"COMBINATORIAL HYPERCODING

(of IMAGE PROCESSING OPERATION LIBRARIES)

with MACRO-DEFINING MACROS"

Robert C. Vogt, Ph.D.
Robert S. Vogt & Partners
(& NovoDynamics, Inc.)

Introduction

- Programmer -Empowerment- via Lisp Macros
- Image Processing Algorithm Problem Domain

Lisp-side Motivations:

- Long fascination with Lisp's unique merging of code/data representations, and the power that provides in automatically writing code
- Ph.D. work used Lisp for automatic discovery of simple image algebraic recognition algorithms, from truthed examples (simple code generation)
- Wanted to see how to use this power in a more significant way--not just to write small utility macros, or embedded languages, but substantial whole programs and libraries of programs
- Wanted to write such programs automatically, using nothing more complex than basic list-handling tools
- Wanted to find out what degree of leverage is possible? What amplification of programmer power can Lisp provide?

Image Processing-side Motivations:

- Career has largely been in creating image-based recognition algorithms
  [CAT, MRI, DCA, TEM, SEM, Egels, MS, SAR, IFSARE, RCS, USPS, Arabic OCR]

- IMAGE EXAMPLES -

Image Processing-side Motivations, cont.:

- All of these problems are basically searches, for an algorithm:
  - Over a very large space of possible operators and parameter choices
  - For sequences of many steps
  - To be optimized over 100's of training/test images
  - With little more to go on than knowledge of:
    - Problem domain/scope (often limited)
    - Image acquisition method/sensor
    - Expert knowlege of what each image operator's effect should be in a given situation

(Very difficult, intractable problems. Fortunately, perfection is not always required.)

- Long-time personal desire to create a (more efficient, intelligent) image processing algorithm 'discovery' system and expert developer's aid, entirely in Lisp, from top to bottom:
  - Macros to optimize low-level operators [this paper]
  - Generic functions to collect common operators over various data types
  - A Lisp-based algebraic language to write image algorithms, compactly
  - Higher level 'semantic' or human-meaningful operations to express concepts
  - Intelligent meta-processes for doing large-scale algorithm discovery and testing

- Daunting task due to so many operators and types of objects to consider (and no help)

Combinatorial 'Hypercoding':

"Automated approach to creating large numbers of functions or macros that share many common characteristics, but which differ in somewhat independent particulars related to data types, exact parameters or internal functions used, object geometries, optimization levels, and others."

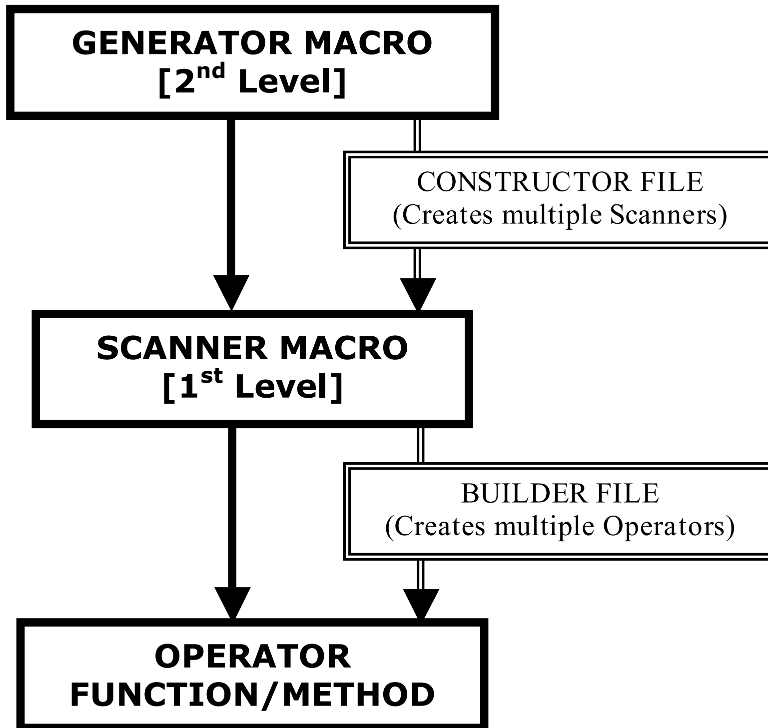
- Shown here via complex macro-generating macros
- These generate critical macro tools for creating a large library of optimized image processing operations/methods--clearly, quickly, and easily

Advantages, Goals include:

- Completely automated, nearly instantaneous generation
- Compact representation of operational code characterisics
- Easier control, maintenance, and modification of macro libraries
- Increased clarity and conciseness in defining functions/methods
- Functions easier to read, verify, and maintain
- Programmer “Empowerment”

Diagram of Generators, Scanners, Functions/Methods, Constructors

Terminology



Background on Image Processing Domain

Decomposition of (Combinatorial) Image Processing Operation Space:

[Generator Macro]

- **Data item geometry** (1D, 2D, CCs, lists, graphs, etc.)
  (What kind of image/data object do you have?)

- **Scan type or class** (raster, sub-windows, neighborhood, Qs)
  (What to you want to do with it?)

[Scanner Macro]

- **Scan direction** or other scanning specifics
  (Which scanning version/order? fwd, rvs)

- **Image argument configuration** (mapping 0/1,1/0,1/1..)
  (How many image inputs and outputs? Operator categories)

- **Data element types** (binary, u/byte, u/short, fix, float, RGB, Mag/Phs)
  (What are the component element types (and interpretations)?)

- **Optimization settings** (speed/safety 3/1, 3/2, 1/1, floating)
  (How do you want the inner loop to be optimized?)

[Operator Function/Method]

- **Data element operation** or expression
  (What is being done to each element?)

- EXAMPLE 'COMBINATORIAL HYPERCODING' ILLUSTRATION -

Generating 24 Files (using 6 Generators):

- 3 geometries (2D-array, 2D-row, 1D-vector)
- 2 scan types (raster element, location)
- 4 image data types (binary, ubyte, ushort, posfix)

Each file containing multiple 'Scanner' macros:

- argument structures (8: 0/1, 0/2, 1/0, 2/0, 1/1, 1/2, 2/1, 1/3)
- scan options (fwd, rvs, column-wise)
- optimizations (speed/safety: 1/1, 3/2, 3/1 or nil-floating)

Also creating one 'instance' of the function code produced by calling (expanding) each Scanner macro (for verification)

- All 6 Generators work correctly
- Generate Scanner code which compiles and creates Function code
- Function code also compiles and is correct
- Function code has been tested at the level of simple image 'point' operations (copy, complement, mod rotate, threshold, constant, zero, draw ramp, random image, or pattern, mask image, compute a count or histogram, etc.)

...> to ACL

from ACL...>

"There is only one other place in life, where I feel this kind of 'exhiliration'."
(Kevin Kline, *French Kiss*)

- HYPERCODING DEMONSTRATION RESULTS -

~4-5 seconds CPU, 5-6 sec clock time
1.67 GHz G4, Power PC processor
1.5 GB SDRAM

Using ACL 7.0 for Windows running under Virtual PC 7.0
Writing all code (1.2MB + 0.5MB) to disk
Printing log messages to screen

1.2 MBytes of Scanner Macro code generated:

~400 Scanner macros
35,000 lines
31,000 non-blank
31,000 non-comment source (a good 'ream' of paper ~500+ pages)
40,000 conses (open parens)

0.5 MBytes of Function Instance code generated (for verification):

13,000 lines
11,000 non-blank
11,000 non-comment source (~200 pages)
22,000 conses

… [finally getting somewhere]

Presentation Overview/Organization:

- Introduction and Motivations
- Image Processing Domain Background & Decomposition

- Example of Combinatorial Hypercoding using a Constructor file to auto-generate many Scanner macros

- My 'Home-Grown' Development Process
  (implementation of IP operator space decomposition):
  - Basic 2D byte-array image Copy operation
  - 1st-level Scanner macro to produce Copy operation
  - Alternate form of 1st-level Scanner macro*
  - 2nd-level Generator macro form, for creating similar Scanner macros
  - Scanner construction and Function creation process
  - [Illustrations of specific subform constructions within Generator code]

- Conclusions

*Issue of representing expanded Backquoted forms within Lisp, in a standard way

Macro Illustrations and Code Fragments

Note the following convention for most examples that follow:

Function code formal arguments: outimage, inimage
Macro code formal arguments: outimg, inimg

#1 - Basic 2D Byte Array Image Copy Operation

Compact, straight, unoptimized form

```
;; Point Transform from one Byte Image to another, same sizes,
;; 1-pixel border on all sides. Images are represented as vectors
;; of *typed* row vectors.


(defun Array-Point-Op (outimage inimage [&optional (val 0)])

  (let ((nrows (Image-Rows inimage))
        (ncols (Image-Cols inimage)) )

    (do ((r 1 (1+ r)))
        ((> r nrows))

      (let ((row-y (aref outimage r))
            (row-x (aref inimage  r)) )

        (do ((c 1 (1+ c)))
            ((> c ncols))

          ;; Copy operation
          (setf (aref row-y c) (aref row-x c))
          )))

    outimage)
  )
```

Other Image Element Function/Operation Choices (some including an additional 'val' argument):

```
;; Threshold (Up)

(setf (aref row-y c)
  (if (>= (aref row-x c) val) 255 0))


;; Invert/Complement

(setf (aref row-y c)
  (- 255 (aref row-x c)))


;; Mod Add Ramp

(setf (aref row-y c)
  (mod (+ (aref row-x c) r c) 256))


;; Mod Rotate

(setf (aref row-y c)
  (mod (+ (aref row-x c) val) 256))


;; Indirect Calc'd LUT Reference
;; (Loops and element expression wrapped within 'let)

(let ((LUT (make-array 256 :element-type
                       '(unsigned-byte 8)
                       :initial-element 0)) )
  (dotimes (i 256)
    (setf (aref LUT i) (mod (+ i val) 256)))
  ...
  ...
  (setf (aref row-y c) (aref LUT (aref row-x c)))
  )
```

And many others:

Threshold Down (<=)
Quantize (into K levels)
Remap (Brightness/Contrast, or Linear stretch)
Histogram Equalize
Cover (value or range with another)
Replace (if NOT 255/0, do calc, else skip)
Spatial Mask (ranges or bit mask)
etc.

Same code in <u>Optimized</u> form
(declares, the's, speed/safety inner loop)

Key added elements are:
- Declarations on argument types
- Declarations on let and loop variables
- Optimization declaration for inner loop
- Use of 'the constructs for expression types
  (not required; for illustration; special challenges)

```
;; "(Over)-Optimized" Form of Array Copy Operation:


(defun Array-Point-Op (outimage inimage [&optional (val 0)])

  "Array-Point-Op Docstring"

  (declare (type byte-image-array outimage inimage)
           ;; (type (unsigned-byte 8) val)
   )

  (let ((nrows (Image-Rows inimage))
        (ncols (Image-Cols inimage)) )
    (declare (type posfix nrows ncols))

    (do ((r 1 (1+ r)))
        ((> r nrows))
      (declare (type posfix r))

      (let ((row-y (the byte-image-vector
                        (aref (the byte-image-array outimage r))))
           ;;      (aref outimage r)    ;; [Alternative]

           (row-x (the byte-image-vector
                        (aref (the byte-image-array inimage) r))))
           ;;      (aref inimage r)     ;; [Alternative]
           )
        (declare (type byte-image-vector row-y row-x))

        (do ((c 1 (1+ c)))
            ((> c ncols))
          (declare (type posfix c))

          (declare (optimize (speed 3) (safety 1)))

          ;; Copy
          (setf (the (unsigned-byte 8) (aref row-y c))  ;;*
            (the (unsigned-byte 8) (aref row-x c)))
          )))

   outimage)
  )
```

*Use of 'the' construct on 'place' arg as well as value is redundant but not illegal.

#2 – 1st-Level Scanner Macro to produce Copy Operation

Backquote form, with Gensyms and Symbol Macros (simpler Exprs)

```
(defmacro 2D-Byte11-Array-Elt-Op-31 (optkey outimg inimg expr)

"Macro to apply an EXPR to each Point of a 2D-byte array,
INIMG, returning the result in a second array, OUTIMG. It is
also possible for INIMG and OUTIMG to be the same array. A
number of symbol macros are provided to make it easier to
express EXPR. OPTKEY specifies inner loop optimization.
Code returns OUTIMG."

  (With-Gensyms
      (goutimg ginimg gnrows gncols gr gc grow-y grow-x)

    `(let* ((,goutimg ,outimg)
            (,ginimg  ,inimg)
            (,gnrows (Image-Rows ,ginimg))
            (,gncols (Image-Cols ,ginimg)) )

       (declare (type posfix ,gnrows ,gncols)
                (type byte-image-array ,goutimg ,ginimg))
       (declare (ignorable))

       (do ((,gr 1 (1+ ,gr)))
           ((> ,gr ,gnrows))
         (declare (type posfix ,gr))

         (let ((,grow-y
                (the byte-image-vector
                  (aref (the byte-image-array ,goutimg)
                        ,gr)))
               (,grow-x
                (the byte-image-vector
                  (aref (the byte-image-array ,ginimg)
                        ,gr))) )

           (declare (type byte-image-vector ,grow-y ,grow-x))
           (declare (ignorable ,grow-y ,grow-x))

           (do ((,gc 1 (1+ ,gc)))
               ((> ,gc ,gncols))
             (declare (type posfix ,gc))

             (declare
              (optimize
               ,@(if `,optkey (Select-Optimization `,optkey)   ;;*
                   '((speed 3) (safety 1)) )))
```

```
          (symbol-macrolet
              ((nrows ,gnrows)
               (ncols ,gncols)

               (r ,gr)
               (c ,gc)

               (y (the u8
                    (aref (the byte-image-vector ,grow-y)
                          ,gc)))
               (x (the u8
                    (aref (the byte-image-vector ,grow-x)
                          ,gc))) )

            ,expr)
          )))

      ,goutimg)
   ))
```

Symbol macros provide two important things:

a) A 'tiny' language to make it easier to write simple, clear expressions of what operator really does at each element visited:

{nrows, ncols, r, c, y, x}   => (setf y x) == Copy x to y

b) A solution for the problem of needing to substitute Backquote-Comma code into data element expressions (expr):

```
         (do ((,gc 1 (1+ ,gc)))
              ((> ,gc ,gncols))
            (declare (type posfix ,gc))

            (declare
             (optimize
              ,@(if `,optkey (Select-Optimization `,optkey)
                   '((speed 3) (safety 1)) )))

            ,expr

            ;; `(setf (aref ,grow-y ,gc) (aref ,grow-x ,gc))        ;;*
            ;;  => (setf (aref #:gnny #:gnnc) (aref #g:nnx #:gnnc))
            )))

      ,goutimg)
   ))
```

;;*Given this macro form, it is hard to insert Exprs w/o text manipulations, or
;;   the use of symbol macros

Macro expansion result (using Gensyms & Symbol Macros):
[Equivalent to previous optimized code – see spaced out version below]

```
cg-user(3): (PO (macroexpand-1

                 '(2D-Byte11-Array-Elt-Op-31
                   :opt11
                   outimage
                   inimage
                   (setf y x))
                 ))

=> [direct result:]

(let* ((#:g347 outimage)
       (#:g348 inimage)
       (#:g349 (Image-Rows #:g348))
       (#:g350 (Image-Cols #:g348)))
  (declare (type posfix #:g349 #:g350)
           (type byte-image-array #:g347 #:g348))
  (declare (ignorable))
  (do ((#:g351 1 (1+ #:g351)))
      ((> #:351 #:g349))
    (declare (type posfix #:g351))
    (let ((#:g353
           (the
            byte-image-vector
            (aref (the byte-image-array #:g347) #:g351)))
          (#:g354
           (the
            byte-image-vector
            (aref (the byte-image-array #:g348) #:g351))))
      (declare (type byte-image-vector #:g353 #:g354))
      (declare (ignorable #:g353 #:g354))
      (do ((#:g352 1 (1+ #:g352)))
          ((> #:g352 #:g350))
        (declare (type posfix #:g352))
        (declare (optimize (speed 1) (safety 1)))
        (symbol-macrolet ((nrows #:g349)
                          (ncols #:g350)
                          (r #:g351)
                          (c #:g352)
                          (y
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g353)
                             #:g352)))
                          (x
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g354)
                             #:g352))))
          (setf y x)))))
  #:g347)
```

Spaced out version:

```
(let* ((#:g347 outimage)
       (#:g348 inimage)
       (#:g349 (Image-Rows #:g348))
       (#:g350 (Image-Cols #:g348)))

  (declare (type posfix #:g349 #:g350)
           (type byte-image-array #:g347 #:g348))
  (declare (ignorable))

  (do ((#:g351 1 (1+ #:g351)))
      ((> #:351 #:g349))
    (declare (type posfix #:g351))

    (let ((#:g353
           (the
            byte-image-vector
            (aref (the byte-image-array #:g347) #:g351)))
          (#:g354
           (the
            byte-image-vector
            (aref (the byte-image-array #:g348) #:g351))))

      (declare (type byte-image-vector #:g353 #:g354))
      (declare (ignorable #:g353 #:g354))

      (do ((#:g352 1 (1+ #:g352)))
          ((> #:g352 #:g350))
        (declare (type posfix #:g352))

        (declare (optimize (speed 1) (safety 1)))

        (symbol-macrolet ((nrows #:g349)
                          (ncols #:g350)
                          (r #:g351)
                          (c #:g352)

                          (y
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g353)
                             #:g352)))
                          (x
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g354)
                             #:g352))))

          (setf y x)))))
  #:g347)
```

Example uses of Generated Code for simple Function Defs

```
;; Corresponding previous Function Defs using
;; Scanner macro, with built-in symbol-macros:

(defun Copy (outim inim)

  (2D-Byte11-Array-Elt-Op-31
   :opt11 outim inim

   (setf y x)
   ))


(defun Threshold (outim inim val)

  (2D-Byte11-Array-Elt-Op-31
   :opt11 outim inim

   (setf y (if (> x val) 255 0))
   ))


(defun Mod-Add-Ramp (outim inim)

  (2D-Byte11-Array-Elt-Op-31
   :opt11 outim inim

   (setf y (mod (+ x r c) 256))
   ))


(defun Mod-Rotate (outim inim val)

  (2D-Byte11-Array-Elt-Op-31
   :opt11 outim inim

   (setf y (mod (+ x val) 256))
   ))


(defun Rotate-by-LUT (outim inim val)

  (let ((LUT (make-array 256 :element-type
                         '(unsigned-byte 8)
                         :initial-element 0)))
    (dotimes (i 256)
      (setf (aref LUT i) (mod (+ i val) 256)))

    (2D-Byte11-Array-Elt-Op-31
     :opt11 outim inim

     (setf y (aref LUT x))
     )))
```

-> These functions are easier to read, verify, maintain, re-use and automatically write.

Building 2nd-Level Macro-Generating Macros

Summary so far:

- Specific 2D image Copy operation
- 1st-Level Scanner macro to create similar ops, in Backquote-Comma style (2D byte-array forward raster-scan operation, with 1 input and 1 output image, and inner loop optimization: (speed 1) (safety 1)
- Use of symbol macros to more concisely inject clear, simple element expressions into the Scanner macro

Next questions:

- How could I define additional Scanner macros of the same type (i.e., 2D array raster-scanners), which differ in other attributes: scan direction, image data type, number of input and output images, and the level of inner loop optimization used? (But—which need to be called with an expr, to create an image operation.)

- Giving a larger repertoire of Scanner macro tools to use in defining image functions or methods within our system library, thereby extending our range to:
  - More image types
  - New classes of raster scan operations
  - Alternative numbers of arguments (operator mapping types)
  - Different versions of an operation (scan direction, loop optimization)

- The idea was that the Scanner library and Operations library would be fixed, sitting on disk–the Scanner macros were not going to be called at the normal expansion time, to insert code to be compiled–but rather were a set of tools for more easily creating a large system library of image operation functions (i.e., more of an off-line process.)

- *How can create (yet debug) such a large array of Scanner macros to use, automatically, and yet in a fairly straightforward manner?*

- First attempt at this was text substitution-based:
  - Used special symbol prefixes
  - Several steps required to get the gensyms and everything else just right
  - Target was a Backquote-Comma form (represented as text), then saved to a file in a manner that the Lisp reader would take in as a macro definition
  - While process worked to some extent, it was time consuming, cumbersome, and ultimately too ugly and inelegant to serve as a long-term solution.

Building 2nd-Level Macro-Generating Macros (cont.)

My basic problem came down to the fact that 'backquote' and 'comma' are in a sense, *outside* of the standard Lisp syntax, and thus cannot be used to generate expressions containing them via standard list manipulation functions.

I thus *could not use standard Lisp list manipulation tools to automatically write a large library of Scanner macros in Backquote-Comma form–by definition*. What I needed to have was something like the special operator 'quote', and its forward quote punctuation character:

```
'x  => (quote x)
`x  => (backquote x)
,x  => (comma x)
```

Then I could write:

```
(backquote (setf (comma gy) 0))  instead of
`(setf ,gy 0)  to get out
 (setf <gy> 0) == (setf #:gnnn 0),
```

in the generated function code, where <gy> is the gensym value of gy.

This would have made it very easy to write my macro-generating macro using only list manipulation tools. To my knowledge though, no such special operators exist. Without them, I was forced to look for a different representation.

Eventually it hit me–"Why worry?"–*Lisp was built from the beginning to make it easy to manipulate code as data via lists*, I didn't strictly need backquotes and commas, so the simplest solution to this problem was to simply go back to the more standard, 'straight list' representation methods, for example:

```
    (list 'setf gy 0) instead of `(setf ,gy 0)
```

for the case just mentioned. As another example, the following two forms are equivalent:

```
`(declare (type posfix ,gr))              ;; Backquote-Comma form, versus

 (list 'declare (list 'type 'posfix gr))  ;; Normal evaluation, no backquotes
```

(The 2nd form is easy to write automatically, using only standard list manipulation tools; the first cannot be, to the best of my knowledge.)

#3 - Alternate form of 1st-level Scanner Macro

Straight, Lisp 'List' form of Macro (Compare to previous B-C form, p. 13)

```
(defmacro 2D-Byte11-Array-Elt-Op-31 (optkey outimg inimg expr)

"Macro to apply an EXPR to each Point of a 2D-byte array,
INIMG, returning the result in a second array, OUTIMG. It is
also possible for INIMG and OUTIMG to be the same array. A
number of symbol macros are provided to make it easier to
express EXPR. OPTKEY specifies inner loop optimization.
Code returns OUTIMG."

  (With-Gensyms
      (goutimg ginimg gnrows gncols gr gc grow-y grow-x)

    (list
     'let*
     (list
      (list goutimg outimg)
      (list ginimg inimg)
      (list gnrows (list 'Image-Rows ginimg))
      (list gncols (list 'Image-Cols ginimg)))

     (cons
      'declare
      (list
       (list 'type 'posfix gnrows gncols)
       (list 'type 'byte-image-array goutimg ginimg)))

     (list 'declare (cons 'ignorable (list)))

     (list
      'do
      (list (list gr 1 (list '1+ gr)))
      (list (list '> gr gnrows))
      (list 'declare (list 'type 'posfix gr))

      (list
       'let
       (list
        (list
         grow-y
         (list 'the 'byte-image-vector
               (list 'aref
                     (list 'the 'byte-image-array goutimg)
                     gr)))

        (list
         grow-x
         (list 'the 'byte-image-vector
               (list
                'aref
                (list 'the 'byte-image-array ginimg)
                gr))))
```

```
      (cons
       'declare
       (list
        (list 'type 'byte-image-vector grow-y grow-x)))

      (list
       'declare
       (cons 'ignorable (list grow-y grow-x)))

      (list
       'do
       (list (list gc 1 (list '1+ gc)))
       (list (list '> gc gncols))
       (list 'declare (list 'type 'posfix gc))

       (list
        'declare
        (cons
         'optimize
         (if optkey (Select-Optimization optkey)
           '((speed 3) (safety 1)))))

       (list
        'symbol-macrolet
        (list
         (list 'nrows gnrows)
         (list 'ncols gncols)
         (list 'r gr)
         (list 'c gc)

         (list
          'y
          (list 'the 'u8
                (list
                 'aref
                 (list 'the 'byte-image-vector grow-y)
                 gc)))

         (list
          'x
          (list 'the 'u8
                (list
                 'aref
                 (list 'the 'byte-image-vector grow-x)
                 gc))))

        expr)
       )))

    goutimg)
   ))
```

It is much easier to write a macro to automatically generate this form of the Scanner macro (using standard Lisp tools for list manipulation), than it is to construct the previously illustrated Backquote-Comma form.

Expansion Call on List form of Macro
[Identical to B-C macro expansion except for exact gensym variable numbers]

```
;; Generated Functional Code from "List Form":

cg-user(191): (PO (macroexpand-1

                    '(2D-Byte11-Array-Elt-Op-31
                      :opt11
                      outimage
                      inimage
                      (setf y x))
                    ))
=>

(let* ((#:g2009 outimage)
       (#:g2010 inimage)
       (#:g2011 (Image-Rows #:g2010))
       (#:g2012 (Image-Cols #:g2010)))
  (declare (type posfix #:g2011 #:g2012)
           (type byte-image-array #:g2009 #:g2010))
  (declare (ignorable))
  (do ((#:g2013 1 (1+ #:g2013)))
      ((> #:g2013 #:g2011))
    (declare (type posfix #:g2013))
    (let ((#:g2015
           (the
            byte-image-vector
             (aref (the byte-image-array #:g2009) #:g2013)))
          (#:g2016
           (the
            byte-image-vector
             (aref (the byte-image-array #:g2010) #:g2013))))
      (declare (type byte-image-vector #:g2015 #:g2016))
      (declare (ignorable #:g2015 #:g2016))
      (do ((#:g2014 1 (1+ #:g2014)))
          ((> #:g2014 #:g2012))
        (declare (type posfix #:g2014))
        (declare (optimize (speed 1) (safety 1)))
        (symbol-macrolet ((nrows #:g2011)
                          (ncols #:g2012)
                          (r #:g2013)
                          (c #:g2014)
                          (y
                           (the u8
                            (aref
                             (the byte-image-vector #:g2015)
                             #:g2014)))
                          (x
                           (the u8
                            (aref
                             (the byte-image-vector #:g2016)
                             #:g2014))))
          (setf y x))
        )))
  #:g2009)
```

Simple Macro-Gen Macro Example and Backquote Rep. Issue

Paul Graham, On Lisp, p.216, accessor-generating macro
[double backquote-comma macro-generating macro]

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',',propname)))


;; Which (in ACL) expands to:

(macroexpand-1 '(propmacro thisprop))
=>
(defmacro thisprop (obj)
  (excl::bq-list `get obj
                 (excl::bq-list `quote 'thisprop)))

;; [Unexplained, cryptic, long, something new & 'outside' of Lisp]


(propmacro thisprop) => thisprop

(symbol-function 'thisprop) =>
#<macro thisprop @ #x20f04492>

;; [And basically 'invisible' in terms of normal Lisp, thus hard
;;  to debug, due precisely to this hidden level of indirection.]

;; How was I going to (correctly) write a 2nd-level macro,
;; 100s of lines long, using double backquotes and commas,
;; without being able to see/understand the generated macro code?
;; I needed something I could READ, CALL/EXP w/ exprs, and DEBUG!]


;; Now the above, cryptic form is equivalent to:

(defmacro thisprop (obj)    ;; preferred 'direct' form (LCL)
  `(get ,obj ',''thisprop))

;; OR [with backquote and comma list forms in Lisp]:

(defmacro thisprop (obj)
  (backquote
   (get (comma obj)
        (quote (comma (quote thisprop))) )))

;; OR [in straight list form]:

(defmacro thisprop (obj)
  (list 'get obj ''thisprop))

[= (defmacro thisprop (obj)
     (list (quote get) obj (quote (quote thisprop))) ]


;; Which (all) then expand to:

(macroexpand-1 '(thisprop glob)) =>
(get glob 'thisprop)

(macroexpand-1 '(thisprop 'sym)) =>
(get 'sym 'thisprop)
```

```
;; My List-based 2nd-level Macro

(defmacro propmacro2 (propname)
  (list 'defmacro propname (list 'obj)
        (list 'list ''get 'obj
              (list 'quote (list 'quote propname)) )))

;; Which expands to (readable, standard Lisp):

(defmacro thatprop (obj)
  (list 'get obj (quote (quote thatprop))))   ;; or in reverse

;; OR equivalently:

(defmacro thatprop (obj)
  (list 'get obj ''thatprop))


(macroexpand-1 '(propmacro2 thatprop)) =>
(defmacro thatprop (obj) (list 'get obj ''thatprop))

(propmacro2 thatprop) =>
thatprop

(macroexpand-1 '(thatprop glob)) =>
(get glob 'thatprop)

(macroexpand-1 '(thatprop 'sym)) =>
(get 'sym 'thatprop)



;; ALTERNATE with (backquote  ) and (comma  ) operators available:

(defmacro propmacro3 (propname)

  `(defmacro ,propname (obj)
     (backquote
      (get (comma obj)
           (quote (comma
                   (quote ,propname))) ))))

;; Which clearly separates 1st and 2nd level macro constructs

;; and would (readably) expand to:

(defmacro thisprop (obj)
  (backquote
   (get (comma obj)
        (quote (comma (quote thisprop))) )))

;; or even, with not much more work:

(defmacro thisprop
   `(get ,obj ','thisprop))   ;; i.e., the preferred form

;; instead of:

(defmacro thisprop (obj)
  (excl::bq-list `get obj
                 (excl::bq-list `quote 'thisprop)))
```

[Questions: List B/C expansion standard, why B/C--the list alternative for muggles]

#4 - 2nd-level Generator Macro to Construct other Scanner Macros

Building Generator Macro Code is a Three-Stage Process

1) Main *template* or skeleton structure of the output macro:
   a) Consisting of a number of variable substitution *components*
   b) Each inserts its required Scanner code when evaluated,
      (which happens when the Generator evaluates the template list).
   c) *Template and components implement the data item geometry and scan class dimensions for which the Generator is responsible*

2) Components of template each have their own unique form or structure:
   a) They may be built from multiple, varying types of *sub-components*.
      (For example, different kinds of components in a 'let' list.)*

3) The sub-components are themselves built from *sub-elements*
   a) Presence/absence determined by different *informational elements*
   b) Informational elements are combinatorial input variables or :keys to the Generator macro that determine which macro form one is trying to construct (i.e., the scanform, imgargs, elttype, and optlevel keys).
   c) *Sub-components and elements thus implement these other four dimensions of the image operation space (Scanner responsibilities)*

*Example: A 'let' list component has argitems, sizeitems, and valueitems as subcomponents, each of which are lists of zero or more elements, that may be present or absent for different reasons, as determined by scanner macro code features (arguments, and image data types).

Thus, in reverse (bottom-up):   [examples later]

1) The informational or combinatorial choice elements, are used to determine the sub-elements for any sub-component
2) Multiple sub-components of different types or categories may then go into the building of a component form (in the more complex cases)
3) The component constructor is the 'value' of a variable that is substituted into the overall template list, making a specific piece of the Scanner macro
4) The substituted form of the template is returned as the Scanner macro code
5) When the Scanner macro in turn is evaluated, it constructs the desired function code, along with the embedded element expression (expr).

Text Form NOTATION for the following Sections:

(B) == Backquote-Comma form
(E) == 1st Eval (Scanner macro), similar to B
(T) == 2nd Eval (Function code or Target code)

--typename-- == symbol or item designator
<expr> == value of expression or variable
{items} == non-delimited set of items eg,: item1 item2 item3
#:gname == gensym var bound to symbol 'gname

Generator Program Abstract Structure:
([JanArrayEltGen.cl] - Around **600+** code lines, previously larger)

```
Generator head/argslist, docstring
Generator argument checks

Variable (let) list
   Intermediate (list sub-component) items
   Macro code Template (component) items


;;  Begin Generation code for <each> Template item, in order:
;;  - Sub-components (args, sizes, loops, values, vects)
;;  - Merging of sub-components into Template component item

Macroargs list
Gensym list

<<Main body items - lets, loops, declares, ignores >>

Optimization declares
Symbol macros
Returns

;; End Template item Generation code


;;  - Main Template Code Generation Structure -

Head
  With-Gensyms
    lets
      loops
        declares, ignores
        optimizations
          symbol macros
            key element Expr    ;; [Argument to Scanner macro]
      returns
```

Backquote Form of Generator Main Template: (B)

```
`(defmacro
   ,macroname
   ,macroargs

   ,macrodoc

   (With-Gensyms
     ,gensymlist

     (list 'let*
       ,arrletlist
       ,arrdeclist
       ,arrignrlist

       (list 'do
         ,loop1start
         ,loop1end
         ,loop1dec

         (list 'let*
           ,vecletlist
           ,vecdeclist
           ,vecignrlist

           (list 'do
             ,loop2start
             ,loop2end
             ,loop2dec

             ,optdecs

             (list 'symbol-macrolet
               ,symbolist

               expr)   ;; Element Op
         )))

       ,returns)       ;; End Scanner template
   ))                  ;; End Generator defmacro
```

Target Code Form: (T)

```
(let* ({arrlets})
  (declare ({arrdecs}))
  (declare (ignorable {arrignrs}))
  (do (--start1--) (--end1--)
    (--dec1--)

    (let* ({veclets})
      (declare ({vecdecs}))
      (declare (ignorable {vecignrs}))
      (do (--start2--) (--end2--)
        (--dec2--)

        (declare (optimize (speed --sp--) (safety --sf--)))
        (symbol-macrolet
            ({symbol-binds})
          <expr>)
        )))
  --returns--)
```

List Form of Main Generator Template

```
;; *** MAIN Code Generation Section ***

(list
 'defmacro
 macroname
 macroargs

 macrodoc

 (list
  'With-Gensyms
  gensymlist

  (list
   'list ''let*
   arrletlist
   arrdeclist
   arrignrlist

   (list
    'list ''do
    loop1start
    loop1end
    loop1dec

    (list
     'list ''let*
     vecletlist
     vecdeclist
     vecignrlist

     (list
      'list ''do
      loop2start
      loop2end
      loop2dec

      optdecs

      (list
       'list ''symbol-macrolet
       symbolist

       'expr)  ;; Element op
    )))

   returns)    ;; End Scanner template
  ))           ;; End Generator defmacro
```

Quoteds are ‘hard’. Unquoteds are soft (evaluated).
Single quoteds are constant in the Scanner.
Double quoteds are constant in the Function code.
(List….) is the ‘composer’.

#5 - Scanner Construction Process
[DecByteArrayEltConstructor.cl, ...Scanners.cl, ...Functions.cl]
(Two files produced; one Scanner and one instance Function for each)

Form of 'Constructor' File for generating multiple Scanner macros

```
    ;; - Construct and Write Scanner Macros -


    ;; ** 2D-raster-fwd, in1 **

    (setq macname '2D-Byte01-Array-Elt-Op-00)

    (setq codekey '(:2d-raster-fwd :in1 :byte nil))
    (setq imgargs '(inimg))
    (setq docstrx
          "Macro to apply an EXPR to each Point of a 2D-byte
image array, INIMG. Typically used to extract a measurement or
statistic from INIMG. NO type checking is performed on
the argument. OPTKEY is used to set the optimization on the
inner loop. A number of symbol macros are provided to make it
easier to express EXPR. Code returns INIMG.")

    (Create-Scanner genname macname docstrx codekey imgargs codestream)

    (Create-Scan-Instance macname imgargs othargs bodyarg funcstream)


    ;; ** 2D-raster-fwd, out1 **

    (setq macname '2D-Byte10-Array-Elt-Op-00)

    (setq codekey '(:2d-raster-fwd :out1 :byte nil))
    (setq imgargs '(outimg))
    (setq docstrx
          "Macro to apply an EXPR to each Point of a 2D-byte
image array, OUTIMG. Typically used to generate a new OUTIMG
based on a set of parameters. NO type checking is performed on
the argument. OPTKEY is used to set the optimization on the
inner loop. A number of symbol macros are provided to make it
easier to express EXPR. Code returns OUTIMG.")

    (Create-Scanner genname macname docstrx codekey imgargs codestream)

    (Create-Scan-Instance macname imgargs othargs bodyarg funcstream)


    ;; ** 2D-raster-fwd, out1/in1 **

    (setq macname '2D-Byte11-Array-Elt-Op-00)

    (setq codekey '(:2d-raster-fwd :out1/in1 :byte nil))
    (setq imgargs '(outimg inimg))
    (setq docstrx
          "Macro to apply an EXPR to each Point of a 2D-byte
array, INIMG, returning the result in a second array, OUTIMG. It is
also possible for INIMG and OUTIMG to be the same array. NO
type or size equivalence checking is performed on the arguments.
OPTKEY is used to set the optimization on the inner loop. A
number of symbol macros are provided to make it easier to
express EXPR. Code returns OUTIMG.")
```

```
(Create-Scanner genname macname docstrx codekey imgargs codestream)

(Create-Scan-Instance macname imgargs othargs bodyarg funcstream)
```

Newer Version of the Constructor File:

- Creates multiple Scanner macro files at once (data type families)
- Loops through combinatorial macro code choice variables
- Automatically constructs the macro names, arg lists, key lists, etc.
- Also constructs the 'docstring' as a full page description of macro

Create-Scanner:
MXP1 constructs def, Eval instantiates/loads

```
(defun Create-Scanner
    (genname scanname docstring codekeys imgnames ostream)

  "Function to construct and write a Scanner macro to OSTREAM
using the scan generator macro GENNAME, having name SCANNAME and
documentation DOCSTRING, based on CODEKEYS with *formal* image
arguments IMGNAMES. The resulting form is also Eval'd to load
the scanner definition, for creating a code example to check."

  (let ((form `(,genname    ;; e.g., Make-Array-Pnt-Scanner
                ,scanname   ;; w/args, e.g.: (optkey inimg &body expr)
                ,docstring
                ,codekeys
                ,@imgnames)))

    (eval
     (write
      (macroexpand-1 form)

      :miser-width 60
      :stream ostream) )

     (terpri ostream)
     (terpri ostream)
     (terpri ostream)
    ))
```

Example of generated Scanner Macros [...Scanners]:
[Equivalent to previously shown all-list macro form]

```
(defmacro 2D-Byte11-Array-Elt-Op-00 (optkey outimg inimg expr)
  "Macro to apply an EXPR to each Point of a 2D-byte
array, INIMG, returning the result in a second array, OUTIMG. It is
also possible for INIMG and OUTIMG to be the same array. NO
type or size equivalence checking is performed on the arguments.
OPTKEY is used to set the optimization on the inner loop. A
number of symbol macros are provided to make it easier to
express EXPR. Code returns OUTIMG."
  (With-Gensyms (goutimg ginimg gnrows gncols gr gc grow-y grow-x)
               (list
                'let*
                (list
                 (list goutimg outimg)
                 (list ginimg inimg)
                 (list gnrows (list 'Image-Rows ginimg))
                 (list gncols (list 'Image-Cols ginimg)))
                (cons
                 'declare
                 (list
                  (list 'type 'posfix gnrows gncols)
                  (list 'type 'byte-image-array goutimg ginimg)))
                (list 'declare (cons 'ignorable (list)))
                (list
                 'do
                 (list (list gr 1 (list '1+ gr)))
                 (list (list '> gr gnrows))
                 (list 'declare (list 'type 'posfix gr))
                 (list
                  'let
                  (list
                   (list
                    grow-x
                    (list
                     'the
                     'byte-image-vector
                     (list
                      'aref
                      (list 'the 'byte-image-array ginimg)
                      gr)))
                   (list
                    grow-y
                    (list
                     'the
                     'byte-image-vector
                     (list
                      'aref
                      (list 'the 'byte-image-array goutimg)
                      gr))))
                  (cons
                   'declare
                   (list
                    (list 'type 'byte-image-vector grow-y grow-x)))
                  (list
                   'declare
                   (cons 'ignorable (list grow-y grow-x)))
                  (list
                   'do
                   (list (list gc 1 (list '1+ gc)))
                   (list (list '> gc gncols))
                   (list 'declare (list 'type 'posfix gc))
                   (list
                    'declare
```

```
                    (cons
                     'optimize
                     (if optkey (Select-Optimization optkey) 'nil)))
                   (list
                    'symbol-macrolet
                    (list
                     (list 'nrows gnrows)
                     (list 'ncols gncols)
                     (list 'r gr)
                     (list 'c gc)
                     (list
                      'x
                      (list
                       'the
                       'u8
                       (list
                        'aref
                        (list 'the 'byte-image-vector grow-x)
                        gc)))
                     (list
                      'y
                      (list
                       'the
                       'u8
                       (list
                        'aref
                        (list 'the 'byte-image-vector grow-y)
                        gc))))
                   expr))))
               goutimg)))
```

<u>Create-Scan-Instance</u>:
MXP1 on scanner call, constructs 'real code' used within Function def.

```
(defun Create-Scan-Instance (scanname imnames othargs bodyarg ostream)

  "Function to construct and write a Scanner macro instance fn
to OSTREAM using the scanner macro SCANNAME, with :opt11 as
optkey, and having *actual* arguments IMNAMES and OTHARGS, as
well as BODYARG [Expr or Pred] in the call."

  (let ((form `(,scanname    ;; e.g., 2D-Byte11-Array-Elt-Op-00
                :opt11
                ,@imnames
                ,@othargs
                ,bodyarg)))  ;; = expr

    (format ostream "\;\; ~A~%~%" scanname)
    (write
     (macroexpand-1 form)

     :miser-width 60
     :stream ostream)

    (terpri ostream)
    (terpri ostream)
    (terpri ostream)
    ))
```

Example of Function code created from scanner [...Functions]:

```
;; Based on 2D-Byte11-Array-Elt-Op-00 macro call

(let* ((#:g1619 outimg)
       (#:g1620 inimg)
       (#:g1621 (Image-Rows #:g1620))
       (#:g1622 (Image-Cols #:g1620)))
  (declare (type posfix #:g1621 #:g1622)
           (type byte-image-array #:g1619 #:g1620))
  (declare (ignorable))
  (do ((#:g1623 1 (1+ #:g1623)))
      ((> #:g1623 #:g1621))
    (declare (type posfix #:g1623))
    (let ((#:g1626
           (the
            byte-image-vector
            (aref (the byte-image-array #:g1620) #:g1623)))
          (#:g1625
           (the
            byte-image-vector
            (aref (the byte-image-array #:g1619) #:g1623))))
      (declare (type byte-image-vector #:g1625 #:g1626))
      (declare (ignorable #:g1625 #:g1626))
      (do ((#:g1624 1 (1+ #:g1624)))
          ((> #:g1624 #:g1622))
        (declare (type posfix #:g1624))
        (declare (optimize (speed 1) (safety 1)))
        (symbol-macrolet ((nrows #:g1621)
                          (ncols #:g1622)
                          (r #:g1623)
                          (c #:g1624)
                          (x
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g1626)
                             #:g1624)))
                          (y
                           (the
                            u8
                            (aref
                             (the byte-image-vector #:g1625)
                             #:g1624))))
          (setf y 0)))))
  #:g1619)
```

Conclusions

- This presentation has demonstrated (*before your very eyes, ladies and gentlemen!!*) the Automatic Creation of Tens of Thousands of Optimized Lines of Lisp Code, *in seconds!*--in the form of Scanner macros for building image processing operations, based on a combinatorial decomposition of that space

- The development process I used was also described, showing how to get there:
  - Starting with a specific single image processing operation (Copy)
  - Moving to a first-level Scanner macro capable of generating many similar image operations, in a compact, easy to read form
  - Then to a second-level Generator macro capable of creating many Scanner macros of the same combinatorial 'family', but with different attributes, like the number of arguments, data types, scan direction, and default optimization

- All of this allows for the concrete ability to generate hundreds of macros and thousands of functions for an image processing operations library, in almost no time at all (once the machinery is set up:)

- One key element in accomplishing this (for me anyway), was the choice of using 'straight list' form, rather than the more common Backquote/Comma macro constructions, in writing and generating the macro code, especially the Scanners

- The other key element concerned how to subdivide the overall problem of building a large macro-generating macro, into four levels, in order to implement the decomposition of the operator space:
  - A structural template, its components (major code constructs–Generator)
  - The sub-components that make up each component (Scanner)
  - The sub-elements that may or may not appear in a given sub-component, based on informational elements used to guide the construction.

This ability to easily generate new combinatorial code automatically in Lisp, can empower programmers, saving an enormous amount of work, with a high degree of control, clarity, readability, and maintainability. (Not to mention being very exciting.)

I would hope that similar approaches to hypercoding with Lisp may be applicable to other problem domains beside image processing operator libraries, ones which also have a combinatorial character–the key remaining question being: How many or which types of problems have this character, or can be reformulated within this type of framework?

"And that, is the end of our tale."