BIT SETS

This module implements sets of integers \( \geq 0 \). 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:Ddifference 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.

```
(= 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 (reverse result))))
  (defun bit-set:universe ( size )
    (loop (initial result ()
      (while (> size s.blts-per-lnteger) )
      (do (push result -1)
        (next size (- size s.blts-per-lnteger) )
      (result (push result (lsh32 -1 (- s.blts-per-lnteger size) ) )
        (reverse result))))
  (defun bit-set:singleton ( element )
    (bit-set:unlonl () element) )
  (defun bit-set:member? ( set element )
    (loop (for word ln set)
      (do (lf (<= element s.blts-per-lnteger-1)
        (return (!== 0 (logand word
          (lsh32 1 (- #.bits-per-integer-1 element))))))
        (next element (- element s.blts-per-lnteger) )
      (result () ) )
  (defun bit-set:choose ( set )
    (loop (for word ln set)
      (do (if (!= 0 word)
        (result (+ bas (jffo32 word) ) ) )
      (result () ) )
  (defun bit-set:intersection args
    (caseq args
      0 *blt-set.eapty-set*)
      1 (arg 1) )
    (t
      (loop (initial result ()
        next-element ()
        all-done? ()
        (do (:s all-done? ()
          (:s next-element -1)
          (loop (incr 1 froa 1 to arga) (do
            (lf (!= 0 word)
              (then
                (= all-done? t)
              (result (+ 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 ()
            all-done? ()
            (do (:s all-done? ()
              (:s next-element -1)
              (loop (incr 1 froa 1 to arga) (do
                (if (!= 0 word)
              (then
                (= all-done? t)
(defun bit-set:union args
  (caseq args
    (0 *empty-set*)
    (1 (arg 1))
    (t
     (loop (initial result ()
               next-element 0
               all-done? ()
               (do
                (:s all-done? t)
                (:= next-element 0)
                (loop (incr 1 from 1 to args) (do
                  (if (arg 1) (then
                    (:= all-done? ()
                    (:= next-element (logor next-element (pop (arg 1) ) )
                    (if all-done?
                      (return (dreverse result) )
                      (push result next-elenent) ))))))))))
  (result (dreverse result) ) )
)

PS:<C.S.BULLDOG.UTILITIES>BIT-SET.LSP.28
(\(\text{defun bit-set:size} (\text{set})\)
 (\text{loop (initial size 0)}
 (\text{for-each-bit-set-element set element)}
 (\text{return t)}))
)

(\(\text{defun bit-set:print} (\text{set})\)
 (\text{msg *(*)}
 (\text{loop (label print-next-range)}
 (\text{do (do (do (do (do}}
 (\text{space-needed t)}
 (\text{caseq (- last-in-range first-in-range)}
 (\text{msg "")
 (\text{msg *)")}
 (\text{msg ")")})
 (\text{})}
)

\text{PS:<C.S.BULLDOG.UTILITIES>BIT-SET.LSP.28}
(:= *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*) )

PS:<C.S.BULLDOG.UTILITIES>BUILD.LSP.4
HUNT helps you find forgotten 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 I) means look for any values that are the direct value 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*)
     (inc current-depth)
     (if (consp val)
       (catch ( hunt.recurse (car val) current-depth)
         (catch ( hunt.recurse (cdr val) current-depth)
           (loop (Initial rest (plist val))
             (while rest)
             (do
               (hunt.recurse (cadr rest) current-depth)
               (next rest (cddr rest))
             )
           )
         )
       )
     )
   ))
  )))
```

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

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:

```lisp
(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 to be VALUE.

(OPTIONS.RESET OPTION1 OPTION2 ...)
Sets the value of the given options back to their default values.
(defun options.all-options ()
  (for (var-default-file in *options.all-var-default-file*)
    (save (car var-default-file)) ))

(defun options.make-file-name (string)
  (loop (for char in (explode string))
    (initial result ())
    (do
      (if (eql char
        '(/A #/B S/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)
        (then (push result char)))))
  (result (pack (reverse result))))

(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))))))))))

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

(defun options.option:help (option)
  (if-let ((var-default-file (options.option:var-default-file option))
    (then (options.option:print option)
      (lota (file (caddr var-default-file) '(In old))
        (within file (error (loop (do (tyo (tyi))))))))
    (else (asg 0 "X" option " is an undefined option." t))))

(defun options.option:setting (option value)
  (if-let ((var-default-file (options.option:var-default-file option))
    (then (asg (car var-default-file) value)
      (else (asg 0 "X" option " is an undefined option." t))))
    (else (asg 0 "X" option " is an undefined option." t))))

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

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

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

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

; *** (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-string rest-string)
            (initial rest-substring substring)
            (do
              (if (! rest-substring) (then
                (leave outer
                  (if (= rest-string string)
                    (return ()
                    'exact
                    'substring)
                  (if (! rest-string)
                    (return ()
                   )
                  (while (= (car rest-string) (car rest-substring))
                    (next rest-string (cdr rest-string))
                    (next rest-substring (cdr rest-substring))
                    (result ()
                    )
                  )))
            )))
          (next rest-string (cdr rest-string))
          (result ()))))
  ))

PS:<C.S.BULLDOG.UTILITIES>OPTIONS.LSP.32
(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 (atonconcat 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 expanded 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) ) )
  )
    (set vector-var (makevector initial-size) )
    (if verbose
      (msg 0 "Initializing vector " vector-var " to size " initial-size t )
    )
  )
  (else
    (loop (incr 1 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-size-vector) (then
    (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 )
  ))
(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 )