IBP banding data system:  
*emacs* customizations

Version 7.02

John W. Shipman

2013-03-30 14:19

Abstract

Describes the implementation of a package for the *emacs* text editor to make entry of bird banding data more efficient.

This publication is available in Web form\(^1\) and also as a PDF document\(^2\). Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

1. Introduction ............................................................................................................................ 2
2. Survey of modules in this package ............................................................................................ 2
3. Notes on key bindings .............................................................................................................. 3
4. Top-level modules ................................................................................................................... 3
   4.1. maps2013m.el: Multiple-station sets, MAPS 2013 protocol ............................................. 3
   4.2. maps2013s.el: Single-station sets, MAPS 2013 protocol ............................................. 4
   4.3. maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol ............................ 5
   4.5. maps1998m.el: Multi-station sets, MAPS 1998 protocol ........................................... 6
   4.6. maps1998s.el: Single-station sets, MAPS 1998 protocol ........................................... 6
   4.7. maws2007m.el: Multi-station sets, MAWS protocol ................................................ 7
   4.8. maws2007s.el: Single-station sets, MAWS protocol ................................................ 8
   4.9. maws2004m.el: Multi-station sets, MAWS protocol ................................................ 8
   4.10. maws2004s.el: Single-station sets, MAWS protocol .............................................. 9
5. ibp.el: Common logic ........................................................................................................... 9
   5.1. Introduction to ibp.el ............................................................................................... 10
   5.2. The pseudo-object convention ............................................................................... 10
   5.3. ibp.el: Prologue .................................................................................................... 10
   5.4. ibp-tab: Tab function ............................................................................................ 11
   5.5. ibp-tab-once: Single tab function ....................................................................... 11
   5.6. ibp-analyze-line: Where are the parts of the current line? ................................... 13
   5.7. ibp-classify-line: What kind of line is this? ......................................................... 14
   5.8. ibp-line-head-length: How long is the head of this line? .................................... 16
   5.9. ibp-bracket-field: What field contains a given position? .................................... 16
   5.10. ibp-bracket-tail-field: What tail field contains the cursor? ............................... 17
   5.11. ibp-field-fill: Move to the end of a field ............................................................ 19
   5.12. ibp-ditto: Field duplication function .................................................................... 21
   5.13. ibp-ditto-once: Duplicate one field .................................................................... 21

\(^1\) [http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/](http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/)
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 specification. 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 page.

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.

Recommended references for emacs and e-lisp:


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.

- maps2013m.el: Section 4.1, “maps2013m.el: Multiple-station sets, MAPS 2013 protocol” (p. 3)
- maps2013s.el: Section 4.2, “maps2013s.el: Single-station sets, MAPS 2013 protocol” (p. 4)
- maps2004m.el: Section 4.3, “maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol” (p. 5)
- maps2004s.el: Section 4.4, “maps2004s.el: Single-station sets, MAPS 2004/2006 protocol” (p. 5)
- maps1998m.el: Section 4.5, “maps1998m.el: Multi-station sets, MAPS 1998 protocol” (p. 6)
- maps1998s.el: Section 4.6, “maps1998s.el: Single-station sets, MAPS 1998 protocol” (p. 6)
3. Notes on key bindings

The extra *emacs* functions we add are bound to two keys:

- The *tab* key is bound to the `ibp-tab` function. It skips to the next field and supplies default content. This binding is accomplished by this code:

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

  The key’s name is described here as *Control-I*, which is another way of describing the ASCII HT (horizontal tab) character.

- The `ibp-ditto` function duplicates the current field using the last previous line that contains that field. The author’s bindings have varied over time, but the important thing is to have a binding that is easy to reach (because this key is heavily used) and works the same way on multiple platforms.

  The current binding uses *Control* plus the key that contains hyphen and underbar. As it happens, *emacs* running in no-window mode (option `-nw`) transmits this as *Control-_**, while in a separate window it transmits as *Control--*. So we’ll bind both those combinations like this:

  ```lisp
  (global-set-key [?\C-_] 'ibp-ditto)
  (global-set-key [?\C--] 'ibp-ditto)
  ```

  The first argument to `global-set-key` in these cases uses *emacs*’s vector notation.

4. Top-level modules

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

4.1. maps2013m.el: Multiple-station sets, MAPS 2013 protocol

This module sets up a field structure for a multi-station set in the 2013 MAPS protocol.

```lisp
;; maps2013m.el: Set up field values and keybindings for
;; MAPS 2013 protocol, multiple stations.
;; To use, run emacs with this command line option:
;; emacs -l maps2013m filename
;;
```

---

12 http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/maws2007m.el
13 http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/maws2007s.el
14 http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/maws2004m.el
15 http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/maws2004s.el
16 http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/ibp.el
First we load the common functions used by all the top-level modules.

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

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

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

Finally, we bind the keys are discussed in Section 3, “Notes on key bindings” (p. 3).

```lisp
(global-set-key \C-i 'ibp-tab)
(global-set-key [?\C-_] 'ibp-ditto)
(global-set-key [?\C--] 'ibp-ditto)
```

### 4.2. maps2013s.el: Single-station sets, MAPS 2013 protocol

Exactly like maps2013m.el except that the station code is omitted.

```lisp
;; maps2013s.el: Set up field values and keybindings for
;; MAPS 2013 protocol, single stations.
;; To use, run emacs with this command line option:
;; emacs -l maps2013s filename
;;
;(load "ibp.el")
(setq ibp-field-list
  (vector
    (ibp-field-def 4 "????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil) ;; Skull through fat
    (ibp-field-def 4 nil) ;; Body molt through JP
    (ibp-field-def 3 nil) ;; WRP molt cycle codes
    (ibp-field-def 4 nil) ;; Pri.cov/Sec.cov/Pri/Sec
    (ibp-field-def 4 nil) ;; Tert/Rect/Body plum/NF
    (ibp-field-def 3 nil) ;; Wing length
    (ibp-field-def 4 nil) ;; Weight
    (ibp-field-def 5 "3????") ;; Status, date
```

---

4 Zoological Data Processing

IBP emacs customizations

Zoological Data Processing
4.3. maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol

This module sets up a field structure for a multi-station set in the 2006 MAPS protocol. Everything here is the same except that the WRP molt cycle codes column is missing.

```
;; 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
;;
;(load "ibp.el")
(setq ibp-field-list
  (vector
    (ibp-field-def 4 "????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil) ;; Skull through fat
    (ibp-field-def 4 nil) ;; Body molt through JP
    (ibp-field-def 4 nil) ;; Pri.cov/Sec.cov/Pri/Sec
    (ibp-field-def 4 nil) ;; Tert/Rect/Body plum/NF
    (ibp-field-def 3 nil) ;; Wing length
    (ibp-field-def 4 nil) ;; Weight
    (ibp-field-def 5 "????") ;; Status, date
    (ibp-field-def 3 nil) ;; Time
    (ibp-field-def 4 nil) ;; Station code
    (ibp-field-def 2 nil))) ;; Net
(global-set-key \C-i 'ibp-tab)
(global-set-key [?\C-_] 'ibp-ditto)
(global-set-key [?\C--] 'ibp-ditto)
```

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

Similar to Section 4.3, “maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol” (p. 5), 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 ibp-field-list
  (vector
    (ibp-field-def 4 "????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil) ;; Skull through fat
    (ibp-field-def 4 nil) ;; Body molt through JP
    (ibp-field-def 4 nil) ;; Pri.cov/Sec.cov/Pri/Sec
    (ibp-field-def 4 nil) ;; Tert/Rect/Body plum/NF
    (ibp-field-def 3 nil) ;; Wing length
    (ibp-field-def 4 nil) ;; Weight
    (ibp-field-def 5 "????") ;; Status, date
    (ibp-field-def 3 nil) ;; Time
    (ibp-field-def 4 nil) ;; Station code
    (ibp-field-def 2 nil))) ;; Net
```
4.5. maps1998m.el: Multi-station sets, MAPS 1998 protocol

Like Section 4.3, “maps2004m.el: Multiple-station sets, MAPS 2004/2006 protocol” (p. 5), but uses the pre-2004 field layout, with ten micro-aging fields.

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

Like Section 4.5, “maps1998m.el: Multi-station sets, MAPS 1998 protocol” (p. 6), but for single-station sets with no station codes.
4.7. 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.

```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 ibp-field-list
  (vector
    (ibp-field-def 6 "??????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil ) ;; Skull through fat
    (ibp-field-def 4 nil ) ;; Body molt through JP
    (ibp-field-def 4 nil ) ;; Micro-aging: 8 fields...
    (ibp-field-def 4 nil ) ;; ...(divided 2/4/2 on the sheet)
    (ibp-field-def 3 nil ) ;; Wing length
    (ibp-field-def 4 nil ) ;; Mass (used to be weight)
    (ibp-field-def 5 "??????") ;; Status, date
    (ibp-field-def 3 nil ) ;; Time
    (ibp-field-def 2 nil ) ) ) ;; Net

(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-_] 'ibp-ditto)
(global-set-key [?\C--] 'ibp-ditto)
```
4.8. **maws2007s.el**: Single-station sets, MAWS protocol

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

```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 ibp-field-list
  (vector
    (ibp-field-def 6 "??????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil ) ;; Skull through fat
    (ibp-field-def 4 nil ) ;; Body molt through JP
    (ibp-field-def 4 nil ) ;; Micro-aging: 8 fields...
    (ibp-field-def 4 nil ) ;; ...(divided 2/4/2 on the sheet)
    (ibp-field-def 3 nil ) ;; Wing length
    (ibp-field-def 4 nil ) ;; Mass (used to be weight)
    (ibp-field-def 5 "3????? ") ;; Status, date
    (ibp-field-def 3 nil ) ;; Time
    (ibp-field-def 4 nil ) ) ) ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-\] 'ibp-ditto)
(global-set-key [?\C--\] 'ibp-ditto)
```

4.9. **maws2004m.el**: Multi-station sets, MAWS protocol

MAWS 2004 multi-station protocol.

```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 ibp-field-list
  (vector
    (ibp-field-def 6 "??????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil ) ;; Skull through fat
    (ibp-field-def 4 nil ) ;; Body molt through JP
    (ibp-field-def 4 nil ) ;; Micro-aging: 8 fields...
    (ibp-field-def 4 nil ) ;; ...(divided 2/4/2 on the sheet)
    (ibp-field-def 3 nil ) ;; Wing length
    (ibp-field-def 4 nil ) ;; Mass (used to be weight)
    (ibp-field-def 5 "3????? ") ;; Status, date
    (ibp-field-def 3 nil ) ;; Time
    (ibp-field-def 4 nil ) ) ) ;; Station code
```
4.10. maws2004s.el: Single-station sets, MAWS protocol

MAWS 2004 single-station protocol.

```lisp
;; 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 ibp-field-list
  (vector
    (ibp-field-def 6 "??????") ;; Species code
    (ibp-field-def 3 "0 ") ;; Age, how-aged
    (ibp-field-def 3 "u ") ;; Sex, how-sexed
    (ibp-field-def 4 nil ) ;; Skull through fat
    (ibp-field-def 4 nil ) ;; Body molt through JP
    (ibp-field-def 4 nil ) ;; Micro-aging: 8 fields...
    (ibp-field-def 4 nil ) ;; ...(divided 2/4/2 on the sheet)
    (ibp-field-def 3 nil ) ;; Wing length
    (ibp-field-def 4 nil ) ;; Mass (used to be weight)
    (ibp-field-def 5 "3????" ) ;; Status, date
    (ibp-field-def 3 nil ) ;; Time
    (ibp-field-def 2 nil ) ) ) ;; Net
(global-set-key "\C-i" 'ibp-tab)
(global-set-key [?\C-_] 'ibp-ditto)
(global-set-key [?\C--] 'ibp-ditto)
```

5. 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 `ibp-field-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 4, “Top-level modules” (p. 3).
5.1. Introduction to \texttt{ibp.el}

Some definitions:


\textbf{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.

\textbf{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 specification \cite{ibp}: in that document, the fixed-format part of the record is called the \textit{body}, while the fields following the net field are called the \textit{tail}.

5.2. The pseudo-object convention

The author prefers an object-oriented style in programming generally. In \texttt{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 \textit{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.

For example, if such an object is contained in some \texttt{e-lisp} variable $F$, then the \textit{accessor function} \texttt{(ibp-field-def-len F)} returns the field length.

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

- Section 5.15, “\texttt{ibp-field-def}: Field definition object” (p. 25).
- Section 5.16, “\texttt{ibp-line-object}: Represents one line” (p. 26).
- Section 5.18, “\texttt{ibp-field-object}: Location of an actual field” (p. 27).

5.3. \texttt{ibp.el}: Prologue

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

\begin{Verbatim}
;; ibp.el: Common functions for emacs IBP banding data extensions
;;------------------------------------------------------------------
;; Do not edit this file. It is extracted automatically from the
;; documentation, here:
;;   http://www.nmt.edu/~shipman/z/ibp/doc/iband7/emacs/
;;------------------------------------------------------------------
\end{Verbatim}

\textsuperscript{17} http://www.nmt.edu/~shipman/z/ibp/doc/iband7/ims/
5.4. **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.

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

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

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).

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

5.5. **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.

```lisp
(defvar 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
  ibp-field-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 an `ibp-line-object` representing the current line.
- **field** will hold an `ibp-field-object` representing the last field currently existing on the current line.
We call the \texttt{ibp-analyze-line} function to look at the line and return a \texttt{ibp-line-object} that tells us the position of the major parts of the current 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 5.17, \texttt{ibp-line-has-tail-p}: Does this line have a tail?" (p. 27).

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

- If the cursor is before the tail, we set \texttt{field} to an \texttt{ibp-field-object} representing the head of the line.
- If the cursor is in some field within the tail, we set \texttt{field} to an \texttt{ibp-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 5.9, \texttt{ibp-bracket-field}: What field contains a given position?" (p. 16), which returns \texttt{null} if the cursor is past the tail, or an \texttt{ibp-field-object} otherwise.

If the end of the line is inside \texttt{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 5.11, \texttt{ibp-field-fill}: Move to the end of a field" (p. 19).
5.6. **ibp-analyze-line**: Where are the parts of the current line?

This function locates the major parts of a line, and returns a `ibp-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 lines.

```lisp
(defun ibp-analyze-line ()
  "Finds the type and cardinal points of the line containing point.
   [ return the ibp-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 5.7, “ibp-classify-line: What kind of line is this?” (p. 14).
- **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.

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

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.

```lisp
;; [ line-beg := starting position of the line containing point
;;   line-end := end position of the line containing point ]
(safe-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 5.7, “ibp-classify-line: What kind of line is this?” (p. 14).
Now we have to set the value of `tail-beg`. If the current line isn’t an encounter line, `ibp-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 5.8, “ibp-line-head-length: How long is the head of this line?” (p. 16).

```lisp
(setq tail-beg
  (if (eq line-kind 'non-trans)
      nil
    (+ line-beg (ibp-line-head-length line-kind))))
```

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

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

### 5.7. ibp-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:

```lisp
(defun ibp-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")
First we start a `let` scope, and within that scope, define a variable `line-text` containing the text of the line.

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

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

```

```
(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.

```
(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”.

```
(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.

```
((= 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.)

```
((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.

```
((= 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.

```
(t 'non-trans)))))
```
5.8. **ibp-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’.

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

  [ if (line is a ibp-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
```

5.9. **ibp-bracket-field**: What field contains a given position?

Given an ibp-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 an ibp-field-object describing the head.
- If the position is within the tail, we return an ibp-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 5.10, “ibp-bracket-tail-field: What tail field contains the cursor?” (p. 17).

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

  [ if (line is a ibp-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 an ibp-field-object representing the head part, with nil filler
      else if p is in a tail field of line ->
        return an ibp-field-object representing that field
      else ->
        return nil ]
  "
  (if (< p (ibp-line-tail line))
      (ibp-field-object (ibp-line-beg line) (ibp-line-tail line) nil)
      (ibp-bracket-tail-field line)))
```
5.10. ibp-bracket-tail-field: What tail field contains the cursor?

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

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

```el
;; - - - i b p - b r a c k e t - t a i l - f i e l d - - -
(defun ibp-bracket-tail-field (line)
  "Finds the tail field containing point, if any.

  [ if (line is a ibp-line-object for the line containing point)
    and (line is a type that has a tail) ->
      if point is within a tail field ->
        return an ibp-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 ibp-field-list.

**n-fields**

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

**field-def**

Holds the current field-def object from ibp-field-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.

```el
(let (flag ;; Changing this value exits the while loop
       fieldx ;; Indexes ibp-field-list
       n-fields ;; Index of last element of ibp-field-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.
)
```
The while loop runs until the value of flag is changed to something other than 'scanning.

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

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

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 ibp-field-list.

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.
(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.

(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 an ibp-field-object made from the current field beginning and end positions, along with the default field content from ibp-field-list, and return that to the caller.

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

5.11. ibp-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.

;; - - - i b p - f i e l d - f i l l - - -
(defun ibp-field-fill (field line)
  "If field is not full, fill it. In any case, move to end of field.

  [ if (line is a ibp-line-object with a tail) ->
    if line.end is inside field ->
      buffer := buffer with ibp-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.

Zoological Data Processing    IBP emacs customizations
**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.

```
(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.

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

```
(progn
  ;; [ field-size := size of field
  ;;   field-off := line.end - field.beg
  ;;   fill-size := field.end - line.end ]
  (setq field-size (- (ibp-field-end field) (ibp-field-beg field)))
  (setq field-off (- (ibp-line-end line) (ibp-field-beg field)))
  (setq fill-size (- (ibp-field-end field) (ibp-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 (ibp-field-filler field))
      (make-string fill-size ? )
      (substring (ibp-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 (ibp-line-end line))
  (insert fill-string)))))
```
5.12. **ibp-ditto: Field duplication function**

This function duplicates one or more fields from a previous line. As in Section 5.4, “ibp-tab: Tab function” (p. 11), this function may take a repeat count from C-u; the default repeat count is 1. It iterates that many times calling Section 5.13, “ibp-ditto-once: Duplicate one field” (p. 21).

```
;; - - - ibp-ditto - - -

(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))))
```

5.13. **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.

```
;; - - - ibp-ditto-once - - -

(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 an `ibp-line-object` named `prev`. The current line's layout is described by an `ibp-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.

First we find the parts of the current line using Section 5.6, “`ibp-analyze-line`: Where are the parts of the current line?” (p. 13).

Duplication is valid only at the end of the line. It is not valid except on encounter lines.
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 ibp-line-object representing that line is stored in prev. This searching is done by Section 5.14, “ibp-find-prev-trans: Find the last preceding line with a tail” (p. 24); this could fail, in which case that function returns null.

Next we need to find the field in the current line containing the cursor, and set field to an ibp-field-object describing that field. This is done by Section 5.9, “ibp-bracket-field: What field contains a given position?” (p. 16); if the cursor is beyond all the tail fields, that function returns null.

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

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

### 5.14. `ibp-find-prev-trans`: Find the last preceding line with a tail

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

```el
;; - - - i b p - f i n d - p r e v - t r a n s - - -
(defun ibp-find-prev-trans ()
  "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 := an ibp-line-object representing that line
  else -> return nil ]"

Local variables include:

**line**
An `ibp-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.

```el
(let (line ;; ibp-line-object for each line we search
      flag) ;; Set to 'found or 'not-found to terminate the loop
  (save-excursion ;; Save point while executing this block
      (setq flag 'scanning)
      (beginning-of-line)
    This while loop searches backwards until the flag is set to 'found.

    ;; [ if there is a line before point that has a tail ->
    ;;   flag := 'found
    ;;   point := the beginning of the last such line
    ;;   line := an ibp-line-object representing that line
```
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.

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

(setq line (ibp-analyze-line)) ;; Make ibp-line-object

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

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

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

(setq line (ibp-analyze-line)) ;; Make ibp-line-object
```

If the line has a tail, we’re done. This check is performed by the predicate Section 5.17, “ibp-lines-tail-p: Does this line have a tail?” (p. 27).

```
(if (ibp-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.

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

### 5.15. ibp-field-def: Field definition object

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

The “constructor” for a field definition object has this syntax:
(ibp-field-def len filler)

where \textit{len} is the field's length in characters, and \textit{filler} is the default content.

The accessor functions are:

\textbf{(ibp-field-def-len \textit{F})}

Returns the field's length.

\textbf{(ibp-field-def-filler \textit{F})}

Returns the field's default content.

Here is the constructor. It takes the two arguments and assembles them into an \textit{emacs} vector object (basically a linear array) with the \texttt{vector} function.

\texttt{ibp.el}

\begin{verbatim}
;; - - - - - c l a s s   i b p - f i e l d - d e f
(defun ibp-field-def (len filler)
  "Constructor for a ibp-field-def, 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 ibp-field-def representing those values ]
  "
  (vector len filler))
\end{verbatim}

There are two accessor functions. They access the attributes of the object by position: the \textit{len} attribute is at position 0, and the \textit{filler} attribute is at position 1. The \texttt{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.

\texttt{ibp.el}

\begin{verbatim}
(defun ibp-field-def-len (field-def) (elt field-def 0))
(defun ibp-field-def-filler (field-def) (elt field-def 1))
\end{verbatim}

\subsection*{5.16. ibp-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 \texttt{ibp-line-object} pseudo-object has this function.

The constructor has this calling sequence:

\begin{verbatim}
(ibp-line-object \textit{kind} \textit{beg} \textit{end} \textit{tail})
\end{verbatim}

\textbf{kind}

A line type symbol, as returned by \texttt{ibp-classify-line}.

\textbf{beg}

Position of the start of the line, as an \textit{emacs} buffer position.

\textbf{end}

Position of the end of the line.

\textbf{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 \texttt{nil}.
Here is the constructor.

```lisp
;;------  class  ibp-line-object

(defun ibp-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 ibp-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 ibp-line-object representing those values ]

  (vector kind beg end tail))
```

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

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

### 5.17. ibp-line-has-tail-p: Does this line have a tail?
A predicate used to test whether a ibp-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).

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

### 5.18. ibp-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 ibp-field-def, 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 ibp-field-def.
Here's the definition:

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

(defun ibp-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 an ibp-field-object representing those values ]
  "
  (vector beg end filler))
```

These accessor functions extract the components of the object:

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