ARNESI

Contents

1 Introduction

It is a collection of lots of small bits and pieces which have proven themselves usefull in various applications. They are all tested, some even have a test suite and a few are even documentated.

```
(defpackage :it.bese.arnesi
  (:documentation "The arnesi utility suite.")
  (:nicknames :arnesi)
  (:use :common-lisp)
  (:export

   #:clean-op

   #:make-reducer
   #:make-pusher
   #:make-collector
   #:with-reducer
   #:with-collector
   #:with-collectors

   #:to-cps
   #:with-call/cc
   #:call/cc
   #:let/cc
   #:*call/cc-returns*
   #:invalid-return-from
   #:unreachable-code
   #:defun/cc
   #:defgeneric/cc
   #:defmethod/cc
   #:assert-cc
   #:fmakun-cc

   #:ppm
   #:ppm1
   #:apropos-list*
   #:apropos*

   #:with-input-from-file
   #:with-output-to-file
   #:read-string-from-file
   #:write-string-to-file
   #:copy-file

   #:if-bind
   #:aif
   #:when-bind
   #:awhen
   #:cond-bind
```

```
#:acond
#:aand
#:and-bind
#:it
#:whichever
#:xor
#:switch
#:eswitch
#:cswitch

#:build-hash-table
#:deflookup-table
#:hash-to-alist

#:write-as-uri
#:escape-as-uri
#:unescape-as-uri
#:nunescape-as-uri
#:write-as-html
#:escape-as-html
#:unescape-as-html

#:compose
#:conjoin
#:curry
#:rcurry
#:noop
#:y
#:lambda-rec

#:dolist*
#:dotree
#:ensure-list
#:ensure-cons
#:partition
#:partitionx
#:proper-list-p
#:push*

#:get-logger
#:log-category
#:stream-log-appender
#:make-stream-log-appender
#:file-log-appender
#:make-file-log-appender
#:log.dribble
#:log.debug
#:log.info
#:log.warn
#:log.error
```

```
#:log.fatal
#:deflogger
#:log.level
#:+dribble+
#:+debug+
#:+info+
#:+warn+
#:+error+
#:+fatal+
#:handle
#:append-message
#:ancestors
#:appenders
#:childer

#:with-unique-names
#:rebinding
#:define-constant

#:make-matcher
#:match
#:match-case
#:list-match-case

#:parse-ieee-double
#:parse-float
#:mulf
#:divf
#:minf
#:maxf
#:map-range
#:do-range
#:10^

#:tail
#:but-tail
#:head
#:but-head
#:starts-with
#:ends-with
#:read-sequence*
#:deletef

#:+lower-case-ascii-alphabet+
#:+upper-case-ascii-alphabet+
#:+ascii-alphabet+
#:+alphanumeric-ascii-alphabet+
#:+base64-alphabet+
#:random-string
#:strcat
```

```
#:strcat*
#:princ-csv
#:fold-strings
#:trim-string
#:~%
#:~T
#:+CR-LF+
#:~D
#:~A
#:~S
#:~W

#:def-special-enviroment

#:intern-concat

#:vector-push-extend*
#:string-from-array

;; decimal arith
#:*precision*
#:with-precision
#:decimal-from-float
#:float-from-decimal
#:round-down
#:round-half-up
#:round-half-even
#:round-ceiling
#:round-floor
#:round-half-down
#:round-up

#:enable-sharp-l

#:defclass-struct

#:with*

#:quit

#:wrapping-standard
))
```

2 Colophon

This documentation was produced by qbook.

arnesi, and the associated documentation, is written by Edward Marco Baringer ¡mb@bese.it¿.

2.1 COPYRIGHT

3 Reducing and Collecting

3.1 Reducing

reducing is the act of taking values, two at a time, and combining them, with the aid of a reducing function, into a single final value.

```
(defun make-reducer (function &optional (initial-value nil initial-value-p))
  "Create a function which, starting with INITIAL-VALUE, reduces
any other values into a single final value.

FUNCTION will be called with two values: the current value and
the new value, in that order. FUNCTION should return exactly one
value.

The reducing function can be called with n arguments which will
be applied to FUNCTION one after the other (left to right) and
will return the new value.

If the reducing function is called with no arguments it will
return the current value.
```

```
Example:

 (setf r (make-reducer #'+ 5))
 (funcall r 0) => 5
 (funcall r 1 2) => 8
 (funcall r) => 8"
  (let ((value initial-value))
    (lambda (&rest next)
      (when next
        ;; supplied a value, reduce
        (if initial-value-p
            ;; have a value to test against
            (dolist (n next)
              (setf value (funcall function value n)))
            ;; nothing to test againts yet
            (setf initial-value-p t
                  value next)))
      ;; didn't supply a value, return the current value
      value)))

(defmacro with-reducer ((name function &optional (initial-value nil))
                        &body body)
  "Locally bind NAME to a reducing function. The arguments
FUNCTION and INITIAL-VALUE are passed directly to MAKE-REDUCER."
  (with-unique-names (reducer)
    `(let ((,reducer (make-reducer ,function ,@(list initial-value))))
       (flet ((,name (&rest items)
                (if items
                    (dolist (i items)
                      (funcall ,reducer i))
                    (funcall ,reducer))))
         ,@body))))
```

3.2 Collecting

Building up a list from multiple values.

```
(defun make-collector (&optional initial-value)
  "Create a collector function.

A Collector function will collect, into a list, all the values
passed to it in the order in which they were passed. If the
callector function is called without arguments it returns the
current list of values."
  (let ((value initial-value)
        (cdr (last initial-value)))
    (lambda (&rest items)
      (if items
          (progn
            (if value
```

```
                (if cdr
                    (setf (cdr cdr) items
                          cdr (last items))
                    (setf cdr (last items)))
                (setf value items
                      cdr (last items)))
            items)
          value))))

(defun make-pusher (&optional initial-value)
  "Create a function which collects values as by PUSH."
  (let ((value initial-value))
    (lambda (&rest items)
      (if items
          (progn
            (dolist (i items)
              (push i value))
            items)
          value))))

(defmacro with-collector ((name &optional initial-value from-end) &body body)
  "Bind NAME to a collector function and execute BODY. If
  FROM-END is true the collector will actually be a pusher, (see
  MAKE-PUSHER), otherwise NAME will be bound to a collector,
  (see MAKE-COLLECTOR)."
  (with-unique-names (collector)
    `(let ((,collector ,(if from-end
                            `(make-pusher ,initial-value)
                            `(make-collector ,initial-value))))
       (flet ((,name (&rest items)
                (if items
                    (dolist (i items)
                      (funcall ,collector i))
                    (funcall ,collector))))
         ,@body))))

(defmacro with-collectors (names &body body)
  "Bind multiple collectors. Each element of NAMES should be a
  list as per WITH-COLLECTOR's first orgument."
  (if names
      `(with-collector ,(ensure-list (car names))
         (with-collectors ,(cdr names) ,@body))
      `(progn ,@body)))
```

4 ASDF extras

4.1 CLEAN-OP - An intelligent make clean for ASDF

```
(defclass clean-op (asdf:operation)
  ((for-op :accessor for-op :initarg :for-op :initform 'asdf:compile-op))
  (:documentation "Removes any files generated by an asdf component."))
```

```
(defmethod asdf:perform ((op clean-op) (c asdf:component))
  "Delete all the output files generated by the component C."
  (dolist (f (asdf:output-files (make-instance (for-op op)) c))
    (when (probe-file f)
      (delete-file f))))

(defmethod asdf:operation-done-p ((op clean-op) (c asdf:component))
  "Returns T when the output-files of (for-op OP) C don't exist."
  (dolist (f (asdf:output-files (make-instance (for-op op)) c))
    (when (probe-file f) (return-from asdf:operation-done-p nil)))
  t)
```

4.2 Creating a single .fas or .fasl file

Instead of creating images another way to distribute systems is to create a single compiled file containing all the code. This is only possible on some lisps, sbcl and clisp are the only ones supported for now.

NB: Unlike the CLEAN-OP this is experimental (its now to have problems on multiple systems with non-trivial dependencies).

```
(defun make-single-fasl (system-name
                         &key (op (make-instance 'asdf:load-op))
                              output-file)
  (let* ((system (asdf:find-system system-name))
         (steps (asdf::traverse op system))
         (output-file (or output-file
                          (compile-file-pathname
                           (make-pathname
                            :name (asdf:component-name system)
                            :defaults (asdf:component-pathname system)))))
         (*buffer* (make-array 4096 :element-type '(unsigned-byte 8)
                                    :adjustable t)))
    (declare (special *buffer*))
    (with-output-to-file (*fasl* output-file
                          :if-exists :error
                          :element-type '(unsigned-byte 8))
      (declare (special *fasl*))
      (dolist (s steps)
        (process-step (car s) (cdr s) output-file)))))

(defgeneric process-step (op comp output-file))

(defmethod process-step
    ((op asdf:load-op) (file asdf:cl-source-file) output-file)
  (declare (ignore output-file)
           (special *buffer* *fasl*))
  (dolist (fasl (asdf:output-files (make-instance 'asdf:compile-op) file))
    (with-input-from-file (input (truename fasl)
                           :element-type '(unsigned-byte 8))
      (setf *buffer* (adjust-array *buffer* (file-length input)))
      (read-sequence *buffer* input)
      (write-sequence *buffer* *fasl*))))
```

```
(defmethod process-step ((op asdf:operation) (comp asdf:component) output-file)
  (declare (ignore output-file))
  (format t "Ignoring step ~S on ~S.~%" op comp))
```

5 A Trivial Compatability Layer

Here we only have the QUIT function, see mopp.lisp for a MOP campatability layer.

```
(defun quit (&optional (exit-code 0))
  #+openmcl (ccl:quit exit-code)
  #+sbcl (sb-ext:quit :unix-status exit-code)
  #+clisp (ext:quit exit-code)
  #+cmu (declare (ignore exit-code))
  #+cmu (ext:quit)
  )
```

6 Automatically Converting a Subset of Common Lisp to CPS

By transforming common lisp code into CPS we allow a restricted form of CALL/CC. While we use the name call/cc what we actually provide is more like the shift/reset operators (where with-call/cc is reset and call/cc is shift).

6.1 Supported Forms

This transformer can not handle all of common lisp, the following special operators are allowed: block, declare, flet, function, go, if, labels, let, let*, macrolet, progn, quote, return-from, setq, symbol-macrolet, and tagbody. The special operators the and unwind-protect are allowed _only_ if their bodies do not attempt to use call/cc. The following special operators could be supported but have not yet been implemented: load-time-value, locally, multiple-value-call and multiple-value-prog1. The following special operators are not allowed and are not implementable (not without the same restrictions we place on the and unwind-protect at least): catch, throw, eval-when, progv.

6.2 Transformer Overview

We convert the code in two distinct passes. The first pass performs macro expansion, variable renaming and checks that only 'good' special operators appear in the code. The second pass then converts the special operators into a CPS (after pass one the code will consist of nothing other than special operators and application forms).

For every special operator we define 3 functions: a transformer, a renamer (which must also do macroexpansion if neccessary) and a requries test. The requires test tells us if a particular form needs to be CPS'd or not (basically it just check whether that form contains a call to call/cc or not. Most of these functions are defined with the macrso defcps-transformer, defcps-rename and

defcps-requires. The exceptions are the handles for atoms and general function/macro application which need more information then the macros provide.

6.3 Entry Point

```
(defmacro with-call/cc (&body body)
  "Execute BODY with quasi continutations.

BODY may not refer to macrolets and symbol-macrolets defined
outside of BODY.

Within BODY the \"operator\" call/cc can be used to access the
current continuation. call/cc takes a single argument which must
be a function of one argument. This function will be passed the
curent continuation.

with-call/cc simply CPS transforms it's body, so the continuation
pass to call/cc is NOT a real continuation, but goes only as far
back as the nearest lexically enclosing with-call/cc form."
  (case (length body)
    (0 NIL)
    (1 (to-cps (first body)))
    (t (to-cps `(progn ,@body)))))

(defvar *call/cc-returns* nil
  "Set to T if CALL/CC should call its continuation, otherwise
the lambda passed to CALL/CC must call the continuation
explicitly.")
```

6.4 CPS Transformer Environment

```
(defun toplevel-k (value)
  (throw 'done value))

(defvar *env* :unitialized)

(defun to-cps (form &optional (k '#'toplevel-k))
  `(drive-cps ,(%to-cps form k)))

(defun %to-cps (form k)
  (let ((*env* nil))
    (setf form (cps-rename form)))
  (let ((*env* nil))
    (setf form (cps-transform form k)))
  form)

(defun drive-cps (cps-lambda)
  (catch 'done
    (loop for f = cps-lambda then (funcall f))))
```

```
(defmacro let/cc (k &body body)
  `(call/cc (lambda (,k)
              (flet ((,k (v)
                       (funcall ,k v)))
                ,@body))))

(defparameter *non-cps-calls* (list)
  "When transforming code to CPS this variable is used to handle
  the names of all function which are called in \"direct\"
  stile.")

(defun add-non-cc (symbol)
  (unless (eq (symbol-package symbol) #.(find-package "COMMON-LISP"))
    (pushnew symbol *non-cps-calls*)))

(defun assert-non-cc (symbols while-defining)
  "Assert that the functions named by SYMBOLS are not CC
  functions. We also record that this assertion was made while
  compiling the function/method while-defining."
  (dolist (sym symbols)
    (if (get sym 'call/cc)
(error "Attempting to register ~S as a non CC function.
However it was previously registered as a CC function in ~S."
       sym (get sym 'call/cc))
(setf (get sym 'non-call/cc) while-defining))))

(defun assert-cc (name &optional (while-defining :toplevel))
  (if (get name 'non-call/cc)
      (error "Attempting to register ~S as a CC function.
However it was previously registered an a non CC function in ~S."
     name (get name 'non-call/cc))
      (setf (get name 'call/cc) while-defining)))

(defun fmakun-cc (name)
  (remf (symbol-plist name) 'call/cc)
  (remf (symbol-plist name) 'non-call/cc)
  name)

(defun gen-cc-definiton (what &key name block-name args body)
  (assert-cc name)
  (multiple-value-bind (declares doc-strings body)
      (split-body-into-parts body)
    (let* ((k (gensym "K-"))
   (*non-cps-calls* (list))
   (code (to-cps `(block ,block-name ,@body) k)))
      (assert-non-cc *non-cps-calls* name)
      `(progn
 (,what ,name ,args
     ,@doc-strings
     ,@declares
     ;; capture the value of *k* at the time we are called
```

```
     (let ((,k (or *k* #'toplevel-k)))
       (declare (ignorable ,k))
       ,code))
 (assert-non-cc ',*non-cps-calls* ',name)
 (assert-cc ',name)
 ',name))))

(defmacro defun/cc (name args &body body)
  (gen-cc-definiton 'defun
    :name name
    :block-name name
    :args args
    :body body))

(defmacro defmethod/cc (name lambda-list &body body)
  "Define a method which can be used \"normally\" with in
  CALL/CC."
  (when (symbolp lambda-list)
    (error "DEFMETHOD/CC can't be used in non-standard method combination."))
  (gen-cc-definiton 'defmethod
    :name name
    :block-name (if (listp name)
    (second name)
    name)
    :args lambda-list
    :body body))

(defmacro defgeneric/cc (fun-name lambda-list &body options)
  (flet ((expand-method-definition (qab) ; QAB = qualifiers, arglist, body
   (if (listp (first qab))
       ;; no qualifiers
       (destructuring-bind (args &body body)
   qab
 `(defmethod/cc ,fun-name ,args ,@body))
       (destructuring-bind (qualifier args &body body)
   qab
 `(defmethod ,fun-name ,qualifier ,args ,@body)))))
    (let ((methods (list))
    (other-options (list)))
      (dolist (option options)
  (let ((car-option (car option)))
    (case car-option
      ((:method)
       (push (cdr option) methods))
      (t
       (push option other-options)))))
      `(progn
 (defgeneric ,fun-name ,lambda-list ,@other-options)
 ,@(mapcar #'(lambda (method)
       (expand-method-definition method))
   methods)
```

```
 (assert-cc ',fun-name)
 ',fun-name))))
```

6.5 Helpers

```
(defvar *k* nil
  "The holder for inter-call continuations. See the application
  cps-transformer for details.")

(defun register (type name &optional (data (gensym)))
  "Register, the current compilation environment, that NAME is an
  object of type TYPE with the associated data DATA. TYPE can be
  one of :flet, :let, :macrolet, :symbol-macrolet, :block
  or :tag."
  (unless (member type '(:flet :let :macrolet :symbol-macrolet :block :tag))
    (warn "Unknow TYPE: ~S." type))
  (setf *env* (cons (list type name data) *env*))
  data)

(defun lookup (type name)
  "Return the data associated with NAME (of type TYPE) in the
  current compilation environment."
  (loop
     for (itype iname data) in *env*
     when (and (eql itype type)
       (eql name iname))
     do (return data)
     finally (return nil)))

(defmacro k (value)
  "Expand into code which will expand into code which will call
  the current continuation (assumed to be bound to the lexical
  variable K) passing it VALUE."  ``(lambda ()
      (funcall ,k ,,value)))

(defun lookup-cps-handler (form)
  "Simple function which tells us what cps-handler should deal
  with FORM. Signals an error if we don't have a handler for
  FORM."
  (if (atom form)
      'atom
      (case (car form)
        ((block declare flet function go if labels let let* macrolet progn quote return-fr
          symbol-macrolet tagbody call/cc the unwind-protect)
         (car form))
        ((catch eval-when load-time-value locally multiple-value-call multiple-value-prog1
         (error "~S not allowed in CPS code (in form ~S), please refactor." (car form) for
        (t 'application))))

(defun cps-requires-body (body)
  (multiple-value-bind (declares actual-body)
```

```
      (split-into-body-and-declares body)
    (declare (ignore declares))
    (some #'cps-requires actual-body)))

(defun cps-transform-body (body k)
  "Unlike most cps transformers this returns a list."
  (multiple-value-bind (declares actual-body)
      (split-into-body-and-declares body)
    (if declares
        `((declare ,@declares) ,(cps-transform `(progn ,@actual-body) k))
        (list (cps-transform `(progn ,@actual-body) k)))))
```

6.5.1 Helpers for dealing with declarations and bodys

```
(defun split-body-into-parts (body)
  "Returns the declares, the doc string, and any forms in BODY."
  (flet ((ret (dec doc body)
           (return-from split-body-into-parts (values dec
                                                      (when doc
                                                        (list doc))
                                                      body))))
    (loop
       with doc-string = nil
       with declares = '()
       for form* on body
       for form = (car form*)
       do (cond
            ((and (stringp form) (cdr form*))
             (setf doc-string form))
            ((stringp form)
             (ret declares doc-string form*))
            ((and (consp form) (eql 'cl:declare (car form)))
             (push form declares))
            (t
             (ret declares doc-string form*))))))

(defun declare-form-p (form)
  (and (consp form)
       (eql 'declare (car form))))

(defun split-into-body-and-declares (body)
  (loop for form on body
        while (declare-form-p (car form))
        collect (car form) into declares
        finally (return (values (mapcan #'cdr declares) form))))

(defun extract-associated-declares (var body)
  "Returns a declare form containig all the options for VAR in
  the declare forms in BODY."
  (multiple-value-bind (declares body)
      (split-into-body-and-declares body)
```

```
    (let ((associated-declares '()))
      (dolist* ((type . args) declares)
        (case type
          ((ignore ignorable dynamic-extent)
           (if (member var args)
               (progn
                 (push `(declare (,type ,var)) associated-declares)
                 (aif (remove var args)
                      (push `(declare (,type ,@it)) body)))
               (push `(declare (,type ,@args)) body)))
          ((special)
           (error "SPECIAL declares not allowed in CALL/CC code."))
          ((inline notinline optimize)
           (push `(declare (,type ,@args)) body))
          ((type)
           (destructuring-bind (type-spec . vars)
               args
             (if (member var vars)
                 (progn
                   (push `(declare (,type ,type-spec ,var)) associated-declares)
                   (aif (remove var vars)
                        (push `(declare ,type ,type-spec ,@it) body)))
                 (push `(declare (,type ,type-spec ,@vars)) body))))
          (t (if (member var args)
                 (progn
                   (push `(declare (,type ,var)) associated-declares)
                   (awhen (remove var args)
                     (push `(declare (,type ,@it)) body)))
                 (push `(declare (,type ,@args)) body)))))
      (values associated-declares body))))

(defun lambda-form-p (form)
  (and (consp form)
       (or (eql 'cl:lambda (car form))
           (and (eql 'cl:function (car form))
                (consp (second form))
                (eql 'cl:lambda (first (second form)))))))
```

6.6 Handlers

6.6.1 Defining Handlers

```
(defvar *cps-handlers* (make-hash-table :test 'eq))

(defun find-cps-handler (form)
  (gethash (lookup-cps-handler form) *cps-handlers*))

(defclass cps-handler ()
  ((transformer :accessor handler.transformer)
   (requires-checker :accessor handler.requires-checker)
   (rename :accessor handler.rename)))
```

```
(defun ensure-exists-handler (name)
  (unless (gethash name *cps-handlers*)
    (setf (gethash name *cps-handlers*)
          (make-instance 'cps-handler))))

(defun cps-transform (form k)
  (if (cps-requires form)
      (funcall (handler.transformer (find-cps-handler form)) form k)
      (k form)))

(defmacro defcps-transformer (name args &body body)
  `(progn
     (ensure-exists-handler ',name)
     (setf (handler.transformer (gethash ',name *cps-handlers*))
           (lambda (form k)
             (declare (ignorable k))
             (let ((*env* *env*))
               (destructuring-bind ,args (cdr form)
                 ,@body))))
     ',name))

(defun cps-requires (form)
  (funcall (handler.requires-checker (find-cps-handler form)) form))

(defmacro defcps-requires (name args &body body)
  `(progn
     (ensure-exists-handler ',name)
     (setf (handler.requires-checker (gethash ',name *cps-handlers*))
           (lambda (form)
             (let ((*env* *env*))
               (destructuring-bind ,args (cdr form)
                 ,@body))))
     ',name))

(defun cps-rename (form)
  (funcall (handler.rename (find-cps-handler form)) form))

(defmacro defcps-rename (name args &body body)
  `(progn
     (ensure-exists-handler ',name)
     (setf (handler.rename (gethash ',name *cps-handlers*))
           (lambda (form)
             (let ((*env* *env*))
               (destructuring-bind ,args (cdr form)
                 ,@body))))
     ',name))
```

6.6.2 Actual handlers

atoms

```
(ensure-exists-handler 'atom)
```

```
(setf (handler.rename (gethash 'atom *cps-handlers*))
      (lambda (atom)
        (acond
         ((lookup :let atom) it)
         ((lookup :symbol-macrolet atom) (cps-rename it))
         (t atom))))

(setf (handler.transformer (gethash 'atom *cps-handlers*))
      (lambda (atom k)
        (k atom)))

(setf (handler.requires-checker (gethash 'atom *cps-handlers*))
      (lambda (atom)
        (declare (ignore atom))
        nil))
```

application, we do this "manually" since we need the name of the form.

```
(ensure-exists-handler 'application)

(setf (handler.rename (gethash 'application *cps-handlers*))
      (lambda (form)
        (destructuring-bind (op &rest args)
            form
          (acond
           ((lookup :flet op) `(,it ,@(mapcar #'cps-rename args)))
           ((lookup :macrolet op) (cps-rename (funcall it (cdr form))))
   #+clisp
   ((and (consp op) (eql 'setf (car op)))
            `(,op ,@(mapcar #'cps-rename args)))
           ((macro-function op)
            (multiple-value-bind (expansion expanded)
                (macroexpand-1 form nil)
              (if expanded
                  (cps-rename expansion)
                  `(,op ,@(mapcar #'cps-rename args)))))
           (t `(,op ,@(mapcar #'cps-rename args)))))))

(setf (handler.transformer (gethash 'application *cps-handlers*))
      (lambda (form k)
        (destructuring-bind (op &rest args)
            form
          (if (and args (some #'cps-requires args))
              (transform-application-args op args k)
              (transform-application-op-only op args k)))))

(setf (handler.requires-checker (gethash 'application *cps-handlers*))
      (lambda (form)
        (list-match-case form
          (((lambda . ?lambda-rest) . ?args)
   (or (cps-requires `(lambda ,@?lambda-rest))
```

```
      (some #'cps-requires ?args)))
         #+clisp
         (((setf ?function-name) . ?args)
          (some #'cps-requires ?args))
         ((?op . ?args)
          (or (lookup :flet ?op)
              (get ?op 'call/cc)
              (progn
                (add-non-cc ?op)
                (some #'cps-requires ?args)))))))

(defun transform-application-args (op args k)
  "need to cps transforme the args for this function call."
  (loop
     with renames = (mapcar (lambda (v)
                              (if (constantp v)
                                  v
                                  (gensym)))
                            (reverse args))
     with code = (cond
                   #+clisp
                   ((and (consp op) (eql 'cl:setf (car op)))
                    (k `(,op ,@(reverse renames))))
                   ((lookup :flet op)
                    `(,op ,k ,@(reverse renames)))
                   ((get op 'call/cc)
                    `(lambda ()
                       (throw 'done
                         (let ((*k* ,k))
                           (,op ,@(reverse renames))))))
                   (t
                    (add-non-cc op)
                    (k `(,op ,@(reverse renames)))))
     for a in (reverse args)
     for a* in renames
     unless (constantp a)
     do (setf code (cps-transform a `(lambda (,a*) ,code)))
     finally (return code)))

(defun transform-application-op-only (op args k)
  (cond
    ((lookup :flet op) `(,op ,k ,@args))
    ((get op 'call/cc) `(lambda ()
                          (throw 'done
                            (let ((*k* ,k))
                              (,op ,@args)))))
    (t
     (add-non-cc op)
     (k `(,op ,@args)))))
```

BLOCK

```
(defcps-transformer block (name &rest body)
  (register :block name k)
  (cps-transform `(progn ,@body) k))

(defcps-requires block (name &rest body)
  (declare (ignore name))
  (cps-requires `(progn ,@body)))

(defcps-rename block (name &rest body)
  (register :block name)
  `(block ,(lookup :block name) ,@(mapcar #'cps-rename body)))
```

RETURN-FROM

```
(defcps-transformer return-from (block-name &optional value)
  (cps-transform value (lookup :block block-name)))

(defcps-requires return-from (block-name &optional value)
  (declare (ignore block-name value))
  ;; since return-from is a control flow operator this should always be cps'd
  t)

(defcps-rename return-from (block-name &optional value)
  (aif (lookup :block block-name)
       `(return-from ,it ,(cps-rename value))
       (error "Can't return outside of cps block: ~S." block-name)))
```

CALL/CC

```
(defun make-call/cc-k (k)
  (lambda (value)
    (catch 'done
      (loop
         for f = (funcall k value)
         then (funcall f)))))

(defcps-transformer call/cc (func)
  (if (and (listp func)
           (eql 'lambda (car func)))
      (destructuring-bind ((kk) &body body)
          (cdr func)
        (cps-transform `(let ((,kk (make-call/cc-k ,k)))
                          ,@body)
                       (if *call/cc-returns*
                           k
                           '#'toplevel-k)))
        (cps-transform `(funcall ,func (make-call/cc-k ,k))
                       (if *call/cc-returns*
                           k
                           '#'toplevel-k))))
```

```
(defcps-requires call/cc (func)
  (declare (ignore func))
  t)

(defcps-rename call/cc (func)
  (if (and (listp func)
           (eql 'lambda (car func)))
      `(call/cc (function
                 (lambda ,(second func)
                  ,@(mapcar #'cps-rename (cddr func)))))
      form))
```

FUNCTION

```
(defcps-transformer function (func)
  (k `(function ,func)))

(defcps-requires function (func)
  (etypecase func
    (symbol nil)
    (cons
     (when (some #'cps-requires (cddr func))
       (error "Attempting to create a LAMBDA which contains a call to call/cc:
~S" func))
     nil)))

(defcps-rename function (func)
  (if (and (consp func)
           (eql 'lambda (car func)))
      `(function (lambda ,(second func)
                  ,@(multiple-value-bind (declares body)
                        (split-into-body-and-declares (cddr func))
                      (if declares
                          `((declare ,@declares) ,@(mapcar #'cps-rename body))
                          (mapcar #'cps-rename body)))))
      (aif (lookup :flet func)
           `(function ,it)
           `(function ,func))))
```

TAGBODY

```
(defcps-transformer tagbody (&rest forms)
  (let (blocks current-block-name current-block-body)
    (flet ((make-block-label (f)
             "Create a labels function definition form named F
             whose body is the CPS transformation of
             CURRENT-BLOCK-BODY and whose continuation calls the
             next block (unless we're the last block in which
             case we continue with K)."
             `(,f ()
                ,(cps-transform
```

```
                `(progn ,@current-block-body)
                (with-unique-names (v)
                  `(lambda (,v)
                     (declare (ignore ,v))
                     ,(if current-block-name
                          ;; since we're traversing in reverse order
                          ;; current-block-name being true implies that
                          ;; we're NOT the last block. so we need to
                          ;; continue with the next block.
                          `(,current-block-name)
                          ;; last block, all done
                          `(funcall ,k nil))))))))
      (dolist (f (reverse forms))
        (if (symbolp f)
            (progn
              (push (make-block-label f) blocks)
              (setf current-block-body nil
                    current-block-name f))
            (push f current-block-body))))
    `(labels ,blocks
       (,(first forms)))))

(defcps-requires tagbody (&rest forms)
  (some #'cps-requires forms))

(defcps-rename tagbody (&rest forms)
  ;; register, and rename, all the tags in this tagbody
  (flet ((rename-tag-symbol (name)
           (register :tag name
                     (gensym (strcat "TAGBODY-TAG-" name "-")))))
    (setf forms (mapcar (lambda (f)
                          (if (symbolp f)
                              (rename-tag-symbol f)
                              f))
                        forms))
    ;; cps-rename all the forms, leave the tags alone.
    (setf forms (mapcar (lambda (f)
                          (if (symbolp f)
                              f
                              (cps-rename f)))
                        forms))
    ;; insert a new tag as the first element, unless one already exists
    (unless (symbolp (first forms))
      (push (gensym "TAGBODY-START-") forms))
    `(tagbody ,@forms)))
```

GO

```
(defcps-transformer go (tag-name)
  `(,tag-name))
```

```
(defcps-requires go (tag-name)
  (declare (ignore tag-name))
  t)

(defcps-rename go (tag-name)
  (aif (lookup :tag tag-name)
       `(go ,it)
       (error "Go outside tag body: ~S." tag-name)))
```

IF

```
(defcps-transformer if (test then &optional else)
  ;; try and produce a minimal transformation of the if
  (if (and (not (cps-requires test))
           (not (cps-requires then))
           (not (cps-requires else)))
      (k `(if ,test ,then ,else))
      (let ((then-branch (if (cps-requires then)
                             (cps-transform then k)
                             (k then)))
            (else-branch (if (cps-requires else)
                             (cps-transform else k)
                             (k else))))
        (if (cps-requires test)
            (cps-transform test (with-unique-names (v)
                                  `(lambda (,v)
                                     (if ,v
                                         ,then-branch
                                         ,else-branch))))
            `(if ,test
                 ,then-branch
                 ,else-branch)))))

(defcps-requires if (test then &optional else)
  (or (cps-requires test)
      (cps-requires then)
      (cps-requires else)))

(defcps-rename if (test then &optional else)
  `(if ,(cps-rename test)
       ,(cps-rename then)
       ,(cps-rename else)))
```

DECLARE

```
(defcps-rename declare (&rest clauses)
  `(declare
    ,@(mapcar (lambda (clause)
                (case (car clause)
                  (type
                   (list* 'type (second clause)
```

```
                        (mapcar #'cps-rename (cddr clause))))
                  ((ignore ignorable dynamic-extent)
                   (list* (first clause)
                        (mapcar #'cps-rename (cdr clause))))
                  ((special)
                   (error "SPECIAL declares not allowed in CALL/CC code."))
                  ((inline notinline optimize) clause)
                  (t
                   ;; assume it's a type-spec
                   (list* (car clause)
                        (mapcar #'cps-rename (cdr clause))))))
              clauses)))

(defcps-requires declare (&rest clauses)
  (declare (ignore clauses))
  nil)

(defcps-transformer declare (&rest clauses)
  `(declare ,@clauses))
```

FLET

```
(defcps-rename flet (binds &body body)
  (let ((orig-env *env*))
    `(flet ,(mapcar (lambda (flet)
                      `(,(register :flet (first flet)) ,(second flet)
                         ,@(let ((*env* orig-env))
                             (mapcar #'cps-rename (cddr flet)))))
                    binds)
       ,@(mapcar #'cps-rename  body))))

(defcps-transformer flet (binds &body body)
  `(flet ,(mapcar #'transform-flet binds)
     ,@(cps-transform-body body k)))

(defcps-requires flet (binds &body body)
  (or (some #'cps-requires-body (mapcar #'cddr binds))
      (cps-requires-body body)))

(defun transform-flet (flet)
  "Returns a new FLET with one extra required arg: the
  continuation."
  (destructuring-bind (name args &body body)
      flet
    `(,name (,(register :flet name) ,@args)
       (declare (ignorable ,(lookup :flet name)))
       ,@(cps-transform-body body (lookup :flet name)))))
```

LABELS

```
(defcps-transformer labels (binds &body body)
  (dolist* (bind binds)
```

```
    (register :flet (first bind)))
  `(labels ,(mapcar #'transform-flet binds)
     ,@(cps-transform-body body k)))

(defcps-rename labels (binds &body body)
  (dolist* ((name args &rest body) binds)
    (declare (ignore args body))
    (register :flet name))
  `(labels
       ,(mapcar (lambda (bind)
                  `(,(lookup :flet (car bind)) ,(second bind)
                     ,@(mapcar #'cps-rename (cddr bind))))
                binds)
     ,@(mapcar #'cps-rename body)))

(defcps-requires labels (binds &body body)
  (or (some #'cps-requires-body (mapcar #'cddr binds))
      (cps-requires-body body)))
```

LET

```
(defcps-transformer let (binds &rest body)
  (if binds
      (with*
          (destructuring-bind (var value)
              (ensure-list (first binds)))
          (multiple-value-bind (decs new-body)
              (extract-associated-declares var body))
          (let ((body (append decs
                              (if (cdr binds)
                                  (list
                                   (cps-transform `(let ,(cdr binds)
                                                    ,@new-body)
                                                  k))
                                  (cps-transform-body new-body k))))))
          (progn
            (if (cps-requires value)
                (cps-transform value `(lambda (,var) ,@body))
                `(let ((,var ,value)) ,@body))))
      `(let () ,@(cps-transform-body body k))))

(defcps-rename let (binds &rest body)
  `(let
       ,(loop
           with orig-env = *env*
           for bind in binds
           for (var value) = (ensure-list bind)
           for new-name = (gensym (strcat "CPS-LET-" (string var) "-"))
           ;; the new bindings will be a gensym and they'll be
           ;; initialized to the value renamed in the old env.
           collect `(,new-name ,(let ((*env* orig-env)) (cps-rename value)))
```

```
          do (register :let var new-name))
     ,@(mapcar #'cps-rename body)))

(defcps-requires let (binds &rest body)
  (or (some #'cps-requires (mapcar #'second (mapcar #'ensure-list binds)))
      (some #'cps-requires body)))
```

LET*

```
(defcps-transformer let* (binds &rest body)
  (if binds
      (let ((var (first (ensure-list (first binds))))
    (value (second (ensure-list (first binds)))))
(multiple-value-bind (decs new-body)
    (extract-associated-declares var body)
  (cps-transform `(let ((,var ,value))
    ,@decs
    (let* ,(cdr binds) ,@new-body))
 k)))
      (cps-transform `(progn ,@body) k)))

(defcps-rename let* (binds &rest body)
  `(let*
       ,(loop
           for bind in binds
           for (var value) = (ensure-list bind)
           for new-name = (gensym (strcat "CPS-LET*-" (string var) "-"))
           collect `(,new-name ,(cps-rename value))
           do (register :let var new-name))
     ,@(mapcar #'cps-rename body)))

(defcps-requires let* (binds &rest body)
  (cps-requires `(let ,binds ,@body)))
```

MACROLET

```
(defcps-rename macrolet (macros &rest body)
  (dolist* ((name args &rest body) macros)
    (register :macrolet name (with-unique-names (macro-form)
                               (eval `(lambda (,macro-form)
                                        (destructuring-bind ,args ,macro-form
                                          ,@body))))))
  (cps-rename `(progn ,@body)))
```

PROGN

```
(defcps-transformer progn (&rest body)
  (cond
    ((cdr body)
     (if (cps-requires (first body))
 (cps-transform (first body)
(with-unique-names (v)
```

```
  `(lambda (,v)
     (declare (ignore ,v))
     ,(cps-transform `(progn ,@(cdr body)) k))))
 `(progn ,(first body)
 ,(cps-transform `(progn ,@(cdr body)) k))))
    (body (cps-transform (first body) k))
    (t (cps-transform NIL k))))

(defcps-requires progn (&rest body)
  (some #'cps-requires body))

(defcps-rename progn (&rest body)
  (cond
    ((cdr body) `(progn ,@(mapcar #'cps-rename body)))
    (body (cps-rename (car body)))
    (t (cps-rename nil))))
```

QUOTE

```
(defcps-rename quote (thing)
  (declare (ignore thing))
  form)

(defcps-transformer quote (thing)
  (declare (ignore thing))
  (k form))

(defcps-requires quote (thing)
  (declare (ignore thing))
  nil)
```

SETQ

```
(defcps-rename setq (&rest vars-values)
  (loop
     for (var value) on vars-values by #'cddr
     collect (acond
               ((lookup :symbol-macrolet var)
                (cps-rename `(setf ,it ,value)))
               ((lookup :let var)
                `(setq ,it ,(cps-rename value)))
               (t
                `(setq ,var ,(cps-rename value))))
     into new-setq
     finally (return `(progn ,@new-setq))))

(defcps-requires setq (&rest vars-values)
  (loop
     for (nil form) on vars-values
     when (cps-requires form)
       return t
     finally (return nil)))
```

```
(defcps-transformer setq (var value)
  ;; due to what the renamer does we can assume that setq will get
  ;; transformed with only two values.
  (if (cps-requires value)
      (cps-transform value (with-unique-names (v)
                             `(lambda (,v)
                                ,(k `(setq ,var ,v)))))
      (k `(setq ,var ,value))))
```

SYMBOL-MACROLET

```
(defcps-rename symbol-macrolet (macros &body body)
  (dolist* ((var expansion) macros)
    (register :symbol-macrolet var expansion))
  (cps-rename `(progn ,@body)))
```

UNWIND-PROTECT

```
(defcps-rename unwind-protect (protected-form &rest cleanups)
  `(unwind-protect
        ,(cps-rename protected-form)
     ,@(mapcar #'cps-rename cleanups)))

(defcps-requires unwind-protect (protected-form &rest cleanups)
  (when (or (cps-requires protected-form)
            (some #'cps-requires cleanups))
    (error "UNWIND-PROTECT can not contain calls to CALL/CC."))
  nil)
```

THE

```
(defcps-rename the (type value)
  `(the ,type ,(cps-rename value)))

(defcps-requires the (type value)
  (declare (ignore type))
  (cps-requires value))

(defcps-transformer the (type value)
  (if (cps-requires value)
      (cps-transform value (with-unique-names (v)
     `(lambda (,v)
,(k `(the ,type ,v)))))
      (k `(the ,type ,value))))
```

7 Reading and Writing files in Comma-Seperated-Values format

Generating CSV files from lisp data

```
(defun princ-csv (items csv-stream
                  &key (quote #\")
                       (separator #\,)
                       (ignore-nulls t)
                       (newline +CR-LF+)
                       (princ #'princ-to-string))
  "Write the list ITEMS to csv-stream."
  (flet ((write-word (word)
           (write-char quote csv-stream)
           (loop
              for char across (funcall princ word)
              if (char= quote char) do
                (progn
                  (write-char quote csv-stream)
                  (write-char quote csv-stream))
              else do
                (write-char char csv-stream))
           (write-char quote csv-stream)))
    (when items
      (write-word (car items))
      (dolist (i (cdr items))
        (write-char separator csv-stream)
        (if ignore-nulls
            (when (not (null i))
              (write-word i))
            (write-word i)))
      (write-sequence newline csv-stream))))

(defun princ-csv-to-string (items)
  (with-output-to-string (csv)
    (princ-csv items csv)))
```

Reading in CSV files

```
(defun parse-csv-string (line &key (separator #\,) (quote #\"))
  "Parse a csv line into a list of strings using @var{seperator}
  as the column seperator and @var{quote} as the string quoting
  character."
  (let ((items '())
        (offset 0)
        (current-word (make-array 20 :element-type 'character
                                     :adjustable t
                                     :fill-pointer 0))
        (state :read-word))
    (labels ((current-char ()
               (aref line offset))
             (current-char= (char)
               (char= (current-char) char))
             (chew-current-word ()
               (push current-word items)
               (setf current-word (make-array 20 :element-type 'character
```

```
                                                :adjustable t
                                                :fill-pointer 0))))
      (loop
         (when (= (length line) offset)
           (ecase state
             (:in-quotes
              (error "Premature end of line."))
             (:read-word
              (chew-current-word)
              (return-from parse-csv-string (nreverse items)))))
         (ecase state
           (:in-quotes
            (if (current-char= quote)
                (progn
                  (when (= (length line) (1+ offset))
                    (error "Premature end of line."))
                  (if (char= (aref line (1+ offset)) quote)
                      (progn
                        (vector-push-extend quote current-word)
                        (incf offset))
                      (setf state :read-word)))
                (vector-push-extend (current-char) current-word)))
           (:read-word
            (if (current-char= quote)
                (setf state :in-quotes)
                (if (current-char= separator)
                    (chew-current-word)
                    (vector-push-extend (current-char) current-word)))))
         (incf offset)))))
```

8 Debugging Utilties

(These were far more usefull in the pre-slime days.)

```
(defmacro ppm1 (form)
  "(pprint (macroexpand-1 ',form)).

NB: C-RET is even shorter."
  `(pprint (macroexpand-1 ',form)))

(defmacro ppm (form)
  `(pprint (macroexpand ',form)))
```

A portable flexable APROPOS implementation

```
(defun apropos-list* (string &key (fbound nil fbound-supplied-p)
                                  (bound nil bound-supplied-p)
                                  (package nil package-supplied-p)
                                  (distance 0 distance-supplied-p))
  (let ((symbols '()))
```

```
    (do-all-symbols (sym)
      (block collect-symbol
        (when fbound-supplied-p
          (when (xor fbound (fboundp sym))
            (return-from collect-symbol)))
        (when bound-supplied-p
          (when (xor bound (boundp sym))
            (return-from collect-symbol)))
        (when package-supplied-p
          (unless (eql package (symbol-package sym))
            (return-from collect-symbol)))
        (when distance-supplied-p
          (unless (and
                   (<= (abs (- (length (symbol-name sym))
                               (length string)))
                       distance)
                   (<= (levenshtein-distance string (symbol-name sym))
                       distance))
            (return-from collect-symbol)))
        (when (not distance-supplied-p)
          ;; regular string= test
          (unless (search string (symbol-name sym) :test #'char-equal)
            (return-from collect-symbol)))
        ;; all the checks we wanted to perform passed.
        (push sym symbols)))
    symbols))

(defun apropos* (&rest apropos-args)
  (flet ((princ-length (sym)
           (if (keywordp sym)
               (+ 1 (length (symbol-name sym)))
               (+ (length (package-name (symbol-package sym)))
                  1
                  (length (symbol-name sym))))))
    (let* ((syms (apply #'apropos-list* apropos-args))
           (longest (apply #'max (mapcar #'princ-length syms))))
      (dolist (sym syms)
        (if (keywordp sym)
            (progn
              (princ ":" *debug-io*)
              (princ (symbol-name sym) *debug-io*))
            (progn
              (princ (package-name (symbol-package sym)) *debug-io*)
              (princ ":" *debug-io*)
              (princ (symbol-name sym) *debug-io*)))
        (princ (make-string (- longest (princ-length sym))
                            :initial-element #\Space)
               *debug-io*)
        (when (fboundp sym)
          (princ " [FUNC] " *debug-io*))
```

```
        (when (boundp sym)
          (princ " [VAR] " *debug-io*))
        (terpri *debug-io*))))
  (values))
```

9 Decimal Arithmetic

Converting to and from external representations

```
(defvar *precision* 2
  "Default precision.")

(defmacro with-precision (prec &body body)
  "Evalute BODY with *precision* bound to @var{prec}."
  (let ((precision (gensym)))
    `(let ((,precision ,prec))
       (assert (integerp ,precision)
       (,precision)
       "Precision must be an integer, not ~S" ,precision)
       (let ((*precision* (10^ ,precision)))
 (declare (special *precision*))
 ,@body))))

(defun decimal-from-float (float
                           &optional (precision *precision*)
                                     (rounding-method #'round-half-up))
  "Convert @var{float} to an exact value with precision
  @var{precision} using @var{rounding-method} to do any
  neccessary rounding."
  (funcall rounding-method float precision))

(defun float-from-decimal (decimal)
  "Convert the exact decimal value @var{decimal} to a (not
  neccassily equal) floating point value."
  (float decimal))
```

Rounding functions

```
(defun round-down (number &optional (precision *precision*))
  "Round towards 0."
  (if (minusp number)
      (round-ceiling number precision)
      (round-floor   number precision)))

(defun round-half-up (number &optional (precision *precision*))
  "Round towards the nearest value allowed with the current
precision. If the current value is exactly halfway between two logal
values round away from 0."
  (multiple-value-bind (value discarded)
      (floor (* number precision))
    (if (<= 1/2 discarded)
```

```
(/ (1+ value) precision)
        (/ value precision))))

(defun round-half-even (number &optional (precision *precision*))
  "Round towards the nearest value allowed with the current
precision. If the current value is exactly halfway between two legal
values round towards the nearest even value."
  (multiple-value-bind (value discarded)
      (floor (* number precision))
    (cond
     ((< discarded 1/2) ;; down
      (/ value precision))
     ((= discarded 1/2) ;; goto even
      (if (evenp value)
  (/ value precision)
  (/ (1+ value) precision)))
     (t ;; (>= discarded 1/2)
      (/ (1+ value) precision)))))

(defun round-ceiling (number &optional (precision *precision*))
  "Round towards positive infintity"
  (/ (ceiling (* number precision)) precision))

(defun round-floor (number &optional (precision *precision*))
  "Round towards negative infinity."
  (/ (floor (* number precision)) precision))

(defun round-half-down (number &optional (precision *precision*))
  "Round towards the nearest legal value. If the current value is
exactly half way between two legal values round towards 0."
  (multiple-value-bind (value discarded)
      (floor number)
    (if (< 1/2 discarded)
(/ (1+ value) precision)
        (/ value precision))))

(defun round-up (number &optional (precision *precision*))
  "Round away from 0."
  (if (minusp number)
      (round-floor number precision)
      (round-ceiling number precision)))
```

10 Defining classes with DEFSTRUCT's syntax

```
(defmacro defclass-struct (name-and-options supers &rest slots)
  "DEFCLASS with a DEFSTRUCT api.

NAME-AND-OPTIONS:

  name-symbol |
```

```
  ( name-symbol [ (:conc-name conc-name ) ]
                [ (:predicate predicate-name ) ] )

SUPERS - a list of super classes passed directly to DEFCLASS.

SLOTS - a list of slot forms:

  name |
  ( name [ init-arg ] [ slot-options* ] )"
  (generate-defclass (first (ensure-list name-and-options))
                     (cdr (ensure-list name-and-options))
                     supers slots))

(defun generate-defclass (class-name class-options supers slots)
  (let ((conc-name nil)
        (predicate nil)
        (predicate-forms nil))
    (loop
       for (option-name . args) in class-options
       do (ecase option-name
            (:conc-name
             (when conc-name
               (error "Can't specify the :CONC-NAME argument more than once."))
             (setf conc-name (first args)))
            (:predicate
             (when predicate
               (error "Can't specify the :PREDICATE argument more than once."))
             (setf predicate (if (eql t (first args))
                                 (intern (strcat class-name :-p)
                                         class-name)
                                 (first args))))))
    (setf slots
          (mapcar
           (lambda (slot-spec)
             (destructuring-bind (name
                                  &optional initform
                                  &rest options)
                 (ensure-list slot-spec)
               `(,name
                 :initform ,initform
                 ,@(when conc-name
                     `(:accessor ,(intern (strcat conc-name name)
                                          (symbol-package conc-name))))
                 :initarg ,(intern (symbol-name name) :keyword)
                 ,@options)))
           slots)
          predicate-forms
          (if predicate
              (with-unique-names (obj)
                `((defmethod ,predicate ((,obj ,class-name)) t)
```

```
                (defmethod ,predicate ((,obj t)) nil)))
            nil))
    `(prog1
         (defclass ,class-name ,supers ,slots)
       ,@predicate-forms)))
```

11 Various flow control operators

11.1 Anaphoric conditionals

```
(defmacro if-bind (var test then &optional else)
  "Anaphoric IF control structure.

VAR (a symbol) will be bound to the primary value of TEST. If
TEST returns a true value then THEN will be executed, otherwise
ELSE will be executed."
  `(let ((,var ,test))
     (if ,var ,then ,else)))

(defmacro aif (test then &optional else)
  "Just like IF-BIND but the var is always IT."
  `(if-bind it ,test ,then ,else))

(defmacro when-bind (var test &body body)
  "Just like @code{WHEN} except @var{var} will be bound to the
  result of @var{test} in @var{body}."
  `(if-bind ,var ,test (progn ,@body)))

(defmacro awhen (test &body body)
  "Just like @code{WHEN} expect the symbol @code{IT} will be
  bound to the result of @var{test} in @var{body}."
  `(when-bind it ,test ,@body))

(defmacro cond-bind (var &body clauses)
  "Just like @code{COND} @var{var} will be bound to the result of
  the first cond which returns true."
  (if clauses
      (destructuring-bind ((test &rest body) &rest others)
          clauses
        `(if-bind ,var ,test
                  (progn ,@body)
                  (cond-bind ,var ,@others)))
      nil))

(defmacro acond (&rest clauses)
  "Just like @code{COND-BIND} except the var is automatically
  @code{IT}."
  `(cond-bind it ,@clauses))

(defmacro aand (&rest forms)
  `(and-bind it ,@forms))
```

```
(defmacro and-bind (var &rest forms)
  (if forms
      `(when-bind ,var ,(first forms)
         (and-bind ,var ,@(cdr forms)))
      t))
```

11.2 Whichever

```
(defmacro whichever (&rest possibilities)
  "Evaluates one (and only one) of its args, which one is chosen at random"
  `(ecase (random ,(length possibilities))
     ,@(loop for poss in possibilities
             for x from 0
             collect (list x poss))))
```

11.3 XOR - The missing conditional

```
(defmacro xor (&rest datums)
  "Evaluates the args one at a time. If more than one arg returns true
  evaluation stops and NIL is returned. If exactly one arg returns
  true that value is retuned."
  (let ((state (gensym "XOR-state-"))
        (block-name (gensym "XOR-block-"))
        (arg-temp (gensym "XOR-arg-temp-")))
    `(let ((,state nil)
           (,arg-temp nil))
       (block ,block-name
         ,@(loop
              for arg in datums
              collect `(setf ,arg-temp ,arg)
              collect `(if ,arg-temp
                           ;; arg is T, this can change the state
                           (if ,state
                               ;; a second T value, return NIL
                               (return-from ,block-name nil)
                               ;; a first T, swap the state
                               (setf ,state ,arg-temp))))
         (return-from ,block-name ,state)))))
```

11.4 Switch

```
(defmacro switch ((obj &key (test #'eql)) &body clauses)
  "Evaluate the first clause whose car satisfies @code{(funcall
  test car obj)}."
  ;; NB: There is no need to do the find-if and the remove here, we
  ;; can just as well do them with in the expansion
  (let ((default-clause (find-if (lambda (c) (eq t (car c))) clauses)))
    (when default-clause
      (setf clauses (remove default-clause clauses :test #'equalp)))
    (let ((obj-sym (gensym))
```

```
          (test-sym (gensym)))
      `(let ((,obj-sym ,obj)
             (,test-sym ,test))
         (cond
           ,@(mapcar (lambda (clause)
                       (let ((keys (ensure-list (car clause)))
                             (form (cdr clause)))
                         `((or ,@(mapcar (lambda (key)
   `(funcall ,test-sym ',key ,obj-sym))
 keys))
   ,@form)))
                     clauses)
           ,@(when default-clause
                   `((t ,@(cdr default-clause)))))))

(defmacro eswitch ((obj &key (test #'eql)) &rest body)
  "Like @code{SWITCH} but signals an error if no clause succeds."
  (let ((obj-sym (gensym)))
    `(let ((,obj-sym ,obj))
       (switch (,obj-sym :test ,test)
               ,@body
               (t
                (error "Unmatched SWITCH. Testing against ~A."
                       ,obj-sym))))))

(defmacro cswitch ((obj &key (test #'eql)) &rest body)
  "Like @code{SWITCH} but signals a continuable error if no
  clause matches."
  (let ((obj-sym (gensym)))
    `(let ((,obj-sym ,obj))
       (switch (,obj-sym :test ,test)
               ,@body
               (t
                (cerror "Unmatched SWITCH. Testing against ~A."
                        ,obj-sym))))))
```

11.5 Eliminating Nesting

```
(defmacro with* (&body body)
  (cond
    ((cddr body)
     (append (first body) `((with* ,@(cdr body)))))
    ((cdr body)
     `(,@(first body) ,(second body)))
    (body (first body))
    (t nil)))
```

12 Convience functions for working with hash tables.

```
(defun build-hash-table (hash-spec inital-contents)
  "Create a hash table containing ``INITAL-CONTENTS``."
  (let ((ht (apply #'make-hash-table hash-spec)))
    (dolist* ((key value) inital-contents)
      (setf (gethash key ht) value))
    ht))

(defmacro deflookup-table
    (name &key (var    (make-lookup-name name "*" name "*"))
               (reader (make-lookup-name name "GET-" name))
               (writer (make-lookup-name name "GET-" name))
               (rem-er (make-lookup-name name "REM-" name))
               (documentation
                (format nil "Global var for the ~S lookup table" name))
               (test 'eql)
               (initial-contents nil))
  "Creates a hash table and the associated accessors."
  ;; if they explicitly pass in NIL we make the name a gensym
  (unless var
    (setf var    (gensym (strcat "var for " name " lookup table "))))
  (unless reader
    (setf reader (gensym (strcat "reader for " name " lookup table "))))
  (unless writer
    (setf writer (gensym (strcat "writer for " name " lookup table "))))
  (assert (symbolp name) (name)
          "The name of the lookup table must be a symbol.")
  (assert (symbolp var) (var)
          "The name of the underlying var must be a symbol.")
  (assert (symbolp reader) (reader)
          "The name of the reader for a lookup table must be a symbol.")
  (assert (symbolp writer) (writer)
          "The name of the writer for a lookup table must be a symbol.")
  `(progn
     (defvar ,var
       (build-hash-table '(:test ,test) ,initial-contents)
       ,documentation)
     (defun ,reader (key &optional default)
       (gethash key ,var default))
     (defun (setf ,writer) (value key)
       (setf (gethash key ,var) value))
     (defun ,rem-er (key)
       (remhash key ,var))
     (list ',name ',var ',reader '(setf ,writer) ',rem-er)))

(defun make-lookup-name (name &rest parts)
  (funcall #'intern-concat parts (symbol-package name)))

(defun hash-to-alist (hash-table)
```

```
  (loop for k being the hash-keys of hash-table
        collect (cons k (gethash k hash-table))))
```

13 HTTP/HTML utilities

13.1 URIs/URLs

```
(defvar *ok-set*
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.,"
  "The list of characters which don't need to be escaped when
  writing URIs.")

(defun escape-as-uri (string)
  "Escapes all non alphanumeric characters in STRING following
  the URI convention. Returns a fresh string."
  (with-output-to-string (escaped)
    (write-as-uri string escaped)))

(defun write-as-uri (string stream)
  (loop
     for char across string
     if (find char *ok-set* :test #'char=)
       do (write-char char stream)
     else do (format stream "%~2,'0X" (char-code char))))

(defun char->hex-value (char)
  "Returns the number associated the hex value of CHAR. CHAR must
  be one of #\0 - #\9, #\a - #\f, #\A - #\F."
  (ecase char
    (#\0 0)
    (#\1 1)
    (#\2 2)
    (#\3 3)
    (#\4 4)
    (#\5 5)
    (#\6 6)
    (#\7 7)
    (#\8 8)
    (#\9 9)
    ((#\a #\A) 10)
    ((#\b #\B) 11)
    ((#\c #\C) 12)
    ((#\d #\D) 13)
    ((#\e #\E) 14)
    ((#\f #\F) 15)))

(defun make-escaped-table ()
  (let ((table (make-array '(16 16)
                           :element-type 'character
                           :initial-element #\\)))
```

```
    (dotimes (i 16)
      (dotimes (j 16)
        (setf (aref table i j) (code-char (+ (* i 16) j)))))
    table))

(defvar *unescape-table* (make-escaped-table))

(defun nunescape-as-uri (string)
  (unescape-as-uri string))

(defun unescape-as-uri (string)
  (with-output-to-string (unescaped)
    (loop
       for index upfrom 0
       while (< index (length string))
       do (case (aref string index)
    (#\% (write-char
                  (aref *unescape-table*
                        (char->hex-value (aref string (incf index)))
                        (char->hex-value (aref string (incf index))))
                  unescaped))
    (#\+ (write-char #\Space unescaped))
    (t (write-char (aref string index) unescaped))))))
```

13.2 HTML

This so blatently wrong its not even funny, and while this is exactly what I need I would do well to start using a "real" html escaping library (there are a couple to choose from).

```
(defun make-html-entities ()
  (let ((ht (make-hash-table :test 'equal)))
    (flet ((add-mapping (char escaped)
             (setf (gethash char ht) escaped
                   (gethash escaped ht) char)))
      (add-mapping #\< "&lt;")
      (add-mapping #\> "&gt;")
      (add-mapping #\& "&amp;")
      (add-mapping #\" "&quot;")
      (add-mapping "a`" "&#224;")
      (add-mapping "a'" "&#225;")
      (add-mapping "e`" "&#232;")
      (add-mapping "e'" "&#233;")
      (add-mapping "i'" "&#236;")
      (add-mapping "i`" "&#237;")
      (add-mapping "o`" "&#242;")
      (add-mapping "o'" "&#243;")
      (add-mapping "u`" "&#249;")
      (add-mapping "u'" "&#250;"))
    ht))
```

```
(defparameter *html-entites* (make-html-entities))

(defun write-as-html (string &key (stream t) (escape-whitespace nil))
  (loop
     for char across string
     do (cond
          ((char= char #\Space)
           (if escape-whitespace
               (princ " " stream)
               (write-char char stream)))
          ((gethash char *html-entites*)
           (princ (gethash char *html-entites*) stream))
          ((> (char-code char) 127)
           (princ "&#x" stream)
           (write (char-code char) :stream stream :base 16)
           (write-char #\; stream))
          (t (write-char char stream)))))

(defun escape-as-html (string &key (escape-whitespace nil))
  (with-output-to-string (escaped)
    (write-as-html string
                   :stream escaped
                   :escape-whitespace escape-whitespace)))

(define-condition html-escape-error (error)
  ((what :accessor html-escape-error.what :initarg :what)))

(define-condition unterminated-html-entity (html-escape-error)
  ())

(define-condition unknown-html-entity (html-escape-error)
  ())

(define-condition unknown-char-escape (warning)
  ((what :accessor html-escape-error.what :initarg :what)))

(defun unescape-as-html (string)
  (with-output-to-string (unescaped)
    (loop
       for offset upfrom 0 below (length string)
       for char = (aref string offset)
       if (char= #\& char)
         do (progn
              (aif (position #\; string :start offset)
                   (let ((escape-tag (subseq string offset (1+ it))))
                     (aif (gethash escape-tag *html-entites*)
                          (progn
                            (princ it unescaped)
                            (incf offset (1- (length escape-tag))))
                          (if (char= #\# (aref escape-tag 1))
                              ;; special code, ignore
```

```
                                (restart-case
                                    (warn 'unknown-char-escape :what escape-tag)
                                  (continue-delete ()
                                    :report "Continue processing, delete this char."
                                    (incf offset (1- (length escape-tag)))))
                                (restart-case
                                    (error 'unknown-html-entity :what escape-tag)
                                  (continue-as-is ()
                                    :report "Continue processing, leaving the string as is."
                                    (write-char #\& unescaped))
                                  (continue-delete ()
                                    :report "Continue processing, delete this entity."
                                    (incf offset (1- (length escape-tag))))))))
                    (restart-case
                        (error 'unterminated-html-entity
                               :what (subseq string offset
                                             (min (+ offset 20)
                                                  (length string))))
                      (continue-as-is ()
                        :report "Continue processing, leave the string as is."
                        (write-char #\& unescaped)))))
       else do (write-char char unescaped))))
```

14 Utilites for file system I/O

```
(defmacro with-input-from-file ((stream-name file-name &rest args) &body body)
  "Evaluate @var{body} with @var{stream-name} bound to an
  input-stream from file @var{file-name}. @var{args} is passed
  directly to @code{OPEN}."
  (when (member :direction args)
    (error "Can't specifiy :DIRECTION in WITH-INPUT-FILE."))
  `(with-open-file (,stream-name ,file-name :direction :input ,@args)
     ,@body))

(defmacro with-output-to-file ((stream-name file-name &rest args) &body body)
  "Evaluate @var{body} with @var{stream-name} to an output stream
  on the file named @var{file-name}. @var{args} is sent as is to
  the call te @var{OPEN}."
  (when (member :direction args)
    (error "Can't specifiy :DIRECTION in WITH-OUTPUT-FILE."))
  `(with-open-file (,stream-name ,file-name :direction :output ,@args)
     ,@body))

(defun read-string-from-file (pathname &key (buffer-size 4096)
                                            (element-type 'character))
  "Return the contents of @var{pathname} as a string."
  (with-input-from-file (file-stream pathname)
    (with-output-to-string (datum)
      (let ((buffer (make-array buffer-size :element-type element-type)))
(loop for bytes-read = (read-sequence buffer file-stream)
```

```
      do (write-sequence buffer datum :start 0 :end bytes-read)
      while (= bytes-read buffer-size))))))

(defun write-string-to-file (string pathname &key (if-exists :error))
  "Write @var{string} to @var{pathname}."
  (with-output-to-file (file-stream pathname :if-exists if-exists)
    (write-sequence string file-stream)))

(defun copy-file (from to &key (if-to-exists :supersede)
                               (element-type '(unsigned-byte 8)))
  (with*
   (with-input-from-file (input  from :element-type element-type))
   (with-output-to-file  (output to   :element-type element-type
                                      :if-exists if-to-exists))
   (progn
     (loop
        with buffer-size = 4096
        with buffer = (make-array buffer-size :element-type element-type)
        for bytes-read = (read-sequence buffer input)
        while (= bytes-read buffer-size)
        do (write-sequence buffer output)
        finally (write-sequence buffer output :end bytes-read)))))
```

15 Higher order functions

```
(defun compose (f1 &rest functions)
  "Returns a function which applies the arguments in order.

 (funcall (compose #'list #'+) 1 2 3) ==> (6)"
  (case (length functions)
    (0 f1)
    (1 (lambda (&rest args)
         (funcall f1 (apply (car functions) args))))
    (2 (lambda (&rest args)
         (funcall f1
                  (funcall (first functions)
                           (apply (second functions) args)))))
    (3 (lambda (&rest args)
         (funcall f1
                  (funcall (first functions)
                           (funcall (second functions)
                                    (apply (third functions) args))))))
    (t
     (let ((funcs (nreverse (cons f1 functions))))
       (lambda (&rest args)
         (loop
            for f in funcs
            for r = (multiple-value-list (apply f args))
               then (multiple-value-list (apply f r))
            finally (return r)))))))
```

```
(defun conjoin (&rest predicates)
  (case (length predicates)
    (0 (constantly t))
    (1 (car predicates))
    (2 (lambda (&rest args)
         (and (apply (first predicates) args)
              (apply (second predicates) args))))
    (3 (lambda (&rest args)
         (and (apply (first predicates) args)
              (apply (second predicates) args)
              (apply (third predicates) args))))
    (t
     (lambda (&rest args)
       (loop
          for p in predicates
          for val = (apply p args)
          while val
          finally (return val))))))

(defun curry (function &rest initial-args)
  "Returns a function which will call FUNCTION passing it
  INITIAL-ARGS and then any other args.

 (funcall (curry #'list 1) 2) ==> (list 1 2)"
  (lambda (&rest args)
    (apply function (append initial-args args))))

(defun rcurry (function &rest initial-args)
  "Returns a function which will call FUNCTION passing it the
  passed args and then INITIAL-ARGS.

 (funcall (rcurry #'list 1) 2) ==> (list 2 1)"
  (lambda (&rest args)
    (apply function (append args initial-args))))

(defun noop (&rest args)
  "Do nothing."
  (declare (ignore args))
  (values))

(defmacro lambda-rec (name args &body body)
  "Just like lambda except BODY can make recursive calls to the
  lambda by calling the function NAME."
  `(lambda ,args
     (labels ((,name ,args ,@body))
       (,name ,@args))))
```

15.1 Just for fun

```
(defun y (lambda)
  (funcall (lambda (f)
```

```
            (funcall (lambda (g)
                       (funcall g g))
                     (lambda (x)
                       (funcall f
                                (lambda ()
                                  (funcall x x))))))
          lambda))
```

16 Working with lists

```
(defmacro dolist* ((iterator list &optional return-value) &body body)
  "Like DOLIST but destructuring-binds the elements of LIST.

If ITERATOR is a symbol then dolist* is just like dolist EXCEPT
that it creates a fresh binding."
  (if (listp iterator)
      (let ((i (gensym "DOLIST*-i-")))
        `(dolist (,i ,list ,return-value)
           (destructuring-bind ,iterator ,i
             ,@body)))
      `(dolist (,iterator ,list ,return-value)
         (let ((,iterator ,iterator))
           ,@body))))

(defun ensure-list (thing)
  "Returns THING as a list.

If THING is already a list (as per listp) it is returned,
otherwise a one element list containing THING is returned."
  (if (listp thing)
      thing
      (list thing)))

(defun ensure-cons (thing)
  (if (consp thing)
      thing
      (cons thing nil)))

(defun partition (list &rest lambdas)
  "Split LIST into sub lists according to LAMBDAS.

Each element of LIST will be passed to each element of LAMBDAS,
the first function in LAMBDAS which returns T will cause that
element to be collected into the corresponding list.

Examples:

 (partition '(1 2 3) #'oddp #'evenp) => ((1 3) (2))

 (partition '(1 2 3) #'oddp t) => ((1 3) (1 2 3))
```

```
 (partition '(1 2 3) #'oddp #'stringp) => ((1 3) nil)"
  (let ((collectors (mapcar (lambda (predicate)
                              (cons (case predicate
                                      ((t :otherwise)
                                       (constantly t))
                                      ((nil)
                                       (constantly nil))
                                      (t predicate))
                                    (make-collector)))
                            lambdas)))
    (dolist (item list)
      (dolist* ((test-func . collector-func) collectors)
        (when (funcall test-func item)
          (funcall collector-func item))))
    (mapcar #'funcall (mapcar #'cdr collectors))))

(defun partitionx (list &rest lambdas)
  (let ((collectors (mapcar (lambda (l)
                              (cons (if (and (symbolp l)
     (member l (list :otherwise t)
                                                     :test #'string=))
                                        (constantly t)
                                        l)
                                    (make-collector)))
                            lambdas)))
    (dolist (item list)
      (block item
        (dolist* ((test-func . collector-func) collectors)
          (when (funcall test-func item)
            (funcall collector-func item)
            (return-from item)))))
    (mapcar #'funcall (mapcar #'cdr collectors))))

(defmacro dotree ((name tree &optional ret-val) &body body)
  "Evaluate BODY with NAME bound to every element in TREE. Return RET-VAL."
  (with-unique-names (traverser list list-element)
    `(progn
       (labels ((,traverser (,list)
                  (dolist (,list-element ,list)
                    (if (consp ,list-element)
                        (,traverser ,list-element)
                        (let ((,name ,list-element))
                          ,@body)))))
         (,traverser ,tree)
         ,ret-val))))

(define-modify-macro push* (&rest items)
  (lambda (list &rest items)
    (dolist (i items)
      (setf list (cons i list)))
```

```
    list)
  "Pushes every element of ITEMS onto LIST. Equivalent to calling PUSH
  with each element of ITEMS.")

(defun proper-list-p (object)
  "Tests whether OBJECT is properlist.

A proper list is a non circular cons chain whose last cdr is eq
to NIL."
  (or
   (null object)
   (and (consp object)
;; check if the last cdr of object is null. deal with
;; circular lists.
(loop
 for turtoise = object then (cdr turtoise)
 for hare = (cdr object) then (cddr hare)
 ;; we need to agressivly check hare's cdr so that the call to
 ;; cddr doesn't signal an error
 when (eq turtoise hare) return nil
 when (null turtoise) return t
 when (null hare) return t
 when (not (consp hare)) return nil
 when (null (cdr hare)) return t
 when (not (consp (cdr hare))) return nil
 when (null (cddr hare)) return t
 when (not (consp (cddr hare))) return nil))))
```

16.1 Simple list matching based on code from Paul Graham's On Lisp.

```
(defmacro acond2 (&rest clauses)
  (if (null clauses)
      nil
      (with-unique-names (val foundp)
        (destructuring-bind ((test &rest progn) &rest others)
            clauses
          `(multiple-value-bind (,val ,foundp)
               ,test
             (if (or ,val ,foundp)
                 (let ((it ,val))
                   (declare (ignorable it))
                   ,@progn)
                 (acond2 ,@others)))))))

(defun varsymp (x)
  (and (symbolp x) (eq (aref (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
```

```
            (aif (assoc x binds)
                 (or (recbind (cdr it) binds)
                     it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))

(defun list-match (x y &optional binds)
  (acond2
    ((or (eql x y) (eql x '_) (eql y '_))
     (values binds t))
    ((binding x binds) (list-match it y binds))
    ((binding y binds) (list-match x it binds))
    ((varsymp x) (values (cons (cons x y) binds) t))
    ((varsymp y) (values (cons (cons y x) binds) t))
    ((and (consp x) (consp y) (list-match (car x) (car y) binds))
     (list-match (cdr x) (cdr y) it))
    (t (values nil nil))))

(defun vars (match-spec)
  (let ((vars nil))
    (labels ((find-vars (spec)
               (cond
                 ((null spec) nil)
                 ((varsymp spec) (push spec vars))
                 ((consp spec)
                  (find-vars (car spec))
                  (find-vars (cdr spec))))))
      (find-vars match-spec))
    (delete-duplicates vars)))

(defmacro list-match-case (target &body clauses)
  (if clauses
      (destructuring-bind ((test &rest progn) &rest others)
          clauses
        (with-unique-names (tgt binds success)
          `(let ((,tgt ,target))
             (multiple-value-bind (,binds ,success)
                 (list-match ,tgt ',test)
               (declare (ignorable ,binds))
               (if ,success
                   (let ,(mapcar (lambda (var)
                                   `(,var (cdr (assoc ',var ,binds))))
                                 (vars test))
                     (declare (ignorable ,@(vars test)))
                     ,@progn)
                   (list-match-case ,tgt ,@others))))))
      nil))
```

17 A Trivial logging facility

A logger is a way to have the system generate a text message and have that messaged saved somewhere for future review. Logging can be used as a debugging mechinasm or for just reporting on the status of a system.

Logs are sent to a particular log category, each log category sends the messages it recieves to its handlers. A handler's job is to take a message and write it somewhere. Log categories are organized in a hierarchy and messages sent to a log cateogry will also be sent to that category's ancestors.

Each log category has a log level which is used to determine whether are particular message should be processed or not. Categories inheirt their log level fro their ancestors. If a category has multiple fathers its log level is the min of the levels of its fathers.

17.1 Log Levels

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defconstant +dribble+ 0)
  (defconstant +debug+   1)
  (defconstant +info+    2)
  (defconstant +warn+    3)
  (defconstant +error+   4)
  (defconstant +fatal+   5)

  (deflookup-table logger))
```

17.2 Log Categories

```
(defclass log-category ()
  ((ancestors :initform '()     :accessor ancestors :initarg :ancestors)
   (childer   :initform '()     :Accessor childer   :initarg :childer)
   (appenders :initform '()     :accessor appenders :initarg :appenders)
   (level     :initform +debug+ :initarg :level :accessor level)
   (name      :initarg :name :accessor name)))

(defmethod shared-initialize :after ((l log-category) slot-names
                                     &key ancestors &allow-other-keys)
  (declare (ignore slot-names))
  (dolist (anc ancestors)
    (pushnew l (childer anc) :test (lambda (a b)
     (eql (name a) (name b))))))

(defmethod enabled-p ((cat log-category) level)
  (>= level (log.level cat)))

(defmethod log.level ((cat log-category))
  (with-slots (level) cat
    (or level
        (if (ancestors cat)
            (loop for ancestor in (ancestors cat)
                  minimize (log.level ancestor))
            (error "Can't determine level for ~S" cat)))))
```

```
(defmethod (setf log.level) (new-level (cat log-category)
                             &optional (recursive t))
  "Change the log level of CAT to NEW-LEVEL. If RECUSIVE is T the
  setting is also applied to the sub categories of CAT."
  (setf (slot-value cat 'level) new-level)
  (when recursive
    (dolist (child (childer cat))
      (setf (log.level child) new-level))))
```

17.3 Handling Messages

```
(defgeneric handle (category message level))

(defmethod handle ((cat log-category) message level)
  (if (appenders cat)
      ;; if we have any appenders send them the message
      (dolist (appender (appenders cat))
(append-message cat appender message level))
      ;; send the message to our ancestors
      (dolist (ancestor (ancestors cat))
(handle ancestor message level))))

(defgeneric append-message (category log-appender message level))
```

17.3.1 Stream log appender

```
(defclass stream-log-appender ()
  ((stream :initarg :stream :accessor log-stream))
  (:documentation "Human readable to the console logger."))

(defmethod append-message ((category log-category) (s stream-log-appender)
                           message level)
  (multiple-value-bind (second minute hour date month year day)
      (decode-universal-time (get-universal-time))
    (declare (ignore date))
    (restart-case
        (progn
          (format (log-stream s)
                  "~4,'0D-~2,'0D-~2,'0DT~2,'0D:~2,'0D.~2,'0D ~S ~S: "
                  year month day hour minute second
                  level (name category))
          (format (log-stream s) "~A~%" message))
      (use-*debug-io* ()
        :report "Use the current value of *debug-io*"
        (setf (log-stream s) *debug-io*)
        (append-message category s message level))
      (use-*standard-output* ()
        :report "Use the current value of *standard-output*"
        (setf (log-stream s) *standard-output*)
        (append-message category s message level))
      (silence-logger ()
```

```
        :report "Ignore all future messages to this logger."
        (setf (log-stream s) (make-broadcast-stream))))))

(defun make-stream-log-appender (&optional (stream *debug-io*))
  (make-instance 'stream-log-appender :stream stream))

(defclass file-log-appender (stream-log-appender)
  ((log-file :initarg :log-file :accessor log-file))
  (:documentation "Logs to a file. the output of the file logger
  is not meant to be read directly by a human."))

(defmethod append-message ((category log-category) (appender file-log-appender)
                           message level)
  (with-output-to-file (log-file (log-file appender)
 :if-exists :append
 :if-does-not-exist :create)
    (let ((*package* (find-package :it.bese.arnesi)))
      (format log-file "(~S ~D ~S ~S)~%" level (get-universal-time) (name category) messag

(defun make-file-log-appender (file-name)
  (make-instance 'file-log-appender :log-file file-name))
```

17.4 Creating Loggers

```
(defmacro deflogger (name ancestors &key level appender appenders documentation)
  (declare (ignore documentation))
  (when appender
    (setf appenders (append appenders (list appender))))
  (let ((ancestors (mapcar (lambda (ancestor-name)
     `(or (get-logger ',ancestor-name)
  (error "Attempt to define a sub logger of the undefined logger ~S."
 ',ancestor-name)))
   ancestors)))
    (flet ((make-log-helper (suffix level)
     `(defmacro ,(intern (strcat name "." suffix)) (message-control &rest message-args)
`(when (enabled-p (get-logger ',',name) ',',level)
   ,(if message-args
`(handle (get-logger ',',name) (format nil ,message-control ,@message-args) ',',level)
`(handle (get-logger ',',name) ,message-control ',',level))))))
      `(progn
 (setf (get-logger ',name) (make-instance 'log-category
  :name ',name
  :level ,level
  :appenders (list ,@appenders)
  :ancestors (list ,@ancestors)))
 ,(make-log-helper '#:dribble '+dribble+)
 ,(make-log-helper '#:info '+info+)
 ,(make-log-helper '#:warn '+warn+)
 ,(make-log-helper '#:error '+error+)
 ,(make-log-helper '#:fatal '+fatal+)))))
```

-*- lisp -*-

```
(in-package :it.bese.arnesi)
```

18 A fare-like matchingfacility

The code is written in CPS style, it's hard to understand at first but once you "get it" it's actually quite simple. Basically the idea is that at every point during a match one of two things can happen, the match can succedd or it can fail. What we do is we pass every match two functions (closures usually), one which specifies what to if it succedds and one which specifies what to do if it fails. These two closures can refer to the original matchs parameter and hence we can easily "backtrack" if we fail. Another important aspect is that we explcitly pass the target against which to match, if we didn't do this it would be impossible to really backtrack.

18.1 The matching and compiling enviroment

```
(deflookup-table match-handler
  :documentation "Table mapping symbol names to the matching function")

(defstruct (match-state (:conc-name ||))
  target
  bindings
  matched)

(defun copy-state (orig-state
   &key (target nil target-supp)
        (bindings nil bindings-supp)
(matched nil matched-supp))
  "Make a copy ORIG-STATE."
  (make-match-state :target (if target-supp
target
           (target orig-state))
    :bindings (if bindings-supp
  bindings
  (bindings orig-state))
    :matched (if matched-supp
 matched
         (matched orig-state))))

(defmacro def-matcher (name args &body body)
  `(progn (setf (get-match-handler ',name)
(lambda ,args ,@body))
  ',name))

(defmacro def-matcher-macro (name args &body body)
  `(progn (setf (get-match-handler ',name)
(lambda ,args
  (%make-matcher (progn ,@body))))
  ',name))
```

18.2 Matching

```
(defun make-matcher (spec)
  "Create a matcher function from SPEC."
  (let ((%bind-vars% '()))
    (declare (special %bind-vars%))
    (values (%make-matcher spec)
    %bind-vars%)))

(defun %make-matcher (spec)
  ;; NIL means many different things, deal with it explicitly
  (if (eql nil spec)
      (%make-matcher `(:EQL ,spec))
      (if (listp spec)
          (aif (get-match-handler (car spec))
               (apply it (cdr spec))
               (error "Don't know how to handle ~S" spec))
          (aif (get-match-handler spec)
               ;; we allow :X as a an abbreviation for (:x)
               (funcall it)
               (if (and (symbolp spec)
                        (not (keywordp spec)))
                   (%make-matcher `(:BIND :ANYTHING ,spec))
                   (if (constantp spec)
                       (%make-matcher `(:EQL ,spec))
                       (error "Don't know how to deal with ~S" spec)))))))

(defun match (matcher target)
  "Attempt to match MATCHER against TARGET. MATCHER can be either a
function or a list."
  (if (functionp matcher)
      (funcall matcher
       (make-match-state :target target
 :bindings '()
 :matched nil)
       (lambda (s k q)
 (declare (ignore k q))
 (return-from match (values t
    (matched s)
    (bindings s))))
       (lambda (s k q)
 (declare (ignore s k q))
 (return-from match (values nil nil nil))))
      (match (make-matcher matcher) target)))

(defmacro match-case (form &rest clauses)
  "NB: the clauses wil be compiled at macro expansion time."
  (when clauses
    (destructuring-bind ((spec &rest body) &rest other-clauses) clauses
      (with-unique-names (form-sym matched-p dummy bindings)
        (multiple-value-bind (matcher-func vars)
```

```
    (make-matcher spec)
  (declare (ignore matcher-func))
  `(let ((,form-sym ,form))
     (multiple-value-bind (,matched-p ,dummy ,bindings)
 (match (make-matcher ',spec) ,form-sym)
       (declare (ignore ,dummy) (ignorable ,bindings))
       (if ,matched-p
   (let ,vars
     ,@(mapcar (lambda (var-name)
 `(setf ,var-name (cdr (assoc ',var-name ,bindings))))
       vars)
     ,@body)
   (match-case ,form-sym ,@other-clauses)))))))))
```

18.3 Matching forms

```
(def-matcher :BIND (spec var)
  "The :bind matcher attempts to match MATCHER and bind whatever
   MATCHER consumnd to VAR. group is equivalent to SPEC except the value
   of matched when spec has matched will be bound to var."
  (declare (special %bind-vars%))
  (push var %bind-vars%)
  (let ((spec-matcher (%make-matcher spec)))
    (lambda (s k q)
      (funcall spec-matcher s
       (lambda (s. k. q.)
 (declare (ignore k.))
 ;; SPEC succeded, bind var
 (funcall k (copy-state s. :bindings (cons (cons var (matched s.)) (bindings s.)))
  k q.))
       q))))

(def-matcher :REF (var &key (test #'eql))
  (lambda (s k q)
    (if (and (assoc var (bindings s))
     (funcall test (target s) (cdr (assoc var (bindings s)))))
(funcall k (copy-state s :matched (target s))
 k q)
        (funcall q s k q))))

(def-matcher :ALTERNATION (a-spec b-spec)
  (let ((a (%make-matcher a-spec))
(b (%make-matcher b-spec)))
    (lambda (s k q)
      ;; first try A
      (funcall a s k
       ;; a failed, try B
       (lambda (s. k. q.)
 (declare (ignore s. k. q.))
 (funcall b s k q))))))
```

```
(def-matcher-macro :ALT (&rest possibilities)
  (case (length possibilities)
    (0 `(:FAIL))
    (1 (car possibilities))
    (t `(:ALTERNATION ,(car possibilities) (:ALT ,@(cdr possibilities))))))

(def-matcher :FAIL ()
  (lambda (s k q)
    (funcall q s k q)))

(def-matcher :NOT (match)
  (let ((m (%make-matcher match)))
    (lambda (s k q)
      (funcall m s q k))))

(def-matcher :ANYTHING ()
  (lambda (s k q)
    (funcall k (copy-state s :matched (target s))
     k q)))
```

18.4 Matching within a sequence

```
(defun next-target ()
  (declare (special *next-target*))
  (funcall *next-target*))

(defun make-greedy-star (m)
  (lambda (s k q)
    (if (funcall m (target s))
        (funcall (make-greedy-star m) (copy-state s
                                                  :matched (target s)
                                                  :target (next-target))
                 k (lambda (s. k. q.)
                     (declare (ignore k. s.))
                     (funcall k s k q.)))
        (funcall q s k q))))

(def-matcher :GREEDY-STAR (match)
  (make-greedy-star (%make-matcher match)))
```

18.5 The actual matching operators

All of the above allow us to build matchers but non of them actually match anything.

```
(def-matcher :TEST (predicate)
  "Matches if the current matches satisfies PREDICATE."
  (lambda (s k q)
    (if (funcall predicate (target s))
(funcall k (copy-state s :matched (target s))
 k q)
        (funcall q s k q))))
```

```
(def-matcher-macro :TEST-NOT (predicate)
  `(:NOT (:TEST ,predicate)))

(def-matcher-macro :SATISFIES-P (predicate)
  `(:TEST ,(lambda (target) (funcall predicate target))))

(def-matcher-macro :EQ (object)
  `(:TEST ,(lambda (target) (eq object target))))

(def-matcher-macro :EQL (object)
  `(:TEST ,(lambda (target) (eql object target))))

(def-matcher-macro cl:QUOTE (constant)
  `(:EQL ,constant))

(def-matcher-macro :EQUAL (object)
  `(:TEST ,(lambda (target) (equal object target))))

(def-matcher-macro :EQUALP (object)
  `(:TEST ,(lambda (target) (equalp object target))))

(def-matcher :CONS (car-spec cdr-spec)
  (let ((car (%make-matcher car-spec))
(cdr (%make-matcher cdr-spec)))
    (lambda (s k q)
      (if (consp (target s))
  (funcall car (copy-state s :target (car (target s)))
   (lambda (s. k. q.)
     (declare (ignore k.))
     ;; car matched, try cdr
     (funcall cdr (copy-state s. :target (cdr (target s)))
      (lambda (s.. k.. q..)
(declare (ignore k.. q..))
;; cdr matched, ok, we've matched!
(funcall k (copy-state s.. :matched (target s))
 k q))
      q.))
   q)
  (funcall q s k q)))))

(def-matcher-macro :LIST (&rest items)
  `(:LIST* ,@items NIL))

(def-matcher-macro :LIST* (&rest items)
  (case (length items)
    (1 (car items))
    (2 `(:CONS ,(first items) ,(second items)))
    (t
     `(:CONS ,(first items) (:LIST* ,@(cdr items))))))
```

19 Mesing with the MOP

```
(defmacro with-class-slots ((object class-name &key except) &body body)
  "Execute BODY as if in a with-slots form containig _all_ the
  slots of (find-clas CLASS-NAME). This macro, which is something
  of an ugly hack, inspects the class named by CLASS-NAME at
  macro expansion time. Should the class CLASS-NAME change form
  containing WITH-CLASS-SLOTS must be recompiled. Should the
  class CLASS-NAME not be available at macro expansion time
  WITH-CLASS-SLOTS will fail."
  (declare (ignore object class-name except body))
  (error "Not yet implemented."))
```

19.1 wrapping-standard method combination

```
(define-method-combination wrapping-standard
    (&key (around-order :most-specific-first)
          (before-order :most-specific-first)
          (primary-order :most-specific-first)
          (after-order :most-specific-last)
          (wrapping-order :most-specific-last))
  ((around (:around))
   (before (:before))
   (wrapping (:wrapping))
   (primary () :required t)
   (after (:after)))
  "Same semantics as standard method combination but allows
\"wrapping\" methods which get called before :around methods and
in :most-specific-last order.

Ordering of methods:

 (around
    (before)
    (wrapping
      (primary))
    (after))

:around, :wrapping and :primary methods call the next least/most
specific method via call-next-method (as in standard method
combination).

The order of method application is settable via parameters to
the :method-combination argument of the defgeneric which uses
this method combination.

The various WHATEVER-order keyword arguments set the order in
which the methods are called and be set to either
:most-specific-last or :most-specific-first."
  (labels ((effective-order (methods order)
```

```
          (ecase order
            (:most-specific-first methods)
            (:most-specific-last (reverse methods))))
        (call-methods (methods)
          (mapcar (lambda (meth) `(call-method ,meth))
                  methods)))
    (let* (;; reorder the methods based on the -order arguments
           (around (effective-order around around-order))
           (wrapping (effective-order wrapping wrapping-order))
           (before (effective-order before before-order))
           (primary (effective-order primary primary-order))
           (after (effective-order after after-order))
           ;; inital value of the effective call is a call its primary
           ;; method(s)
           (form (case (length primary)
                   (1 `(call-method ,(first primary)))
                   (t `(call-method ,(first primary) ,(rest primary))))))
      (when wrapping
        ;; wrap form in call to the wrapping methods
        (setf form `(call-method ,(first wrapping)
                                 (,@(rest wrapping) (make-method ,form)))))
      (when before
        ;; wrap FORM in calls to its before methods
        (setf form `(progn
                      ,@(call-methods before)
                      ,form)))
      (when after
        ;; wrap FORM in calls to its after methods
        (setf form `(multiple-value-prog1
                        ,form
                      ,@(call-methods after))))
      (when around
        ;; wrap FORM in calls to its around methods
        (setf form `(call-method ,(first around)
                                 (,@(rest around)
                                  (make-method ,form)))))
      form)))
```

20 A MOP compatability protocol

```
(defpackage :it.bese.arnesi.mopp
    (:nicknames :mopp)
    (:documentation "A MOP compatabilitly layer.

This package wraps the various similar but slightly different MOP
APIs. All the MOP symbols are exported (even those which are
normally exported from the common-lisp package) though not all
maybe be properly defined on all lisps.
```

```
The name of the library in an acronym for \"the Meta Object
Protocol Package\".

This package is nominally part of the arnesi utility library but
has been written so that this single file can be included in
other applications without requiring the rest of the arnesi
library.

Implementation Notes:

1) The mopp package also exports the function
   SLOT-DEFINITION-DOCUMENTATION which while not strictly part of
   the MOP specification really should be and is implementened on
   most systems.

2) On Lispworks (tested only lightly) the MOPP package
   implementes an eql-specializer class and defines a version of
   method-specializers built upon clos:method-specializers which
   returns them.")
    (:use)
    (:export
     ;; classes
     #:standard-object
     #:funcallable-standard-object
     #:metaobject
     #:generic-function
     #:standard-generic-function
     #:method
     #:standard-method
     #:standard-accessor-method
     #:standard-reader-method
     #:standard-writer-method
     #:method-combination
     #:slot-definition
     #:direct-slot-definition
     #:effective-slot-definition
     #:standard-slot-definition
     #:standard-direct-slot-definition
     #:standard-effective-slot-definition
     #:specializer
     #:eql-specializer
     #:class
     #:built-in-class
     #:forward-referenced-class
     #:standard-class
     #:funcallable-standard-class
     ;; Taken from the MOP dictionary
     #:accessor-method-slot-definition
     #:add-dependent
     #:add-direct-method
```

```
#:add-direct-subclass
#:add-method
#:allocate-instance
#:class-default-initargs
#:class-direct-default-initargs
#:class-direct-slots
#:class-direct-subclasses
#:class-direct-superclasses
#:class-finalized-p
#:class-name
#:class-precedence-list
#:class-prototype
#:class-slots
#:compute-applicable-methods
#:compute-applicable-methods-using-classes
#:compute-class-precedence-list
#:compute-default-initargs
#:compute-discriminating-function
#:compute-effective-method
#:compute-effective-slot-definition
#:compute-slots
#:direct-slot-definition-class
#:effective-slot-definition-class
#:ensure-class-using-class
#:ensure-generic-function
#:ensure-generic-function-using-class
#:eql-specializer-object
#:extract-lambda-list
#:extract-specializer-names
#:finalize-inheritance
#:find-method-combination
#:funcallable-standard-instance-access
#:generic-function-argument-precedence-order
#:generic-function-declarations
#:generic-function-lambda-list
#:generic-function-method-class
#:generic-function-method-combination
#:generic-function-methods
#:generic-function-name
#:intern-eql-specializer
#:make-instance
#:make-method-lambda
#:map-dependents
#:method-function
#:method-generic-function
#:method-lambda-list
#:method-specializers
#:method-qualifiers
#:reader-method-class
#:remove-dependent
```

```
      #:remove-direct-method
      #:remove-direct-subclass
      #:remove-method
      #:set-funcallable-instance-function
      #:slot-boundp-using-class
      #:slot-definition-allocation
      #:slot-definition-documentation
      #:slot-definition-initargs
      #:slot-definition-initform
      #:slot-definition-initfunction
      #:slot-definition-location
      #:slot-definition-name
      #:slot-definition-readers
      #:slot-definition-writers
      #:slot-definition-type
      #:slot-makunbound-using-class
      #:slot-value-using-class
      #:specializer-direct-generic-functions
      #:specializer-direct-methods
      #:standard-instance-access
      #:update-dependent
      #:validate-superclass
      #:writer-method-class))

(defpackage :it.bese.arnesi.mopp%internals
  (:use :common-lisp))

(in-package :it.bese.arnesi.mopp%internals)

(defgeneric provide-mopp-symbol (symbol implementation)
  (:documentation "Provide the implementation of the MOP symbol SYMBOL.

SYMBOL - One of the external symbols of the package it.bese.arnesi.mopp

IMPLEMENTATION - A keyword indetifying the implementation, one
of: :OPENMCL, :SBCL, :CMU, :LISPWORKS, :ALLEGRO.

Do \"something\" such that the external symbol SYMBOL in the mopp
package provides the sematics for the like named symbol in the
MOP. Methods defined on this generic function are free to
destructivly modify SYMBOL (and the mopp package) as long as when
the method terminates there is a symbol with the same name as
SYMBOL exported form the package mopp.

Methods must return a true value if they have successfully
provided SYMBOL and nil otherwise."))

(defun import-to-mopp (symbol)
  (let ((sym (find-symbol (string symbol) :it.bese.arnesi.mopp)))
    (when sym
      (unexport sym :it.bese.arnesi.mopp)
```

```
      (unintern sym :it.bese.arnesi.mopp)))
  (import symbol :it.bese.arnesi.mopp)
  (export symbol :it.bese.arnesi.mopp)
  t)
```

OpenMCL

```
(defmethod provide-mopp-symbol ((symbol symbol)
                                (implementation (eql :openmcl)))
  "Provide MOP symbols for OpenMCL.

All of OpenMCL's MOP is defined in the CCL package."
  (when (find-symbol (string symbol) :ccl)
    (import-to-mopp (find-symbol (string symbol) :ccl))))
```

SBCL

```
(defmethod provide-mopp-symbol ((symbol symbol)
                                (implementation (eql :sbcl)))
  (when (find-symbol (string symbol) :sb-mop)
    (import-to-mopp (find-symbol (string symbol) :sb-mop))))

(defmethod provide-mopp-symbol ((symbol (eql 'mopp:slot-definition-documentation))
                                (implementation (eql :sbcl)))
  "Provide SLOT-DEFINITION-DOCUMENTATION for SBCL.

On SBCL SLOT-DEFINITION-DOCUMENTATION is just a call to
sb-pcl:documentation."
  t)

#+sbcl
(defun mopp:slot-definition-documentation (slot)
  (sb-pcl::documentation slot t))
```

CMUCL

```
(defmethod provide-mopp-symbol ((symbol symbol) (implementation (eql :cmu)))
  (when (find-symbol (string symbol) :pcl)
    (import-to-mopp (find-symbol (string symbol) :pcl))))

(defmethod provide-mopp-symbol ((symbol (eql 'mopp:slot-definition-documentation))
                                (implementation (eql :cmu)))
  "Provide SLOT-DEFINITION-DOCUMENTATION on CMUCL.

Like SBCL SLOT-DEFINITION-DOCUMENTATION on CMUCL is just a call
to documentation."
  t)

#+cmu
(defun mopp:slot-definition-documentation (slot)
  (documentation slot t))
```

Lispworks

```
(defmethod provide-mopp-symbol ((symbol symbol) (implementation (eql :lispworks)))
  (when (find-symbol (string symbol) :clos)
    (import-to-mopp (find-symbol (string symbol) :clos))))

(defmethod provide-mopp-symbol ((symbol (eql 'mopp:eql-specializer))
                                (implementation (eql :lispworks)))
  t)

(defmethod provide-mopp-symbol ((symbol (eql 'mopp:eql-specializer-object))
                                (implementation (eql :lispworks)))
  t)

(defmethod provide-mopp-symbol ((symbol (eql 'mopp:method-specializers))
                                (implementation (eql :lispworks)))
  "We can not simply export CLOS:METHOD-SPECIALIZERS as we have
to insert mopp:eql-specializers"
  t)

#+lispworks
(defclass mopp:eql-specializer ()
  ((object :accessor mopp::eql-specializer-object :initarg :object))
  (:documentation "Wrapper class representing eql-specializers.

Lispworks does not implement an eql-specializer class but simply
returns lists form method-specializers, this class (along with a
wrapper for clos:method-specializers) hide this detail."))

#+lispworks
(defun mopp:method-specializers (method)
  "More MOP-y implementation of clos:method-specializers.

For every returned value of clos:method-specializers of the
form `(eql ,OBJECT) this function returns a mopp:eql-specializer
object wrapping OBJECT."
  (mapcar (lambda (spec)
            (typecase spec
              (cons (make-instance 'mopp:eql-specializer :object (second spec)))
              (t spec)))
          (clos:method-specializers method)))
```

CLISP

```
(defmethod provide-mopp-symbol ((symbol symbol) (implementation (eql :clisp)))
  (when (find-symbol (string symbol) :clos)
    (import-to-mopp (find-symbol (string symbol) :clos))))
```

ALLEGRO

```
(defmethod provide-mopp-symbol ((symbol symbol) (implementation (eql :allegro)))
  (when (find-symbol (string symbol) :mop)
    (import-to-mopp (find-symbol (string symbol) :mop))))
```

```
(defmethod provide-mopp-symbol ((symbol (eql 'mopp:slot-definition-documentation))
(implementation (eql :allegro)))
  t)

#+allegro
(defun mopp:slot-definition-documentation (slot)
  (documentation slot t))
```

20.1 Building the MOPP package

we can't just do a do-external-symbols since we mess with the package and that would put us in implementation dependent territory, so we first build up a list of all the external symbols in mopp and then work on that list.

```
#+(or
   openmcl
   sbcl
   cmu
   lispworks
   clisp
   allegro)
(eval-when (:compile-toplevel :load-toplevel :execute)
  (push 'mopp::have-mop *features*))

#+mopp::have-mop
(let ((external-symbols '()))
  (do-external-symbols (sym (find-package :it.bese.arnesi.mopp))
    (push sym external-symbols))
  (dolist (sym external-symbols)
    (unless (provide-mopp-symbol sym #+openmcl :openmcl
                                     #+sbcl :sbcl
                                     #+cmu :cmu
                                     #+lispworks :lispworks
                                     #+clisp :clisp
     #+allegro :allegro)
      (warn "Unimplemented MOP symbol: ~S" sym))))

#-mopp::have-mop
(warn "No MOPP implementation available for this lisp implementation.")
```

21 Messing with numbers

```
(defun parse-ieee-double (u64)
  "Given an IEEE 64 bit double representeted as an integer (ie a
  sequence of 64 bytes), return the coressponding double value"
  (* (expt -1 (ldb (byte 1 63) u64))
     (expt 2 (- (ldb (byte 11 52) u64) 1023))
     (1+ (float (loop for i from 51 downto 0
                      for n = 2 then (* 2 n)
                      for frac = (* (/ n) (ldb (byte 1 i) u64))
                      sum frac)))))
```

```
(defun parse-float (string
                    &key (start 0) (end nil) (radix 10)
                         (type 'single-float ))
  (let* ((*read-eval* nil)
 (*read-base* radix)
 (value (read-from-string string nil nil
                                    :start start :end end)))
    (if (and value (numberp value))
(coerce value type)
nil)))

(define-modify-macro mulf (delta)
  *
  "SETF NUM to the result of (* NUM B).")

(define-modify-macro divf (delta)
  /
  "SETF NUM to the result of (/ NUM B).")

(define-modify-macro minf (other)
  (lambda (current other)
    (if (< other current)
        other
        current))
  "Sets the place to new-value if new-value is #'< the current value")

(define-modify-macro maxf (other)
  (lambda (current other)
    (if (> other current)
        other
        current))
  "Sets the place to new-value if new-value is #'> the current value")

(defun map-range (lambda min max &optional (step 1))
  (loop for i from min upto max by step
     collect (funcall lambda i)))

(defmacro do-range ((index &optional min max step return-value)
                    &body body)
  (assert (or min max)
          (min max)
          "Must specify at least MIN or MAX")
  `(loop
      for ,index ,@(when min `(from ,min))
                 ,@(when max `(upto ,max))
                 ,@(when step `(by ,step))
      do (progn ,@body)
      finally (return ,return-value)))

(defun 10^ (x)
  (expt 10 x))
```

22 Miscalaneous stuff

```
(defun intern-concat (string-designators &optional (package *package*))
  (intern (strcat* string-designators) package))

(defmacro with-unique-names ((&rest bindings) &body body)
  "Evaluate BODY with BINDINGS bound to fresh unique symbols.

Syntax: WITH-UNIQUE-NAMES ( { var | (var x) }* ) declaration* form*

Executes a series of forms with each VAR bound to a fresh,
uninterned symbol. The uninterned symbol is as if returned by a call
to GENSYM with the string denoted by X - or, if X is not supplied, the
string denoted by VAR - as argument.

The variable bindings created are lexical unless special declarations
are specified. The scopes of the name bindings and declarations do not
include the Xs.

The forms are evaluated in order, and the values of all but the last
are discarded \(that is, the body is an implicit PROGN)."
  ;; reference implementation posted to comp.lang.lisp as
  ;; <cy3bshuf30f.fsf@ljosa.com> by Vebjorn Ljosa - see also
  ;; <http://www.cliki.net/Common%20Lisp%20Utilities>
  `(let ,(mapcar (lambda (binding)
                   (check-type binding (or cons symbol))
                   (destructuring-bind (var &optional (prefix (symbol-name var)))
                       (if (consp binding) binding (list binding))
                     (check-type var symbol)
                     `(,var (gensym ,(concatenate 'string prefix "-")))))
                 bindings)
     ,@body))

(defmacro rebinding (bindings &body body)
  "Bind each var in BINDINGS to a gensym, bind the gensym to
var's value via a let, return BODY's value wrapped in this let.

Evaluates a series of forms in the lexical environment that is
formed by adding the binding of each VAR to a fresh, uninterned
symbol, and the binding of that fresh, uninterned symbol to VAR's
original value, i.e., its value in the current lexical
environment.

The uninterned symbol is created as if by a call to GENSYM with the
string denoted by PREFIX - or, if PREFIX is not supplied, the string
denoted by VAR - as argument.

The forms are evaluated in order, and the values of all but the last
are discarded \(that is, the body is an implicit PROGN)."
  ;; reference implementation posted to comp.lang.lisp as
  ;; <cy3wv0fya0p.fsf@ljosa.com> by Vebjorn Ljosa - see also
```

```
  ;; <http://www.cliki.net/Common%20Lisp%20Utilities>
  (loop for binding in bindings
        for var = (car (if (consp binding) binding (list binding)))
        for name = (gensym)
        collect `(,name ,var) into renames
        collect ``(,,var ,,name) into temps
        finally (return `(let* ,renames
                           (with-unique-names ,bindings
                             `(let (,,@temps)
                                ,,@body))))))

(defmacro with-accessors* (accessor-names object &body body)
  "Just like WITH-ACCESSORS, but if the slot-entry is a symbol
  assume the variable and accessor name are the same."
  `(with-accessors ,(mapcar (lambda (name)
      (if (consp name)
  name
  `(,name ,name)))
    accessor-names)
       ,object
     ,@body))

(defmacro define-constant (name value doc-string &optional export-p)
  "DEFCONSTANT with extra EXPORT-P argument."
  `(eval-when (:compile-toplevel :load-toplevel :execute)
     ,(when export-p
        `(export ',name ,(package-name (symbol-package name))))
     (defconstant ,name ,value ,doc-string)))
```

23 Manipulating sequences

```
(defun tail (seq &optional (how-many 1))
  "Returns the last HOW-MANY elements of the sequence SEQ. HOW-MANY is
  greater than (length SEQ) then all of SEQ is returned."
  (let ((seq-length (length seq)))
    (cond
      ((<= 0 how-many seq-length)
       (subseq seq (- seq-length how-many)))
      ((< seq-length how-many)
       (copy-seq seq))
      (t ; (< how-many 0)
       (head seq (- how-many))))))

(defun but-tail (seq &optional (how-many 1))
  "Returns SEQ with the last HOW-MANY elements removed."
  (let ((seq-length (length seq)))
    (cond
      ((<= 0 how-many seq-length)
       (subseq seq 0 (- seq-length how-many)))
      ((< seq-length how-many)
```

```
       (copy-seq seq))
      (t
       (but-head seq (- how-many))))))

(defun head (seq &optional (how-many 1))
  "Returns the first HOW-MANY elements of SEQ."
  (let ((seq-length (length seq)))
    (cond
      ((<= 0 how-many (length seq))
       (subseq seq 0 how-many))
      ((< seq-length how-many)
       (copy-seq seq))
      (t
       (tail seq (- how-many))))))

(defun but-head (seq &optional (how-many 1))
  "Returns SEQ with the first HOW-MANY elements removed."
  (let ((seq-length (length seq)))
    (cond ((<= 0 how-many (length seq))
           (subseq seq how-many))
          ((< seq-length how-many)
           (copy-seq seq))
          (t
           (but-tail seq (- how-many))))))

(defun starts-with (sequence prefix &key (test #'eql))
  "Test whether the first elements of SEQUENCE are the same (as
  per TEST) as the elements of PREFIX."
  (let ((length1 (length sequence))
        (length2 (length prefix)))
    (when (< length1 length2)
      (return-from starts-with nil))
    (dotimes (index length2 t)
      (when (not (funcall test (elt sequence index) (elt prefix index)))
        (return-from starts-with nil)))))

(defun ends-with (seq1 seq2 &key (test #'eql))
  "Test whether SEQ1 ends with SEQ2. In other words: return true if
  the last (length seq2) elements of seq1 are equal to seq2."
  (let ((length1 (length seq1))
        (length2 (length seq2)))
    (when (< length1 length2)
      ;; if seq1 is shorter than seq2 than seq1 can't end with seq2.
      (return-from ends-with nil))
    (loop
       for seq1-index from (- length1 length2) below length1
       for seq2-index from 0 below length2
       when (not (funcall test (elt seq1 seq1-index) (elt seq2 seq2-index)))
         do (return-from ends-with nil)
       finally (return t))))
```

```
(defun read-sequence* (sequence stream &key (start 0) end)
  "Like READ-SEQUENCE except the sequence is returned as well.

The second value returned is READ-SEQUENCE's primary value, the
primary value returned by READ-SEQUENCE* is the medified
sequence."
  (let ((pos (read-sequence sequence stream :start start :end end)))
    (values sequence pos)))

(defmacro deletef
    (item sequence &rest delete-args
     &environment e)
  "Delete ITEM from SEQUENCE, using cl:delete, and update SEQUENCE.

DELETE-ARGS are passed directly to cl:delete."
  (multiple-value-bind (vars vals store-vars writer-form reader-form)
      (get-setf-expansion sequence e)
    `(let* (,@(mapcar #'list vars vals)
            (,(car store-vars) ,reader-form))
       (setq ,(car store-vars) (delete ,item ,(car store-vars)
                                       ,@delete-args))
       ,writer-form)))
```

23.1 Levenshtein Distance

1) Set n to be the length of s. Set m to be the length of t. If n = 0, return m and exit. If m = 0, return n and exit. Construct a matrix containing 0..m rows and 0..n columns.

2) Initialize the first row to 0..n. Initialize the first column to 0..m.

3) Examine each character of s (i from 1 to n).

4) Examine each character of t (j from 1 to m).

5) If s[i] equals t[j], the cost is 0. If s[i] doesn't equal t[j], the cost is 1.

6) Set cell d[i,j] of the matrix equal to the minimum of: a. The cell immediately above plus 1: d[i-1,j] + 1. b. The cell immediately to the left plus 1: d[i,j-1] + 1. c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.

7) After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m].

```
(defun levenshtein-distance (source target &key (test #'eql))
  (block nil
    (let ((source-length (length source))
  (target-length (length target)))
      (when (zerop source-length)
(return target-length))
      (when (zerop target-length)
(return source-length))
      (let ((buffer (make-array (1+ target-length))))
(dotimes (i (1+ target-length))
  (setf (aref buffer i) i))
```

```
;; we make a slight modification to the alogrithm described
;; above. we don't create the entire array, just enough to
;; keep the info we need, which is an array of size
;; target-length + the "above" value and the "over". (this is
;; similar to the optimizaiont for determining lcs).
(loop
   for i from 1 upto source-length
   do (setf (aref buffer 0) i)
   do (loop
 with above-value = i
 with over-value = (1- i)
 for j from 1 upto target-length
 for cost = (if (funcall test (elt source (1- i))
      (elt target (1- j)))
0 1)
 do (let ((over-value* (aref buffer j)))
      (setf (aref buffer j) (min (1+ above-value)
 (1+ (aref buffer j))
 (+ cost over-value))
    above-value (aref buffer j)
    over-value over-value*))))
(return (aref buffer target-length))))))
```

24 A reader macro for simple lambdas

```
(defun enable-sharp-l ()
  (set-dispatch-macro-character #\# #\L #'sharpL-reader))

(defun sharpL-reader (stream subchar min-args)
  "Reader macro for simple lambdas."
  (declare (ignore subchar))
  (let* ((form (read stream t nil t))
         (lambda-args (loop
                        for i upfrom 1 upto (max (or min-args 0)
                                                 (highest-bang-var form))
                        collect (make-sharpl-arg i))))
    `(lambda ,lambda-args
       , (when lambda-args
           `(declare (ignorable ,@lambda-args)))
       ,form)))

(defun highest-bang-var (form)
  (acond
   ((consp form) (max (highest-bang-var (car form))
                      (highest-bang-var (cdr form))))
   ((bang-var-p form) it)
   (t 0)))

(defun bang-var-p (form)
  (and (symbolp form)
```

```
       (char= #\! (aref (symbol-name form) 0))
       (parse-integer (subseq (symbol-name form) 1) :junk-allowed t)))

(defun make-sharpl-arg (number)
  (intern (format nil "!~D" number)))
```

25 def-special-environment

```
(defun check-required (name vars required)
  (dolist (var required)
    (assert (member var vars)
            (var)
            "Unrecognized symbol ~S in ~S." var name)))

(defmacro def-special-enviroment (name (&key accessor binder binder*)
                                  &rest vars)
  "Define two macros for dealing with groups or related special variables.

ACCESSOR is defined as a macro: (defmacro ACCESSOR (VARS &rest
BODY)).  Each element of VARS will be bound to the
current (dynamic) value of the special variable.

BINDER is defined as a macro for introducing (and binding new)
special variables. It is basically a readable LET form with the
prorpe declarations appended to the body. The first argument to
BINDER must be a form suitable as the first argument to LET.

ACCESSOR defaults to a new symbol in the same package as NAME
which is the concatenation of \"WITH-\" NAME. BINDER is built as
\"BIND-\" and BINDER* is BINDER \"*\"."
  (unless accessor
    (setf accessor (intern-concat (list '#:with- name) (symbol-package name))))
  (unless binder
    (setf binder   (intern-concat (list '#:bind- name) (symbol-package name))))
  (unless binder*
    (setf binder*  (intern-concat (list binder '#:*) (symbol-package binder))))
  `(eval-when (:compile-toplevel :load-toplevel :execute)
     (flet ()
       (defmacro ,binder (requested-vars &body body)
         (check-required ',name ',vars (mapcar #'car requested-vars))
         `(let ,requested-vars
            (declare (special ,@(mapcar #'car requested-vars)))
            ,@body))
       (defmacro ,binder* (requested-vars &body body)
         (check-required ',name ',vars (mapcar #'car requested-vars))
         `(let* ,requested-vars
            (declare (special ,@(mapcar #'car requested-vars)))
            ,@body))
       (defmacro ,accessor (requested-vars &body body)
         (check-required ',name ',vars requested-vars)
```

```
        `(locally (declare (special ,@requested-vars))
           ,@body))
      ',name)))
```

26 Manipulating strings

```
(defvar +lower-case-ascii-alphabet+
  "abcdefghijklmnopqrstuvwxyz"
  "All the lower case letters in 7 bit ASCII.")

(defvar +upper-case-ascii-alphabet+
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  "All the upper case letters in 7 bit ASCII.")

(defvar +ascii-alphabet+
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
  "All letters in 7 bit ASCII.")

(defvar +alphanumeric-ascii-alphabet+
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
  "All the letters and numbers in 7 bit ASCII.")

(defvar +base64-alphabet+
  "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
  "All the characters allowed in base64 encoding.")

(defun random-string (&optional (length 32) (alphabet +ascii-alphabet+))
  "Returns a random alphabetic string.

The returned string will contain LENGTH characters chosen from
the vector ALPHABET.
"
  (loop with id = (make-string length)
        with alphabet-length = (length alphabet)
        for i below length
        do (setf (cl:aref id i)
                 (cl:aref alphabet (random alphabet-length)))
        finally (return id)))

(defun strcat (&rest items)
  "Returns a fresh string consisting of ITEMS concat'd together."
  (strcat* items))

(defun strcat* (string-designators)
  "Concatenate all the strings in STRING-DESIGNATORS."
  (with-output-to-string (strcat)
    (dotree (s string-designators)
      (princ s strcat))))

(defun fold-strings (list)
  "Traverse list and cancatenates any sequential elements which
```

```
  are lists. removes any \"\" elements for LIST. returns a fresh
  list."
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((return-list '())
        (string-buffer (make-array 20 :element-type 'character
                                      :fill-pointer 0
                                      :adjustable t)))
    (flet ((collect-string (string)
             (if (stringp (car return-list))
                 ;; collecting another string, just add it.
                 (loop
                    for char across string
                    do (vector-push-extend char (car return-list)))
                 ;; new string
                 (progn
                   (loop
                      initially (setf (fill-pointer string-buffer) 0)
                      for char across string
                      do (vector-push-extend char string-buffer))
                   (push string-buffer return-list))))
           (collect-object (object)
             (if (stringp (car return-list))
                 (setf (car return-list)
                       (map-into (make-string (length string-buffer))
                                 #'identity
                                 string-buffer)
                       return-list (cons object return-list))
                 (push object return-list))))
      (dolist (l list)
        (if (stringp l)
            (collect-string l)
            (collect-object l)))
      (nreverse return-list))))

(defun trim-string (string &optional (char '(#\Space #\Tab #\Newline
                                             #\Return #\Linefeed)))
  (let ((chars (ensure-list char)))
    (subseq string
    (loop for index upfrom 0 below (length string)
  when (not (member (aref string index) chars))
    do (return index)
  ;; if we get here we're trimming the entire string
                  finally (return-from trim-string ""))
    (loop for index downfrom (length string)
  when (not (member (aref string (1- index)) chars))
    do (return index)))))

(defvar ~%
  (format nil "~%")
  "A string containing a single newline")
```

```
(defvar ~T
  (string #\Tab)
  "A string containing a single tab character.")

(defvar +CR-LF+
  (make-array 2 :element-type 'character
                :initial-contents (list (code-char #x0D)
                                        (code-char #x0A)))
  "A string containing the two characters CR and LF.")

(defun ~D (number &optional stream &key mincol pad-char)
  (format stream "~v,vD" mincol pad-char number))

(defun ~A (object &optional stream)
  (format stream "~A" object))

(defun ~S (object &optional stream)
  (format stream "~S" object))

(defun ~W (object &optional stream)
  (format stream "~W" object))
```

27 vector/array utilities

```
(defun vector-push-extend* (vector &rest items)
  (let ((element-type (array-element-type vector)))
    (dolist (item items)
      (cond
        ((typep item element-type) ;; item can be put directly into the
         (vector-push-extend item vector))
        ((typep item `(vector ,element-type)) ;; item should be a vector
         (loop
            for i across item
            do (vector-push-extend i vector)))
        (t
         (error "Bad type for item ~S." item))))
    vector))

(defun string-from-array (array &key (start 0) (end (1- (length array))))
  "Assuming ARRAY is an array of ASCII chars encoded as bytes return
the corresponding string. Respect the C convention of null terminating
strings. START and END specify the zero indexed offsets of a sub range
of ARRAY."
  ;; This is almost always the case
  (assert (<= 0 start (1- (length array)))
          (start)
          "START must be a valid offset of ARRAY.")
  (assert (<= 0 end (1- (length array)))
          (end)
          "END must be a valid offset of ARRAY.")
```

```
(assert (<= start end)
        (start end)
        "START must be less than or equal to END.")
(assert (every (lambda (element) (<= 0 element 255)) array)
(array)
"Some element of ~S was not > 0 and < 255" array)
(let* ((working-array (make-array (1+ (- end start))
                                  :element-type (array-element-type array)
                                  :displaced-to array
                                  :displaced-index-offset start))
(length (if-bind pos (position 0 working-array)
    pos
    (length working-array))))
  (map-into (make-array length :element-type 'character)
    #'code-char
    working-array)))
```