn` construct executes all the functions inside it, and returns the value of the last one. First we set up the values of the loop variables by extracting them from the `fieldx`th element of `field-def-list`.

ibp.el

```
(progn                ;; The field exists, is point in it?
  (setq field-def (elt field-def-list fieldx))
  (setq f-len (field-def-len field-def))    ;; Get the length...
  (setq f-end (+ f-beg f-len))             ;; ...and end of next
field)
```

If the cursor is at or beyond `f-beg`, and it is before `f-end`, we have found the field containing the cursor; we set `flag` to `'found` so the loop will terminate successfully.

ibp.el

```
(if (and (>= (point) f-beg)    ;; Is f-beg <= point < f-end?
        (< (point) f-end))
    (setq flag 'found)        ;; Yes, found it)
```

If the cursor isn't in the current field, move `f-beg` to the end of the field, increment `fieldx`, and go around the loop again.

ibp.el

```
(progn                ;; No, keep looking
  (setq f-beg f-end)
  (setq fieldx (1+ fieldx))))))
```

If the loop terminated unsuccessfully, `flag` will now be `'not-found`, so we should return `nil` to signify that we couldn't find the cursor's field. If it was successful, we package up a `field-object` made from the current field beginning and end positions, along with the default field content from `field-def-list`, and return that to the caller.

ibp.el

```
;; [ if flag is 'not-found ->
;;   return nil
;; else ->
;;   return a field-object whose .beg=f-beg, .end=f-end, and
;;   .filler=field-def-list[fieldx].filler ]
(if (eq flag 'not-found)
    nil
    (field-object f-beg f-end
                  (field-def-filler (elt field-def-list fieldx)))))
```

4.11. `field-fill`: Move to the end of a field

This function always moves the cursor to the end of the current field. It will also fill out the field's contents if it is incomplete.

ibp.el

```
;; - - -   f i e l d - f i l l   - - -
```

```
(defun field-fill (field line)
  "If field is not full, fill it.  In any case, move to end of field.

   [ if (line is a line-object with a tail) ->
     if line.end is inside field ->
       buffer := buffer with field-fill (field, line.end,
                                         field.filler) appended after line.end
       point  := field.end
     else ->
       point  := field.end ]
-----"
```

First we open a `let` scope and define some local variables:

fill-size

If the field is too short, this is the number of characters we have to add to fill it up.

fill-string

If the field is too short, this is the actual content we need to add to fill it up.

field-off

If the field is too short, this is the distance between the start of the field and the end of its current contents.

field-size

Size of the field if full.

ibp.el

```
(let (fill-size      ;; Size of fill string to be inserted
      fill-string    ;; Fill string to be inserted
      field-off      ;; Offset within the field to insertion point
      field-size)    ;; Size of the field
```

If the end of the line is past the end of the field, all we have to do is jump the cursor to the first character after the field, and we're done.

ibp.el

```
(if (>= (line-end line) (field-end field))
    (goto-char (field-end field)))
```

At this point, we know that the field is not full, so we need to fill it out. We compute the field's size, the offset to the point where we need to add filler content, and the size of the filler content.

ibp.el

```
(progn
  ;; [ field-size := size of field
  ;;   field-off  := line.end - field.beg
  ;;   fill-size  := field.end - line.end ]
  (setq field-size (- (field-end field) (field-beg field)))
  (setq field-off  (- (line-end line) (field-beg field)))
  (setq fill-size  (- (field-end field) (line-end line)))
```

If the field's definition doesn't specify any particular filler content, we make up a string of spaces. Otherwise, we extract its filler content starting at position `field-off`. In either case, the filler content to be added is left in `fill-string`.

```

;; [ if field.filler is nil ->
;;   fill-string := a string of fill-size blanks
;; else ->
;;   fill-string := field.filler[field.off:field-size] ]
(setq fill-string
      (if (null (field-filler field))
          (make-string fill-size ?)
          (substring (field-filler field) field-off field-size)))

```

All that remains is to jump the cursor to the end of the line and append `fill-string`. The `insert` function leaves the cursor at the end of the inserted content, which is where we want it.

```

;; [ buffer := buffer with fill-string appended at line.end
;;   point := field.end ]
(goto-char (line-end line))
(insert fill-string))))

```

4.12. `ibp-ditto`: Field duplication function

This function duplicates one or more fields from a previous line. As in Section 4.4, “`ibp-tab`: Tab function” (p. 9), this function may take a repeat count from `C-u`; the default repeat count is 1. It iterates that many times calling Section 4.13, “`ibp-ditto-once`: Duplicate one field” (p. 19).

```

;; - - - i b p - d i t t o - - -

(defun ibp-ditto (count)
  "Field duplication function for IBP data entry; supports repeat count."
  (interactive "p")
  (while (> count 0)
    (ibp-ditto-once)
    (setq count (1- count))))

```

4.13. `ibp-ditto-once`: Duplicate one field

This function first figures out which field we're in on the current line, and then it tries to find the last previous encounter line that has that same field, and duplicates that value onto the current line.

```

;; - - - i b p - d i t t o - o n c e - - -

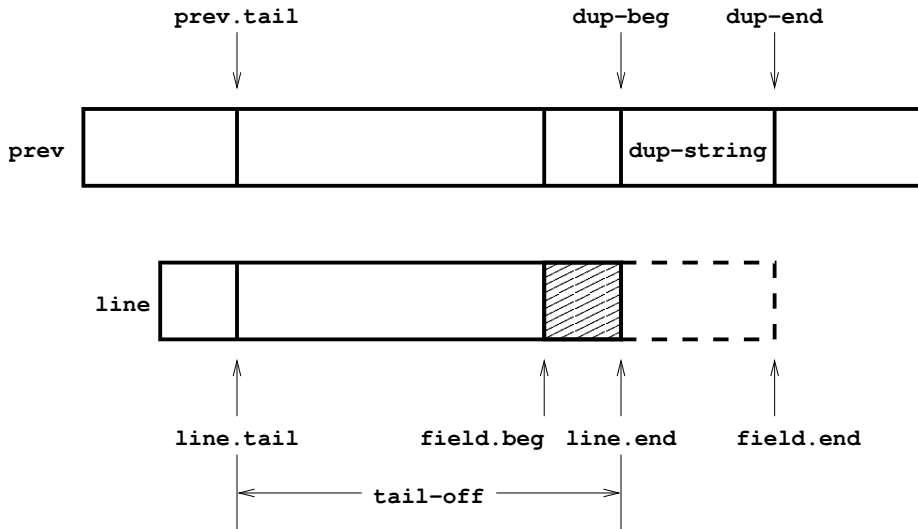
(defun ibp-ditto-once ()
  "Duplicate a field from the corresponding field of the previous tail.

 [ if there are no previous lines with tails ->
   error/exit
 else if point is not at end of line, or beyond all tail fields ->
   error/exit
 else ->
   buffer := buffer with text appended after point
            copied from the corresponding tail position of
            the last previous line with a tail

```

```
point := point advanced to the end of the field containing
point ]
```

Here is an illustration of the process.



The previous line is the top box in this figure, and its layout is described in a `line-object` named `prev`. The current line's layout is described by a `line-object` named `line`. Note that the previous line's head portion may be a different size than the head of the current line, e.g., an N-type encounter line can ditto a field from a G-type line.

The hatched portion represents content in the current field already entered, if any. Variables `dup-beg` and `dup-end` bracket the portion of the previous line that needs to be copied to the current line. Variable `dup-string` will contain the actual text to be copied.

First we open a `let` scope and define the local variables. These variables are as illustrated above.

ibp.el

```
(let (line          ;; line-object for the line containing point
      prev         ;; line-object for the last previous line with a
tail
      field        ;; field-object for the field containing point
      dup-beg      ;; Start position of string to be duplicated
      dup-end      ;; End position of string to be duplicated
      dup-len      ;; Length of string to be duplicated
      dup-string   ;; String to be duplicated
      tail-off)   ;; Offset relative to tail of point
```

First we find the parts of the current line using Section 4.6, “analyze-line: Where are the parts of the current line?” (p. 11).

ibp.el

```
;; [ line := a line-object representing the line
;;         containing point ]
(setq line (analyze-line))
```

Duplication is valid only at the end of the line. It is not valid except on encounter lines.

```

;; [ if (line.kind is 'non-trans)
;;   or (point is not at end of line) ->
;;     error/exit
;; else -> I ]
(if (or (eq (line-kind line) 'non-trans)
        (/= (point) (line-end line)))
    (error "Duplication is valid only at the end of a transaction
line."))

```

Next we need to find the line from which we are copying. One nice feature of this process is that it searches back through the buffer until it finds a transaction line; a `line-object` representing that line is stored in `prev`. This searching is done by Section 4.14, “`find-prev-transaction`: Find the last preceding line with a tail” (p. 22); this could fail, in which case that function returns `null`.

```

;; [ if there is at least one line with a tail preceding the line
;;   containing point ->
;;   prev := a line-object representing that line
;; else -> error/exit ]
(setq prev (find-prev-transaction))
(if (null prev)
    (error "No previous line to duplicate."))

```

Next we need to find the field in the current line containing the cursor, and set `field` to a `field-object` describing that field. This is done by Section 4.9, “`bracket-field`: What field contains a given position?” (p. 14); if the cursor is beyond all the tail fields, that function returns `null`.

```

;; [ if point is in the head part of line ->
;;   field := a field-object representing the head, with nil filler

;;   if point is in a tail field ->
;;   field := a field-object representing that field
;;   if point is beyond all tail fields ->
;;   error/exit ]
(setq field (bracket-field line (point)))
(if (null field)
    (error "You are beyond the fields we know."))

```

Now we want to find the part of `prev` corresponding to the field we are duplicating, and set `dup-string` to the part to be copied. First we compute `tail-off`, the distance into the tail where we want to start copying. Then we bracket the corresponding field in `prev` between `dup-beg` and `dup-end`. If `dup-end` is past the end of the previous line, that's an error.

```

;; [ if prev does not have characters corresponding to [line.end:
;;   field.end] ->
;;   error/exit
;; else ->
;;   dup-string := characters from prev whose position relative to

;;                   prev's tail correspond to characters [line.end:

;;                   field.end] ]
(setq tail-off

```

```

        (- (line-end line) (line-tail line)))
      (setq dup-beg      (+ (line-tail prev) tail-off))
      (setq dup-len     (- (field-end field) (line-end line)))
      (setq dup-end     (+ dup-beg dup-len))
      (if (> dup-end (line-end prev))
          (error "Previous line is too short to duplicate."))
      (setq dup-string (buffer-substring dup-beg dup-end))

```

All that remains is to insert the duplicated text before the cursor. The cursor is left at the end of the insertion.

ibp.el

```

;; [ buffer := buffer with dup-string inserted before point ]
(insert dup-string))

```

4.14. find-prev-transaction: Find the last preceding line with a tail

This function searches backwards from the current line until it finds another encounter line.

ibp.el

```

;; - - -   f i n d - p r e v - t r a n s a c t i o n   - - -

(defun find-prev-transaction ()
  "Search backward from the current line to find the last line with a tail.

  [ if there is at least one line with a tail preceding the line
    containing point ->
    prev := a line-object representing that line
    else -> return nil ]"

```

Local variables include:

line

A line-object pointing to the line currently being inspected.

flag

Initially set to 'scanning, this variable is set to 'found when the search is successful, so that the loop terminates. If the search terminates because we backed up all the way to the start of the buffer, it is set to 'not-found.

ibp.el

```

(let (line      ;; line-object for each line we search
      flag)    ;; Set to 'found or 'not-found to terminate the loop

```

The *emacs* `save-excursion` function creates a block in which we can move the cursor around arbitrarily, but its original position is restored when the block exits.

ibp.el

```

(save-excursion      ;; Save point while executing this block

```

Set the initial value of `flag`, and then jump the cursor to the beginning of the current line.

ibp.el

```

;; [ flag := 'scanning
;;   point := beginning of line containing point ]

```

```
(setq flag 'scanning)
(beginning-of-line)
```

This while loop searches backwards until the flag is set to 'found.

ibp.el

```
;; [ if there is a line before point that has a tail ->
;;   flag   := 'found
;;   point  := the beginning of the last such line
;;   line   := a line-object representing that line
;; else ->
;;   flag   := 'not-found
;;   point  := anything
;;   line   := anything ]
(while (eq flag 'scanning)
```

In the body of the loop, we first test to see if we are all the way back to the beginning of the buffer. If so, it's an error.

ibp.el

```
;; [ if point is at the start of the buffer ->
;;   flag := 'not-found
;; else if the line before point has no tail ->
;;   point := beginning of line before line containing point
;; else ->
;;   flag := 'found
;;   line := a line-object representing the line before
;;           the line containing point ]
(if (= (point) (point-min))      ;; Beginning of buffer?
    (setq flag 'not-found)      ;; Yes, fail
```

The beginning-of-line function with argument 0 backs up one line.

ibp.el

```
(progn
  (beginning-of-line 0)          ;; Move to previous line
```

See what kind of line we're sitting on now by calling Section 4.6, "analyze-line: Where are the parts of the current line?" (p. 11).

ibp.el

```
(setq line (analyze-line))      ;; Make a line-object from it
```

If the line has a tail, we're done. This check is performed by the predicate Section 4.17, "line-has-tail-p: Does this line have a tail?" (p. 25).

ibp.el

```
(if (line-has-tail-p line)      ;; Does it have a tail?
    (setq flag 'found))))))    ;; Yes, succeed
```

That completes the while loop. We examine flag to see if the search was successful. If it has the value 'found, we return line to signify success. If flag is 'not-found, we return nil to signify failure.

ibp.el

```
;; [ if flag is 'not-found ->
;;   return nil
;; else ->
;;   return line ]
(if (eq flag 'not-found)
```

```
nil
line)))
```

4.15. field-def-object: Field definition object

The `field-def-object` represents the definition of one field in the record tail.

The “constructor” for a field definition object has this syntax:

```
(field-def-object len filler)
```

where *len* is the field's length in characters, and *filler* is the default content.

The accessor functions are:

(field-def-len *F*)

Returns the field's length.

(field-filler *F*)

Returns the field's default content.

Here is the constructor. It takes the two arguments and assembles them into an *emacs* vector object (basically a linear array) with the `vector` function.

ibp.el

```
;; - - - - - c l a s s   f i e l d - d e f - o b j e c t   - - - - -

(defun field-def-object (len filler)
  "Constructor for a field-def-object, representing one field definition

  [ if (len is the length of a field as a positive integer)
    and (filler is the default content of the field, or nil if
        its default content is blank) ->
    return a field-def-object representing those values ]

  "
  (vector len filler))
```

There are two accessor functions. They access the attributes of the object by position: the `len` attribute is at position 0, and the `filler` attribute is at position 1. The `elt` function is a built-in function that takes two arguments, a sequence and a position, and returns the element of that sequence at that position.

ibp.el

```
(defun field-def-len (field-def) (elt field-def 0))
(defun field-def-filler (field-def) (elt field-def 1))
```

4.16. Line-object: Represents one line

In order to implement the `tab` and `duplicate` function, we need to break a line down into its parts. The `line-object` pseudo-object has this function.

The constructor has this calling sequence:

ibp.el

```
(line-object kind beg end tail)
```

kind

A line type symbol, as returned by `classify-line`.

beg

Position of the start of the line, as an *emacs* buffer position.

end

Position of the end of the line.

tail

If the line has a tail, this is the position of the start of the tail. If the line has no tail (e.g., a lost-band record), this value is `nil`.

Here is the constructor.

ibp.el

```
;; - - - - - c l a s s   l i n e - o b j e c t   - - - - -

(defun line-object (kind beg end tail)
  "Constructor for a line object, representing the parts of a line.

   [ if (kind is a line type symbol as returned by classify-line)
     and (beg is the position of the start of the line)
     and (end is the position of the end of the line)
     and (tail is the position of the start of the tail part,
         or nil if this is not a tail-type line) ->
     return a line-object representing those values ]
"
  (vector kind beg end tail))
```

There are four accessor functions that retrieve the attributes of this object.

ibp.el

```
(defun line-kind (line-object) (elt line-object 0))
(defun line-beg (line-object) (elt line-object 1))
(defun line-end (line-object) (elt line-object 2))
(defun line-tail (line-object) (elt line-object 3))
```

4.17. `line-has-tail-p`: Does this line have a tail?

A predicate used to test whether a `line-object` describes a line that has a tail. Only three line types don't have a tail: lost-band encounters, destroyed-band encounters, and non-transaction lines (such as page header lines).

ibp.el

```
(defun line-has-tail-p (line-object)
  "Predicate: is this line one of the types that has a tail?"
  "
  (if (or (eq (line-kind line-object) 'non-trans)
         (eq (line-kind line-object) 'lost-destroyed))
      nil
      t))
```

4.18. field-object: Location of an actual field

An object of this type describes the location of an actual field in an actual record (unlike a `field-def-object`, which talks about field positions within an arbitrary tail). It has three attributes:

beg

Position of the first character in the field, as an *emacs* position type.

end

Position just past the last character in the field. If the field is incomplete, this is the position just after the completed part.

filler

For fields whose default content is spaces, this is `nil`; otherwise it is the default content from the appropriate `field-def-object`.

Here's the definition:

ibp.el

```
;; - - - - - c l a s s   f i e l d - o b j e c t   - - - - -
(defun field-object (beg end filler)
  "Constructor for a field object, representing the location of a field.

  [ if (beg is the position of the start of a field)
    and (end is the position of the end of a field, which may be
    beyond the end of its line)
    and (filler is the default contents of the field, or nil if
    the default is blank) ->
    return a field-object representing those values ]
  "
  (vector beg end filler))
```

These accessor functions extract the components of the object:

ibp.el

```
(defun field-beg (field-object) (elt field-object 0))
(defun field-end (field-object) (elt field-object 1))
(defun field-filler (field-object) (elt field-object 2))
```