```
;===============================================================================
;
; BIT SETS
;
; This module implements sets of integers >= 0.  The sets are represented
; by bit vectors.  A bit vector is just represented here as a list of
; INUMs (32-bit integers that aren't boxed).  The sets may be varying
; length -- all the operations will explicitly extend shorter sets with
; 0s.  We've gone to a little pain to use 32-bit INUMs for two reasons:
; to avoid number consing and to be compatible with future 32-bit Lisps?
;
; *BIT-SET.EMPTY-SET*
;     The empty set (just () ).
;
; (BIT-SET:NEW SIZE)
;     Creates a new empty set of SIZE elements.
;
; (BIT-SET:UNIVERSE SIZE)
;     Creates a new set of SIZE elements, containing all the elements.
;
; (BIT-SET:MEMBER? SET ELEMENT)
;     Returns true if ELEMENT is a member of SET.
;
; (BIT-SET:CONTAINS? SET1 SET2)
;     Returns true if SET1 contains SET2.
;
; (BIT-SET:CHOOSE SET)
;     Returns the smallest element of SET, or NIL if SET is empty.
;
; (BIT-SET:SINGLETON ELEMENT)
;     Creates a new set containing the single ELEMENT.
;
; (BIT-SET:INTERSECTION SET1 SET2 ...)
;     Returns a new set that is the intersection of all the given sets.
;
; (BIT-SET:UNION SET1 SET2 ...)
;     Returns a new set that is the union of all the given sets.
;
; (BIT-SET:UNION1 SET ELEMENT)
;     Returns a new set that is the union of SET and the single ELEMENT.
;
; (BIT-SET:DIFFERENCE SET1 SET2)
;     Returns a new set that contains all elements in SET1 not in SET2.
;
; (BIT-SET:DIFFERENCE1 SET ELEMENT)
;     Returns a new set that is the difference of SET and the single ELEMENT.
;
; (BIT-SET:= SET1 SET2)
;     Returns true if the two sets are equal.
;
; (LOOP (FOR-EACH-BIT-SET-ELEMENT SET ELEMENT) ...
;     Enumerates ELEMENT through each element of the set.  This is a
;     DEF-SIMPLE-LOOP-CLAUSE.
;
; (BIT-SET:SIZE SET)
;     Returns the number of elements in the set.
;
; (BIT-SET:PRINT SET1 [STREAM])
;     Print the set to STREAM (defaults to T) in a compact manner.
;
;===============================================================================

(eval-when (eval compile)
    (:= bits-per-integer   32)
    (:= bits-per-integer-1 31) )

(defvar *bit-set.empty-set* () )

(defun bit-set:new ( size )
    (loop (initial result () )
    (do
        (push result 0) )
    (while (> size #.bits-per-integer) )
    (next size (- size #.bits-per-integer) )
    (result
        (dreverse result) ) ) )


(defun bit-set:universe ( size )
    (loop (initial result () )
    (while (> size #.bits-per-integer) )
    (do
        (push result -1) )
    (next size (- size #.bits-per-integer) )
    (result
        (push result (lsh32 -1 (- #.bits-per-integer size) ) )
        (dreverse result) ) ) )


(defun bit-set:singleton ( element )
    (bit-set:union1 () element) )

(defun bit-set:member? ( set element )
    (loop (for word in set)
    (do
        (if (<= element #.bits-per-integer-1)
            (return
                (!== 0 (logand word
                               (lsh32 1 (- #.bits-per-integer-1 element)))))))
    (next element (- element #.bits-per-integer) )
    (result () ) ) )


(defun bit-set:choose ( set )
    (loop (for word in set)
          (incr bas from 0 by #.bits-per-integer)
    (do
        (if (!== 0 word)
            (return (+ bas (jffo32 word) ) ) ) )
    (result () ) ) )


(defun bit-set:intersection args
    (caseq args
        (0 *bit-set.empty-set*)
        (1 (arg 1) )
        (t
            (loop (initial result       ()
                           next-element 0
                           all-done?    () )
            (do
                (:= all-done? () )
                (:= next-element -1)
                (loop (incr i from 1 to args) (do
                    (if (! (arg i) ) (then
                        (:= all-done? t)
```

```
                    (return () ) ) )
                (:= next-element (logand next-element (pop (arg i) ) ) ) )
            (result
                (push result next-element) ) ) )

        (until all-done?)
        (result
            (dreverse result) ) ) ) ) )


(defun bit-set:union args
    (caseq args
        (0 *bit-set.empty-set*)
        (1 (arg 1) )
        (t
            (loop (initial result       ()
                           next-element 0
                           all-done?    () )
            (do
                (:= all-done? t)
                (:= next-element 0)
                (loop (incr i from 1 to args) (do
                    (if (arg i) (then
                        (:= all-done? () )
                        (:= next-element (logior next-element (pop (arg i) ) ) ) )

                (if all-done?
                    (return (dreverse result) )
                    (push result next-element) ) ) ) ) ) )


(defun bit-set:union1 ( set element )
    (loop (initial result   ()
                   rest-set set
                   word     0)
    (next word (if rest-set
                   (pop rest-set)
                   0) )
    (do
        (if (&& (<= element #.bits-per-integer-1)
                (>= element 0) )
        (then
             (push result
                   (logior word (lsh32 1 (- #.bits-per-integer-1 element)))))
        (else
             (push result word) ) ) )
    (until (&& (! rest-set)
               (<= element #.bits-per-integer-1) ) )
    (next element (- element #.bits-per-integer) )
    (result
        (dreverse result) ) ) )


(defun bit-set:difference ( set1 set2 )
    (loop (initial rest-set1 set1
                   rest-set2 set2
                   result    () )
    (while rest-set1)
    (do
        (if rest-set2 (then
            (push result (logand (pop rest-set1) (lognot (pop rest-set2)))))
        (else
            (push result (pop rest-set1) ) ) ) )
```

3

```
    (result (dreverse result) ) ) )


(defun bit-set:difference1 ( set element )
    (loop (for word in set) (save
        (if (&& (>= element 0)
                (<  element #.bits-per-integer) )
        (then
            (boole 4 word (lsh32 1 (- #.bits-per-integer-1 element) ) ) )
        (else
            word) ) )
    (next element (- element #.bits-per-integer) ) ) )



(defun bit-set:= ( set1 set2 )
    (loop (initial rest-set1 set1
                   rest-set2 set2
                   word1     0
                   word2     0)
    (while (|| rest-set1 rest-set2) )
    (next word1 (if rest-set1
                    (pop rest-set1)
                    0)
          word2 (if rest-set2
                    (pop rest-set2)
                    0) )
    (do
        (if (!== word1 word2)
            (return () ) ) )
    (result t) ) )


(defun bit-set:contains? ( set1 set2 )
    (bit-set:= set1 (bit-set:union set1 set2) ) )


(eval-when (eval compile load)
(def-simple-loop-clause for-each-bit-set-element ( clause )
    (let ( ( (for-each-bit-set-element set var) clause)
           (rest-set (intern (gensym) ) )
           (word     (intern (gensym) ) )
           (bas      (intern (gensym) ) )
           (offset   (intern (gensym) ) ) )
        (if (! (&& (== 3 (length clause) )
                   (litatom var) ) )
            (error (list clause
                         "Invalid syntax in FOR-EACH-BIT-SET-ELEMENT.") ) )
        `( (initial ,rest-set ,set
                    ,word     0
                    ,var      0
                    ,bas      #.(- 0 bits-per-integer)
                    ,offset   32)
           (while
               (loop (do
                   (if (== 32 ,offset) (then
                       (if (! ,rest-set)
                           (return () ) )
                       (pop ,rest-set ,word)
                       (:= ,bas  (+ ,bas  #.bits-per-integer) ) ) )
                   (:= ,offset (jffo32 ,word) )
                   (if (!== #.bits-per-integer ,offset) (then
```

4

```
                    (:= ,word
                        (boole 4 ,word
                                 (lsh32 1 (- #.bits-per-integer-1 ,offset))))
                    (:= ,var (+ ,bas  ,offset) )
                    (return t) ) ) ) ) ) ) ) )
)


(defun bit-set:size ( set )
    (loop (initial size 0)
    (for-each-bit-set-element set element)
    (next size (+ 1 size) )
    (result size) ) )


(defun bit-set:print ( set )
    (msg "{")

    (loop (label print-next-range)
          (initial first-in-range 0
                   last-in-range  0
                   size           (* #.bits-per-integer (length set) )
                   space-needed   () )
    (do
        (loop (while (! (bit-set:member? set first-in-range) ) )
              (next first-in-range (+ 1 first-in-range) )
        (do
            (if (>= first-in-range size)
                (leave print-next-range) ) ) )

        (:= last-in-range first-in-range)
        (loop (next last-in-range (+ 1 last-in-range) )
              (while (< last-in-range size) )
              (while (bit-set:member? set last-in-range) ) )

        (if space-needed
            (msg " ") )
        (:= space-needed t)

        (caseq (- last-in-range first-in-range)
            (1 (msg first-in-range) )
            (2 (msg first-in-range " " (+ -1 last-in-range) ) )
            (t (msg first-in-range ":" (+ -1 last-in-range) ) )))
    (next first-in-range last-in-range) )

    (msg "}")
    () )
```

```
(:= *util.build-module-list* '(

    utilities:vector-map
    utilities:bit-set
    utilities:sharp-sharp
    utilities:options
    utilities:visible-fields
    ) )

(:= *build-module-list* (append *build-module-list* *util.build-module-list*) )
```

```
;=============================================================================
;
; (HUNT PREDICATE DEPTH)
;
; HUNT helps you find forgotton pointers to data structures that aren't
; getting GCed.  It traces out from the value and property list of every
; interned symbol, looking for a value for which (FUNCALL PREDICATE VAL)
; is true.  To prevent looping on circular structures, DEPTH is the maximum
; number of pointers to follow in any one direction from a root symbol.
; (HUNT PRED 1) means look for any values that are the direct value of
; of a global symbol that satisfy PRED.
;
; Currently HUNT knows about lists, symbols, and vectors.
;
;=============================================================================


(declare (special
    *hunt.predicate*
    *hunt.max-depth*
    *hunt.symbol*
    ) )


(defun hunt ( *hunt.predicate* *hunt.max-depth* )
    (mapatoms 'hunt.map-function) )


(defun hunt.map-function ( *hunt.symbol* )
    (catch (hunt.recurse *hunt.symbol* 0)
        hunt.map-function-exit) )


(defun hunt.recurse ( val current-depth )
    (? ( (== val (unbound) )
         () )

       ( (funcall *hunt.predicate* val)
         (msg 0 "Within " *hunt.symbol* t)
         (throw () hunt.map-function-exit) )

       ( t
         (if (< current-depth *hunt.max-depth*) (then
            (++ current-depth)
            (? ( (! val)
                 () )

               ( (== (unbound) val)
                 () )

               ( (litatom val)
                 (if (boundp val) (then
                     (hunt.recurse (eval val) current-depth) ) )

                 (loop (initial rest (plist val) )
                 (while rest)
                 (do
                    (hunt.recurse (cadr rest) current-depth) )
                 (next rest (cddr rest) ) ) )

               ( (consp val)
                 (hunt.recurse (car val) current-depth)
                 (hunt.recurse (cdr val) current-depth) )
```

1

```
               ( (vectorp val)
                 (loop (incr i from 0 to (+ -1 (vectorlength val) ) ) (do
                    (hunt.recurse ([] val i) current-depth) ) ) )

               ( t
                 () ) ) ) ) ) ) ) )
```

2

PS:<C.S.BULLDOG.UTILITIES>HUNT.LSP.15

```
;===============================================================================
;
; OPTIONS
;
; This module provides support for manipulating "options" (switches) in
; a coherent way.  An option is represented as a global variable with
; some additional information associated with it.  Functions that want
; the current value of the option just access the global variable.
;
; (DEF-OPTION VARIABLE DEFAULT-VALUE DIRECTORY HELP-STRING)
;     This defines VARIABLE to be an option whose default (initial) value
;     is DEFAULT-VALUE.  DIRECTORY should be the name of the directory in
;     which the descriptive message HELP-STRING will be stored.  HELP-STRING
;     is stored in the file DIRECTORY VARIABLE .OPTION-HELP.  An example:
;
;         (def-option *display-level* 4 trace:
;             "Controls the level of output from the bookkeeper.")
;
; In the following functions, if no options are supplied, then all the
; currently defined options are assumed.  An option may be identified by
; any unique substring; e.g. "TEST" would identify "TESTING" if there were
; no other option containing "TEST".
;
; (OPTIONS.PRINT OPTION1 OPTION2 ...)
;     For each of the given options, prints out its name followed by its
;     current value and default value if different.
;
; (OPTIONS.HELP OPTION1 OPTION2 ...)
;     Prints out the current value of each option and the help string
;     associated with it.
;
; (OPTIONS.SET OPTION VALUE)
;     Sets the value of the option ot be VALUE.
;
; (OPTIONS.RESET OPTION1 OPTION2 ...)
;     Sets the value of the given options back to their default values.
;
;===============================================================================

(defvar *options.all-var&default&file* () )
                        ;*** List of triples of the form:
                        ;***
                        ;***     (variable default-value file-name)
                        ;***
                        ;*** where VARIABLE is the variable name of the
                        ;*** of the option, DEFAULT-VALUE is the initial
                        ;*** value of the option, and FILE-NAME is the
                        ;*** the name of a file containing a help
                        ;*** description.
                        ;***


(defvar *options.new-line* "
")


(defmacro def-option ( variable default-value directory help-string )
    (let ( (filename
            (atomconcat directory
                        (options.make-file-name variable)
                        ".OPTION-HELP") ) )

        (if (! (eqstr *options.new-line* (substring help-string 1 2) ) )
```



```
        (then
            (:= help-string (stconcat *options.new-line* &&&) ) ) )
        (if (! (eqstr *options.new-line* (substring help-string -2) ) )
        (then
            (:= help-string (stconcat &&& *options.new-line*) ) ) )

        '(eval-when (eval compile load)
            (eval-when (eval load)
                (options.define ',variable ,default-value ',filename) )
            (eval-when (eval compile)
                (iota ( (file ',filename '(out newversion) ) )
                    (without file
                        (linelength 80)
                        (msg ,help-string) ) ) )
            (eval-when (load)
                (declare (special ,variable) ) ) ) ) )


(defun options.define ( variable default-value filename )
    (let ( (var&default&file (assoc variable *options.all-var&default&file*) ) )
        (if (! var&default&file) (then
            (:= *options.all-var&default&file*
                (append &&&
                        (list (:= var&default&file (list () () () ) ) ) ) ) )))

        (:= (car   var&default&file) variable)
        (:= (cadr  var&default&file) default-value)
        (:= (caddr var&default&file) filename)
        (set variable default-value) ) )


(defun options.help (&rest option-list)
    (if (! option-list) (then
        (:= option-list (options.all-options) ) ) )
    (loop (for option in option-list) (do
        (options.option:help option) ) ) )


(defun options.print (&rest option-list)
    (if (! option-list) (then
        (:= option-list (options.all-options) ) ) )
    (loop (for option in option-list) (do
        (options.option:print option) ) ) )


(defun options.set ( option value )
    (options.option:set option value) )


(defun options.reset (&rest option-list)
    (if (! option-list) (then
        (:= option-list (options.all-options) ) ) )
    (loop (for option in option-list) (do
        (options.option:reset option) ) ) )


;***============================================================================
;***
;*** (OPTIONS.ALL-OPTIONS)
;***
;*** Returns a list of all the options current defined.
;***
;***============================================================================
```

```
(defun options.all-options ()
    (for (var&default&file in *options.all-var&default&file*) (save
        (car var&default&file) ) ) )


;***============================================================================
;***
;*** (OPTIONS.MAKE-FILE-NAME STRING)
;***
;*** Strips all non-alphanumeric characters from STRING so that it is suitable
;*** for a TOPS-20 filename.
;***
;***============================================================================

(defun options.make-file-name ( string )
    (loop (for char in (aexplodec string) )
          (initial result () )
    (do
        (if (memq char
                '(#/A #/B #/C #/D #/E #/F #/G #/H #/I #/J #/K #/L #/M #/N
                  #/O #/P #/Q #/R #/S #/T #/U #/V #/W #/X #/Y #/Z
                  #/a #/b #/c #/d #/e #/f #/g #/h #/i #/j #/k #/l #/m #/n
                  #/o #/p #/q #/r #/s #/t #/u #/v #/w #/x #/y #/z
                  #/0 #/1 #/2 #/3 #/4 #/5 #/6 #/7 #/8 #/9 #/_ #/- #/$) )
        (then
            (push result char) ) ) )
    (result (packc (dreverse result) ) ) ) )


;***============================================================================
;***
;*** (OPTIONS.OPTION:VAR&DEFAULT&FILE OPTION)
;***
;*** Returns the descriptive triple for an option, () if the option is not
;*** defined.  OPTION may be a unique substring of the options full name.
;***
;***============================================================================

(defun options.option:var&default&file ( option )
    (if-let ( (var&default&file (assoc option
                                       *options.all-var&default&file*) ) )
    (then
        var&default&file)

    (else
        (loop (for var&default&file in *options.all-var&default&file*)
              (initial match () )
        (do
            (caseq (options.string-match (car var&default&file) option)
                (exact
                    (return var&default&file) )
                (substring
                    (if match (then
                        (return () ) )
                    (else
                        (:= match var&default&file) ) ) ) ) )
        (result match) ) ) ) )



;***============================================================================
;***
```



```
;*** (OPTIONS.OPTION:PRINT OPTION)
;***
;*** Prints out a single option.  OPTION may be a unique substring of
;*** the options full name.
;***
;***============================================================================

(defun options.option:print ( option )
    (if-let ( (var&default&file (options.option:var&default&file option) ) )
    (then
        (let ( (value (eval (car var&default&file) ) ) )
            (msg 0 (car var&default&file) (t 35) " = " value)
            (if (!= value (cadr var&default&file) ) (then
                (msg " [" (cadr var&default&file) "]") ) )
            (terpri) ) )
    (else
        (msg 0 "%" option " is an undefined option." t) ) ) )


;***============================================================================
;***
;*** (OPTIONS.OPTION:HELP OPTION)
;***
;*** Gives the help for a single option.  OPTION may be a unique substring of
;*** the options full name.
;***
;***============================================================================

(defun options.option:help ( option )
    (if-let ( (var&default&file (options.option:var&default&file option) ) )
    (then
        (options.option:print option)
        (iota ( (file (caddr var&default&file) '(in old) ) )
            (within file (errset
                (loop (do (tyo (tyi) ) ) ) ) ) ) )
        (msg 0 "------" t t) )
    (else
        (msg 0 "%" option " is an undefined option." t) ) ) )


;***============================================================================
;***
;*** (OPTIONS.OPTION:SET OPTION VALUE)
;***
;*** Gives the help for a single option.  OPTION may be a unique substring
;*** of the options full name.
;***
;***============================================================================

(defun options.option:set ( option value )
    (if-let ( (var&default&file (options.option:var&default&file option) ) )
    (then
        (set (car var&default&file) value) )
    (else
        (msg 0 "%" option " is an undefined option." t) ) ) )


;***============================================================================
;***
;*** (OPTIONS.OPTION:RESET OPTION)
;***
;*** Resets a single option.  OPTION may be a unique substring
;*** of the options full name.
```

```
;***============================================================================

(defun options.option:reset ( option )
    (if-let ( (var&default&file (options.option:var&default&file option) ) )
    (then
        (set (car var&default&file) (cadr var&default&file) ) )
    (else
        (msg 0 "%" option " is an undefined option." t) ) ) )


;***============================================================================
;***
;*** (OPTIONS.STRING-MATCH STRING SUBSTRING)
;***
;*** Compares the print names of two symbols or strings, returning EXACT
;*** if they are equal, SUBSTRING if SUBSTRING is a substring of STRING,
;*** and () otherwise.
;***
;***============================================================================

(defun options.string-match ( string substring )
    (let ( (string    (aexplodec string) )
           (substring (aexplodec substring) ) )

        (loop (label outer)
              (initial rest-string string)
        (while rest-string)
        (do
            (loop (initial rest-string1   rest-string)
                  (initial rest-substring substring)
            (do
                (if (! rest-substring) (then
                    (leave outer
                           (if (&& (== rest-string string)
                                   (! rest-string1) )
                               'exact
                               'substring) ) ) )
                (if (! rest-string1)
                    (return () ) ) )
            (while (= (car rest-string1) (car rest-substring) ) )
            (next rest-substring (cdr rest-substring)
                  rest-string1   (cdr rest-string1) ) ) )
        (next rest-string (cdr rest-string) )
        (result () ) ) ) )
```

```
;***============================================================================
;
; ## (Sharp-Sharp)
;
; This module provides an easy way of defining concise syntax for
; interactively accessing global objects in the Bulldog compiler.  For
; example, the trace compacter records are called MI and are numbered.
; Using ## we can easily refer to a particular record, say 36, by typing:
;
;     ##mi 36
;
; To define that syntax you would do:
;
;     (def-sharp-sharp mi
;         '(mi-with-number ,(read) ) )
;
; DEF-SHARP-SHARP defines a special read (syntax) macro that should return
; the form for accessing the desired object.
;
;***============================================================================

(def-sharp-character /#
    (let*( (object-name (read) )
           (func (prop 'sharp-sharp.function object-name) ) )
        (if func
            (funcall func)
            (error (list object-name "Unknown ## object type.") ) ) ) )

(defmacro def-sharp-sharp ( name . body )
    (let ( (function (atomconcat name '-sharp-sharp.function) ) )
        `(eval-when (eval load)
            (defun ,function () ,,body)
            (:= (prop 'sharp-sharp.function ',name) ',function)
            ',name) ) )
```

```
;;;============================================================================
;; VECTOR-MAP
;;
;; This module contains functions for initializing and adding elements
;; to vectors that are used for mapping integers onto values.  The vector
;; maps are automatically expanded as necessary.
;;
;;
;; (VECTOR-MAP:INITIALIZE VECTOR-VAR CURRENT-SIZE-VAR INITIAL-SIZE
;;                        &OPTIONAL VERBOSE)
;;
;;     VECTOR-VAR and CURRENT-SIZE-VAR are symbols.  The value of the
;;     symbol in VECTOR-VAR is set to be an vector of at least INITIAL-SIZE
;;     elements (all ()).  If the value of the symbol in VECTOR-VAR is
;;     already an vector, it is expanded if necessary to size
;;     INITIAL-SIZE, and then cleared to all ().  The value of the symbol
;;     in CURRENT-SIZE-VAR is set to 0.  If VERBOSE, then some
;;     informative messages are printed.
;;
;; (VECTOR-MAP:ADD-ELEMENT VECTOR-VAR CURRENT-SIZE-VAR NEW-ELEMENT
;;                         SIZE-INCREMENT &OPTIONAL VERBOSE)
;;
;;     VECTOR-VAR and CURRENT-SIZE-VAR are symbols.  The value of the
;;     symbol in VECTOR-VAR should be an vector.  NEW-ELEMENT is stored
;;     at the next empty slot in the vector (specified by the value of
;;     the symbol CURRENT-SIZE-VAR), and the value of the
;;     CURRENT-SIZE-VAR symbol is incremented by 1.  If the vector is
;;     not big enough, it is expaned by SIZE-INCREMENT elements.  If
;;     VERBOSE, an informative message is printed whenever the vector
;;     is expanded.
;;
;;============================================================================


(defun vector-map:initialize
       ( vector-var current-size-var initial-size &optional verbose )

    (set current-size-var 0)
    (if (|| (! (boundp vector-var) )
            (! (atomeval vector-var) )
            (> initial-size (vectorlength (atomeval vector-var) ) ) )
    (then
        (set vector-var (makevector initial-size) )
        (if verbose
            (msg 0 "Initializing vector " vector-var " to size "
                 initial-size t) ) )
    (else
        (loop (incr i from 0 to (+ -1 (vectorlength (atomeval vector-var))))
        (do
            (:= ([] (atomeval vector-var) i) () ) ) ) ) ) )



(defun vector-map:add-element
       ( vector-var current-size-var new-element size-increment
         &optional verbose )

    (let*( (vector        (atomeval vector-var) )
           (vector-size   (vectorlength vector) )
           (current-size  (atomeval current-size-var) ) )

        (if (>= current-size vector-size) (then
```

1

```
            (set vector-var (:= vector
                (vector:copy vector (+ size-increment vector-size) ) ) )
            (if verbose (then
                (msg 0 "Expanding vector " vector-var " to size "
                    (vectorlength vector) ) ) ) ) )
        (:= ([] vector current-size) new-element)
        (set current-size-var (+ 1 current-size) )
        new-element) )
```

2

```
;============================================================================
;
; (VISIBLE-FIELDS STRUCT-NAME FIELD-NAME-1 FIELD-NAME-2 ...)

;     Declares that only the given fields are to printed when printing
;     out an instance of the given structure.
;
;============================================================================

(defun visible-fields* (struct fields)
    (loop (for description in (prop 'structure-fields struct) )
    (do
        (if (memq (car description) fields)

            (:= (nth-elt description 4) () )

            (:= (nth-elt description 4) 'suppress) ) ) ) )


(defmacro visible-fields (struct . fields)
   '(visible-fields* ',struct ',fields) )
```