

# Macintosh Common LISP

Michael S. Engber

engber@ils.nwu.edu (from  
InterNet)

engber@ils.nwu.edu@INTERNET#  
(from AppleLink)

>INTERNET:engber@ils.nwu.edu  
(from CompuServe)

not going to teach LISP  
(but free feel to ask LISP questions)

going to show off what's nice about MCL

talk mainly consists of demos

don't squint - code is in the handout

slides serve mainly to keep place in the handout

# MCL Environment

malleable environment - let's add a Font menu

```
(load "ccl:examples;font-menus.lisp")
```

Listener window handles standard i/o

```
(print "hello, world")
```

expressions in edit windows can be evaluated with a  
keystroke

```
(print "hello, world")
```

# Interface Designer

yet another graphical user interface editor

create a simple dialog

some generated LISP code

MCL doesn't use resources as templates for dialogs, menus, ...

Instead, they're generated from LISP code

```
(setf *d* (make-instance 'dialog
  :view-subviews (list
    (make-instance 'sequence-dialog-item
      :view-nick-name :seq
      :view-size #@(150 100)
      :table-vscrollp t
      :table-hscrollp nil)
    (make-instance 'button-dialog-item
      :view-nick-name :butt
      :dialog-item-text "push me"
      :dialog-item-action 'foo))))
```

edit UI while program continues to run

# Let's Hack - Process Manager

IM VI (p. 29-11) shows how to loop through current processes

LISP code to try this out  
returns a list of process names

```
(with-returned-pstrs ((process-name ""))

  (rlet ((psn      :ProcessSerialNumber
              :highLongOfPSN 0
              :lowLongOfPSN  #kNoProcess)
        (spec    :FSSpec)
        (pinfo   :ProcessInfoRec
              :processInfoLength (rlength :ProcessInfoRec)
              :processName       process-name
              :processAppSpec    spec))

    (let ((name-list nil))
      (loop
        (unless (zerop (#_GetNextProcess psn)) (return name-list))
        (unless (zerop (#_GetProcessInformation psn pinfo))
          (error "getting process info"))
        (push (%get-string process-name) name-list))))))
```

define a function from the code

```
(defun get-process-list ()
  ;;insert code here
)
```

dynamically link the code into the dialog box

```
(defun foo (di)
  (set-table-sequence (find-named-sibling di :seq) (get-process-list)))
```

# Let's Hack - DeskTop Database

```

;;init the DTDB refNum
(rlet ((pb :DTPBRec
        :ioNamePtr (%null-ptr)
        :ioVRefNum 0))
 (#_PBDTGetPath pb)
 (pref pb :DTPBRec.ioDTRefNum))

(defvar *DTDB-refNum* 1130)

;; GetComment
(with-pstrs ((fn "HD:TeachText"))
 (%stack-block ((buf 200))
  (rlet ((pb :DTPBRec
            :ioNamePtr fn
            :ioDTRefNum *DTDB-refNum*
            :ioDTBuffer buf
            :ioDirID 0))
    (when (zerop (#_PBDTGetComment pb))
      (%get-text buf (pref pb :DTPBRec.ioDTActCount))))))

;; GetIconInfo
(rlet ((pb :DTPBRec
        :ioDTRefNum *DTDB-refNum*
        :ioIndex 1
        :ioTagInfo 0
        :ioDTReqCount 1024
        :ioFileCreator "RSED"))
 (format t "~%~2@a: ~s ~3@s ~4@s~%" #\# 'type 'icon 'size)

(loop
 ;;break when #$afpItemNotFound
 (unless (zerop (#_PBDTGetIconInfo pb)) (return (pref pb :DTPBRec.ioResult)))
 (format t "~2@s: ~s ~3@s ~4@s~%"
  (pref pb :DTPBRec.ioIndex)
  (symbol-name (pref pb :DTPBRec.ioFileType))
  (pref pb :DTPBRec.ioIconType)
  (pref pb :DTPBRec.ioDTActCount))
 (incf (pref pb :DTPBRec.ioIndex))))

;; check out paul
(%stack-block ((buf #$kLarge8BitIconSize))
 (rlet ((pb :DTPBRec
          :ioDTRefNum *DTDB-refNum*
          :ioTagInfo 0
          :ioDTBuffer buf
          :ioDTReqCount #$kLarge8BitIconSize
          :ioIconType -1
          :ioFileCreator "RSED"
          :ioFileType "paul"
        ))
  (when (zerop (#_PBDTGetIcon pb))
    (print (pref pb :DTPBRec.ioDTActCount))
    (%get-text buf (pref pb :DTPBRec.ioDTActCount))))))

```

# Multiple Inheritance & Mixins

mixins can simplify class design

draggable-svm

one line of code creates a draggable view class

```
(defclass drag-item (draggable-svm static-text-dialog-item) ())
```

let's try it out

```
(setf *test-w*
      (make-instance
        'dialog
        :window-type      :document
        :view-position    :centered
        :view-size        #@ (200 100)
        :window-title     "draggable-svm demo"
        :close-box-p      t
        :grow-icon-p      t
        :view-subviews
        (list (make-instance
              'drag-item
              :view-position      #@ (10 20)
              :dialog-item-text   "change my position"
              :view-nick-name     :i1
              :dialog-item-action #'(lambda (di)
                                     (declare (ignore di))
                                     (ed-beep))
              :drag-end-action-fn #'(lambda (sv delta pt)
                                     ;end action moves the item
                                     (declare (ignore pt))
                                     (offset-view-position sv delta))
              :drag-bounds       :window
              )
            (make-instance
              'drag-item
              :view-position      #@ (10 50)
              :dialog-item-text   "drag me anywhere"
              :view-nick-name     :i2
              :dialog-item-action #'(lambda (di)
                                     (declare (ignore di))
                                     (print "hi,ho"))
              :drag-action-fn     #'(lambda (di)
                                     (declare (ignore di))
                                     (ed-beep))
              :drag-bounds        :none
              )))
      ))

;;nothing special about static text
;; redefine drag-item as a button & recreate the dialog
(defclass drag-item (draggable-svm button-dialog-item) ())
```

# Macros

## macros improve code legibility and reliability

```
(with-focused-view *top-listener*
  (with-text-state (:txSize 48 :txFace (ash 1 #$italic))
    (#_MoveTo 20 60)
    (with-pstrs ((str "the quick brow fox"))
      (#_DrawString str))))
```

Expanding the with-text-state yields:

```
(LET* ((#:G259 (%SETF-MACPTR (%NULL-PTR) (CCL::%GETPORT))))
  (DECLARE (DYNAMIC-EXTENT #:G259))
  (DECLARE (TYPE MACPTR #:G259))
  (LET ((#:G261 (PREF #:G259 :GRAFFPORT.TXFACE))
        (#:G263 (PREF #:G259 :GRAFFPORT.TXSIZE)))
    (UNWIND-PROTECT
      (PROGN
        (REQUIRE-TRAP TRAPS:_TEXTFACE (ASH 1 TRAPS::$ITALIC))
        (REQUIRE-TRAP TRAPS:_TEXTSIZE 48)
        (TRAPS:_MOVETO 20 60)
        (WITH-PSTRS ((STR "the quick brow fox")) (TRAPS:_DRAWSTRING STR)))
      (REQUIRE-TRAP TRAPS:_TEXTFACE #:G261)
      (REQUIRE-TRAP TRAPS:_TEXTSIZE #:G263))))
```

## full LISP language available at expansion time

macros can expand "intelligently"

generate minimal code

(in above example, only TextFace and TextSize traps are called)

detect and handle special cases

## examples of other useful macros

with-locked-GWorld

with-purgeable-resource

without-res-load

with-QDProcs

That's Nice

So What do I use MCL for?

development platform

prototyping tool

ToolBox exploration



# How to Get Started

buy MCL - it's not currently on ETO

get a LISP book (references in the paper)

accessing the ToolBox takes practice

look at oodles-of-utils + other PD code

# Q&A

What's  $1000!$  ( $1000 \times 999 \times 998 \times \dots$ )

Can you write a single program that compiles under LISP, C, Pascal, and FORTRAN?

How easy is it to implement RSA in LISP?

# A

## function for computing factorial

```
(defun fact (n) (if (plusp n) (* n (fact (1- n))) 1))
```

## compiles in LISP, C, Pascal & Fortran

Recall columns are significant in Fortran (anything past column 72 is ignored)

```

/* main() { puts ("Compiled by a C compiler"); } /*
*|# (print "Compiled by a Lisp compiler") #| *
  program chameleon
*)          (output);
*)    begin
          1 ( *,* ) 'Compiled by a FORTRAN compiler'
*)          ('Compiled by a Pascal compiler');
          end
#define end_pascal_comment |#'( *)
col 72
(* pi);/*
#|
          (*
          (*
writeln (*
          . (*

```

## compiles in LISP, C, & Pascal

Without Fortran the code is much less obfuscated

```

(* pi);/*
#|
*/ main() { puts ("Compiled by a C compiler"); } /*
|# (print "Compiled by a Lisp compiler") #| *
program chameleon (output);
  begin
    writeln('Compiled by a Pascal compiler');
  end.
*/
#define end_pascal_comment |#'( *)

```

## RSA is actually a pretty simple algorithm

treat your message as an integer

encrypt by raising message to a power (the public key)

decrypt by raising message to a power (the private key)

the catch is that these are large integers (hundreds of digits)

LISP handles integers of arbitrary size

```

(defvar pub)
(defvar pri)
(multiple-value-setq (pub pri) (RSA-gen-keys 47251 35747))

;;encode & decode
(RSA-decode-string (RSA-encode-string "the rain in spain" pub) pri)

;;digital signature
(RSA-decode-string (RSA-encode-string "it's me 6/20/92" pri) pub)

;;use droppable mixin to create an rsa dialog item
(defclass rsa-widget (droppable-svm static-text-dialog-item)
  ((public-key :accessor public-key
               :initarg :public-key)
   (private-key :accessor private-key
                :initarg :private-key)))

(defun encoder-fn (di target-di offset where)
  (declare (ignore offset where))
  (set-dialog-item-text target-di (RSA-encode-string
                                   (dialog-item-text target-di)
                                   (private-key di))))

(defun decoder-fn (di target-di offset where)
  (declare (ignore offset where))
  (set-dialog-item-text target-di (RSA-decode-string
                                   (dialog-item-text target-di)
                                   (public-key di))))

(setf *test-w*
      (make-instance
       'dialog
       :window-type :document
       :view-position :centered
       :view-size #@ (220 200)
       :window-title "rsa demo"
       :close-box-p t
       :view-subviews
       (list (make-instance
              'rsa-widget
              :private-key pri
              :view-position #@ (10 20)
              :dialog-item-text "drag & drop to encode"
              :view-nick-name :i1
              :drop-action-fn 'encoder-fn
              :drag-bounds :none
              )
             (make-instance
              'rsa-widget
              :public-key pub
              :view-position #@ (10 50)
              :dialog-item-text "drag & drop to decode"
              :view-nick-name :i2
              :drop-action-fn 'decoder-fn
              :drag-bounds :none
              )
             (make-instance
              'editable-text-dialog-item
              :wrap-p t
              :view-position #@ (10 80)
              :view-size #@ (200 100)
              :dialog-item-text "I sure hope the NSA isn't watching this demo"
              ))))

```