Macintosh Common LISP

Michael S. Engber
The Institute for the Learning Sciences
Northwestern University

Abstract:
Macintosh Common LISP (MCL) is a powerful development environment which is often overlooked by Macintosh programmers. This paper will show why you should consider using MCL and explore some of the ways it can enhance your productivity. Prior LISP or MCL experience is not assumed.

This paper is aimed at the typical Mac developer who uses C or Pascal. I'm going to try to avoid arguing the general merits of LISP versus other programming languages. There is already plenty written on this topic. Instead, this paper will emphasize aspects of MCL and LISP most relevant to Mac development.

Languages are a religious issue among programmers and opinions about LISP are even more polarized than most. So I'd like to apologize in advance for the editorial and first person comments, but one of the purposes of this paper is to get you to overcome any prejudice you have and take a closer look at Macintosh Common LISP. I could present just the cold facts, but that's not likely to be any more informative than reading the manual.

“LISP, doesn't that stand for Lots of Infernal Stupid Parentheses?”

If you're like most programmers, you probably regard LISP as a toy language, something you used years ago to do a few AI assignments. You were probably fluent in Pascal or C at the time, so LISP was an obstacle which made otherwise simple programs more difficult. That was certainly my experience. At the University of Wisconsin, we had a cluster of thirty or so Xerox Dandelions whose CRTs put out enough heat to keep the lab at 90 degrees in the dead of winter. Apart from Phys Ed, Intro to AI was the only course which required changing into a t-shirt and shorts. Ignoring the physical environment, just the simple act of logging in took forever, using the editor was difficult at best, and as for saving your files: don't even ask.

So, if you already have a bad taste in your mouth from LISP, let me start by assuring you that MCL is very easy to use. A novice can write `"hello, world"` faster in MCL than any other Mac development environment, even THINK Pascal. All you have to do is type `(print "hello, world")` and hit return. There are no libraries to include, no compilation scripts to run, you just type in your code and let MCL evaluate it.

On the surface, the editor looks pretty much like any other Mac text editor. In addition, it has an addictive code formatting command, an EMACS mode for EMACS fans, programming hooks from LISP so you can customize it to do anything you please, and a variety of LISP specific niceties. The compiler is editor aware, allowing you to select a section of code and execute it with a keystroke. MCL provides a variety of development tools including a stepper to let you execute your code one line at a time, and a backtrace tool that lets you explore the LISP stack (annotated and readable – not MacsBug style) after an error occurs.

MCL comes with an object library that supports the standard user interface elements. It offers all of the familiar benefits of object oriented user interface code. I'll forego the gratuitous examples of how to put up a dialog box, attach an action to a menu, or customize a dialog item. Given the widespread use of MacApp, I don't think the details of how you do this in MCL will be very enlightening.

Before going on, I'd like to tackle that classic LISP objection.

“Isn't LISP slow?”

First, LISP is not an interpreted language by definition. LISP can be compiled into efficient code. That's how MCL normally evaluates expressions. It compiles the expression and then executes the compiled code. This process is so fast that you'll probably think it's interpreting the expression.

Second, this question really needs to be directed at a particular LISP implementation rather than at LISP in general. It's true that there are aspects of LISP (dynamic linking, for instance) that introduce runtime overhead you don't have in C or Pascal, but the actual effect is less than most people imagine.

It's almost pointless to talk about speed without reference to a particular piece of code or algorithm. In addition, speed-critical sections of code comprise only a small fraction of most applications. I wish I could end the discussion here by just saying that MCL is generally fast enough to get the job done, but I don't imagine the skeptics will be satisfied unless I say more.

I'm going to try to divide and conquer the issue by addressing it in three parts: the user interface, the development cycle, and general execution speed. I'll attempt to make some meaningful generalizations about the first two and I'll wave my hands a bit at the third.

User Interface Speed

A user's impression of speed is strongly influenced by the user interface. Does it scroll fast? Do buttons highlight immediately when hit? Does dragging keep up with the mouse? Fortunately, this paper is focused on Mac programming, so I can also argue that this is what most of your code is devoted to.

MCL definitely excels when it comes to writing user interface code. I've implemented a variety of widgets and graphic effects in MCL including: PICT buttons, draggable dialog items, image dissolves, animation, on-screen video, and even an MDEF. In fact, MCL's own user interface is written using MCL's object library. Speed is not a problem. No one is going to be saying, "Boy, these dialog items take forever to draw," or "These buttons are sure sluggish."

This should come as no surprise since the ToolBox, especially QuickDraw, does most of the work. However, even the simple act of pressing a button involves more than just ToolBox calls. MCL detects the mouse down event and determines which window object should handle it. The window object has to determine which of its sub-views was hit. Views can be nested so this step recurs until we bottom out at the dialog item. The dialog item then highlights itself and begins tracking the mouse. It's important to point out that MCL's object system is implemented efficiently enough not to interfere with performance.

Development Cycle Speed

The speed of the compile/link/run cycle is important in evaluating a development environment. This is the biggest advantage that the THINK compilers have over MPW and probably one of the bigger complaints among MPW users. With dynamic linking, MCL excels in this area. The bigger the project, the more pronounced MCL's advantage becomes. The ability to recompile just the function you changed is hard to beat.

For developers, this will turn out to be biggest advantage of using MCL. The effect has to be seen to be believed. This is discussed further in the section on dynamic linking.

General Execution Speed

The best I can do is assert that MCL is fast enough for most purposes. If you're doing computationally intensive stuff, there's bound to be some code that just won't go as fast as you want. Probably the same code you find yourself wanting to hand code in assembly language.

Keep in mind that it's easier for novices to write inefficient code in LISP than in other languages. What I mean by inefficient is more serious than just calculating the same value twice; I mean turning an O(N) algorithm into an $O(N^2)$ algorithm (in time or space). This is hard to illustrate without getting into the details of LISP, but if you come up with an example of something that's really fast in C, yet your LISP version is slow as molasses, odds are it's the way you implemented it in LISP, not the LISP compiler.

Currently, MCL is slow dealing with double floats. This is a black mark if you're doing high-precision numerical analysis. This is strictly an implementation issue, not something inherent in LISP. It's reasonable to expect improvement in future releases.

Before you can really say anything definitive on speed, you have to try out MCL for your particular needs. In general, I think you'll be pleasantly surprised.

What LISP/MCL/CLOS offers over more traditional languages

Dynamic Linking

Many programmers have a hard time understanding what the big deal is about dynamic linking. "Sure, there's this new thing from Apple called Dinker. It lets you do dynamic linking of MacApp classes, but doesn't that just make it easier to distribute optional code modules for your product? Big deal."

In MCL, dynamic linking means being able to modify and add code continuously, even while your code is running. Say you're testing your program, you pull down a menu and it doesn't do what you expect. Maybe the code executed is totally bogus and generates an error. You locate and edit the function associated with that menu item's action. You hit the enter key to recompile that one function. The compilation only takes a second. Then you try the menu again. Notice that your program has continued

to run the whole time. This is the development cycle in MCL – you test and repair as you go along.

Dynamic linking encourages changing programs in small increments, testing each change before going on. In theory, that's how you're supposed to develop software. The long compile/link/run turnaround time in most development environments discourages this, especially when the program gets big.

For example, the above menu scenario played out using MacApp and C++ would have taken minutes instead of seconds.

I don't think you can fully appreciate dynamic linking until you use it. If you've written scripts in HyperCard you've had a taste of what dynamic linking offers. The ease with which you can try code out is very addicting. Once you get used to it, you won't want to go back.

Macros

In LISP, macros are far more powerful than `#define` is in C. Their syntax is much richer and you have the full LISP language available at macro expansion time.

As a common example from Mac programming, consider changing the drawing state of the current port, doing some drawing, and restoring the drawing state. It's pretty simple to do, but it clutters up your code and it's easy to make a mistake. Using macros improves your code's legibility and reliability with no cost at runtime.

Here are two simple examples. The first executes its body with the clip region temporarily changed. The second temporarily alters the pen state.

```
(with-clip-rgn some-rgn
   statement1
   statement2
   …)

(with-text-state (  :txMode #$srcBic
                    :txFace #$italic)
   statement1
   statement2)

(let* ((#:g210 (%setf-macptr (%null-ptr) (ccl::%getport)))) ;get the current port
   (declare (dynamic-extent #:g210))
   (declare (type macptr #:g210))
   (let (   (#:g212 (pref #:g210 :grafport.txface))         ;save current text face
            (#:g213 (pref #:g210 :grafport.txmode)))        ;save current text mode
      (unwind-protect
         (progn
            (require-trap traps:_textface traps::$italic)   ;set text face
            (require-trap traps:_textmode traps::$srcbic)   ;set text mode
            statement1                                      ;do text drawing
            statement2)                                     ;do more text drawing
         (require-trap traps:_textface #:g212)              ;restore text face
         (require-trap traps:_textmode #:g213))))           ;restore text mode
```

Figure 1 - Expansion of `with-text-state` Macro

This second form expands into the code shown in Figure 1. It looks nasty, but this essentially the same code you'd have to write in any language. In LISP you never see this mess unless you choose to expand the macro. In addition, there's no way to forget one of the steps. It's also worth pointing out that `with-text-state` accepts keywords for text font and size, but since they weren't used in this case, code wasn't generated to save and restore them.

Here's a brief list of some more interesting `with-xxx` macros along with the changes they temporarily affect.

`with-locked-GWorld`
  Locks the specified GWorld's pixels.

`with-purgeable-resource`
Loads the specified resource and makes it non-purgeable.

`without-res-load`
Turns resource loading off.

`with-QDProc`
Installs custom QuickDraw bottlenecks

`with-res-file`
Makes the specified resource file current. Takes keyword options to specify what to do in special cases like: file isn't open, file doesn't exist, file doesn't have a resource fork.

Error Cleanup

The `unwind-protect` LISP form allows you to guarantee a certain section of code will execute, even if an error occurs. This is useful for all sorts of things like ensuring a file will be closed, restoring the current port, disposing a handle. Most of the `with-xxx` macros previously presented expand into `unwind-protect` forms to ensure the changes they make will be undone. A typical use might be to ensure the disposal of a temporarily created region.

```
(let ((rgn (#_NewRgn))
 (unwind-protect
      (progn
            ;misc rgn calculations
            (#_FrameRgn rgn))
      (#_DisposeRgn rgn))
```

Multiple Inheritance

I know there are those who believe multiple inheritance is as evil as using goto statements, but I disagree. Not having it sometimes leads to clumsy and awkward class design. MacApp ended up introducing adorners to view drawing because it didn't have multiple inheritance to take care of the problem properly. There are times when multiple inheritance is the most direct and elegant solution.

For example, in MCL I have a class that handles dragging a dialog item around. If I want to make a draggable button, I create a dialog item that inherits from button and draggable. If I want to make an icon draggable, I create one that inherits from icon and draggable. No extra code is written apart from specifying draggable in their class inheritance lists. Without multiple inheritance you end up writing a special dragging method for each dialog item class. Isn't one of the points of object oriented programming to avoid duplicating code?

At this point someone will point out that I could put all the dragging code in the view which contains the items. I would argue that each class should handle dragging itself so it can easily customize the way it drags. This type of back and forth can continue endlessly. Let me end it here by saying that the Common LISP Object System (CLOS) supports multiple inheritance. I find it simplifies my code. If you don't want to use it, you don't have to.

Modest Development System Requirement

Since MCL applications are a bit large in terms of disk and memory space, you might mistakenly assume MCL itself is a major resource hog. MCL works adequately in its default 3M memory partition. At first glance, this may not sound modest, but remember there are no additional memory requirements. The partition is shared among the compiler, editor, debugger, and your code. As for disk space, the MCL compiler and libraries take up under 5M of disk space. This is about the same as Think C and significantly less than MPW. It’s feasible to use MCL on a 4M PowerBook 100. The same can’t be said for MacApp which wouldn’t even fit on the 20M hard drive.

Embedded Languages

Many of the high-end spreadsheets, communication packages, and word processors provide pseudo-programming languages for their “power users.” Most of these languages are poorly designed and a nightmare to use. Ever try using Excel's macro language? Applications written in LISP get the full LISP language and all its debugging tools for free. On other (non-Macintosh) platforms there are popular

LISP-based products, AutoCad and Interleaf, which take advantage of this.

If you don't think your users will like LISP's syntax, it's easy to write your own language using LISP. You just have to do enough to translate the user's source into LISP and you're done. Language design is one of the classic applications for LISP, so it has many features that make this easy to do.

Bignums, Fractions, and Complex Numbers

In addition to the numerical types you find in other languages, LISP provides an assortment of more exotic types. Bignums are integers which can be of any size. Fractions are rational numbers like 2/5 (i.e., no round off error, 7 * 1/7 equals 1 exactly). LISP even supports complex numbers. Needless to say, all the standard library functions operate correctly on any type of number they're passed.

I know I said I wouldn't get into issues not directly related to Mac programming, but MCL has a really impressive implementation of these types. It's hard to resist showing off MCL computing 1000 factorial. Consider this simple definition of factorial:

```
(defun fact (x)
 (if (<= x 1) 1 (* x (fact (- x 1)))))
```

It computes 1000!, a 2568 digit number, in 1.2 seconds on an SE/30 and less than half a second on a IIfx.[1]

You might ask, “Of what practical value is this?” Try implementing the RSA public key encryption algorithm. You need to do computations like raising x^y mod z where x, y, and z are 100 digit numbers. I used MCL to implement RSA along with the Solovay and Strassen prime test so I could generate 100 digit primes to use as keys. The point is, if your job was to implement RSA, I'd have the entire algorithm up and running in MCL long before you even got two 100 digit numbers multiplied together in C.

Is MCL of any practical use to developers?

I see three ways Mac developers can view MCL; as a delivery platform, as a prototyping environment, and as a ToolBox exploration environment.

MCL as a Delivery Platform

Yes, it's feasible to deliver software written in MCL, but don't count on using it to write 50K utilities. MCL is suitable for delivering large applications. It creates stand-alone applications by dumping an image of LISP memory to disk. These applications are big, starting at 1.5M and requiring a 2M RAM partition to run.

As your code grows in size, the application size doesn't really increase that fast. A typical LISP application might take up 4M on disk. However, RAM needs can go up more quickly. It's hard to make meaningful generalizations about RAM requirements. They are driven by application specific things like how many GWorlds you use and how many PICT handles you keep around, but it's not too unusual for a good-sized LISP application to require a 4M or 6M partition.

Before you hurl, take a look at the sizes and memory requirements of some of the high-end graphics programs and word processors. You'll see that MCL's requirements are comparable.

Don't make the mistake of extrapolating that your application will quickly grow to requiring 100M of

[1] In case you're interested, 1000! is:
40238726007709377354370243392300398571937486421071463254379991042993851239862902059204420848696940480047998861019719605863166687299480855890132382966994459099742450408
70737599188236277271887325197795059509952761208749754624970436014182780946464962910563938874378864873371191810458257836478499770124766328898359557354325131853239584630
75557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558
61103697680135730421616874760967587134831202547858932076716913244842623613141250878020800026168315102734182797770478463586817016436502415369139828126481021309276124489
63599287051149649754199093422215668325720808213331861168115536158365469840467089756029009505376164758477284218896796462449451607653534081989013854424879849599533191017
23355556602139450399736280750137837615307127761926849034352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476088
50627686296714667469756291123408243920816015378088989396451826324367161676217916890977991190375403127462228998800519544441428201218736174599264295658174662830295557029
90243241531816172104658320367869061172601587835207515162842255402651704833042261439742869330616908979684825901254583271682264580665267699586526822728070757813918581788
89652208164348344825993266043367660176999612831860788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084024432438465657245
01440282188525247093519062092902313649327349756551395872055965422874977401141334696271542284586237738753823048386568897646192738381490014076731044664025989949022222176
59043399018860185665264850617997023561938970178600408118897299183110211712298459016419210688843871218556461249607987229085192968193723886426148396573822911231250241866
49353143970137428531926649875337218940694281434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506217095002597389863554277
19674282224875758676575234422020757363056949882508796892816275384886339690995982628095612145099487170124451646126037902930912088908694202851064018215439945715680594187
27489980942547421735824010636774045957417851608292301353580818400969963725242305608559037006242712434169090041536901059339838357779394109700277534720000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000

RAM. Once you pay the initial overhead, you're over the hump. Doubling the functionality of your program doesn't double the size or memory requirements. All that user interface code is already there and can be reused. If this argument sounds familiar, you've probably heard it given in the early days of MacApp when people complained about its overhead.

MCL as a Prototyping Environment

I'm sure the term prototyping strikes fear into the hearts of many programmers. It probably brings back memories of writing a quick demo in HyperCard only to find yourself hacking it into a bigger and bigger mess because there was never enough time to go back and do a total rewrite in Pascal.

With MCL you're not going to run into the same roadblocks you do with HyperTalk. LISP is a fully developed programming language; you won't find yourself wishing for data structures. MCL gives you access to the ToolBox, so you won't have to extend the language with XCMDs. You can take your prototypes as far as you want, even turning it into a finished product.

I won't claim MCL is as easy to use as HyperCard, but a novice LISP coder will be able to get working dialog boxes and menus up in an afternoon. He won't have to learn MPW, he won't have to know what resources are, and he won't have to know what a handle is.

MCL as a ToolBox Exploration Environment

I find the best way to read "Inside Macintosh" is with MCL up and running so I can try things out as I go along. MCL lets you evaluate the ToolBox call without having to build a supporting program.

Why is this such a big win? How many hours have you wasted because you wrote a bunch of code only to throw it out because a ToolBox call doesn't work the way you thought it would? Sometimes it's a poor description in "Inside Macintosh," other times the ToolBox is so complex that it defies written description. The solution is to try things out, but because it's such a pain, most programmers don't. You have to create a project, write code to put up a window, make your ToolBox call, convert your results into strings, and write code to print the strings. In MCL, all you have to do is make the call and use standard LISP i/o primitives to print the results.

Here's an example. Have you ever look at the description of `PBGetVInfo`? It's nasty. If `ioVolIndex` is positive, it does one thing; if it's zero, it does another; if it's negative, it does yet another. By the time you're done reading it, you can't even remember what you originally set out to do. In MCL you can try the call with three different values for `ioVolIndex` in less time than it takes to reread the description. Better yet, you can be confident you got it right. You won't be saying to yourself, “I'll try this for now, but if my code doesn't work I'll have to remember to come back here again.” Or, “Should I take the time to make sure this call works in all cases? Nah.”

Using MCL to explore the ToolBox

Of the three uses I've presented, ToolBox exploration is the only thing I can demonstrate quickly. Certainly, the biggest gains to be made from MCL come from using it as your implementation language, but it would be tough to demonstrate that in a short example and without first teaching you some LISP.

So in this section, I'll scratch the surface of MCL's power by showing how easy it is to access the ToolBox. Even if this is all you ever use MCL for, it will be well worth the price. The examples I'll present are taken from the development of a freeware utility, Save A BNDL, which I recently released. Save A BNDL was prototyped in MCL and then translated to Think C for compilation into a 15K application.

Save A BNDL installs a file's `BNDL` information into the Finder without requiring rebuilding the desktop or rebooting. It uses the desktop database, the process manager, System 7 file manager calls, and apple events. These are all things I'd never done before, so of course I immediately got MCL out and tried them.

The code that follows is what I used to explore the desktop database. The code is initially straightforward, but one quickly realizes there are far more subtleties to the desktop database than “Inside Macintosh” would lead you to believe. MCL made this exploration process much less painful

than it might have been.

But first, some basic syntax

Even if you don't know LISP, you should be able to get the gist of this example code. In fact, knowing LISP might even be a hindrance since this code uses MCL extensions to access the ToolBox. Here are a few things you should know.

- LISP symbols are not limited to alpha-numeric characters as they are in most languages. For example, `%stack-block` is a legal function name.
- Function calls are made using the syntax:
  `(function-name arg1 arg2 …)`
- Trap calls look like function calls, but they begin with `#_`.
- LISP data is not in the same format as ToolBox data. The fact that LISP integers can be 100 digits long may have tipped you off that they aren't simply stored as 32 bits. This means that data passed to the ToolBox must be allocated and retrieved in a special way. Table 1 covers what you need to know for the purposes of this paper.

example

explanation

```
(rlet ((pb :DTPBRec
           :ioNamePtr (%null-ptr)
           :ioVRefNum 0))
```

`rlet` declares and initializes a Pascal style record. The example declares `pb` of type `DTPBRec` and initializes two of its fields.

```
(%stack-block ((buf 200))
```

`%stack-block` declares an untyped variable and allocates space on the stack for it. The example binds `buf` to 200 bytes.

```
(with-pstrs ((fn "HD:TeachText"))
```

`with-pstrs` declares and initializes a Pascal style string. The example declares `fn` to be string containing a full pathname.

```
(pref pb :DTPBRec.ioDTRefNum)
```

`pref` references a field of a Pascal style record. The example returns the `ioDTRefNum` field of a `DTPBRec` variable.

```
(%get-text buf 10)
```

`%get-text` returns a LISP string created by using the specified bytes of memory as ASCII code. The example returns a 10 character string created from the memory pointed to by `buf`.

Table 1 - Access to ToolBox Data from MCL

And now, for the code

Start by using `PBDTGetPath` to get the reference number of the desktop database.

```
(rlet ((pb :DTPBRec
           :ioNamePtr (%null-ptr)
           :ioVRefNum 0))
  (#_PBDTGetPath pb)
  (pref pb :DTPBRec.ioDTRefNum))
```

➔ `754`

Create a global variable for the desktop database reference number so we can refer to it symbolically.

```
(defvar *DTDB-refNum* 754)
```

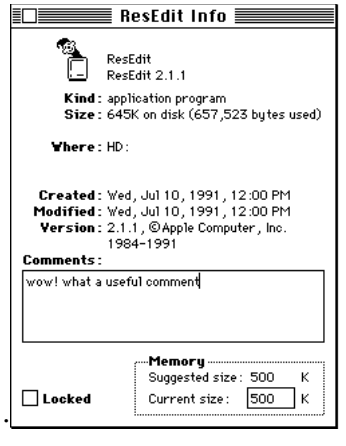
➔ `*DTDB-refNum*`

Try out `PBDTGetComment` which returns the Finder comment associated with a file.

```
(with-pstrs ((fn "HD:ResEdit"))
  (%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))))))
```

➔ `"wow! what a useful comment"`

A quick check with the Finder's Get Info command verifies it's the correct value.



`PBDTGetIconInfo` returns the file type, icon type, and icon size of icons entered in the desktop database for the specified creator, in this case ResEdit. It's an indexed call. You make it repeatedly until it returns `afpItemNotFound`.

```
(rlet ((pb :DTPBRec
           :ioDTRefNum    *DTDB-refNum*
           :ioIndex       1
           :ioTagInfo     0
           :ioDTReqCount  1024
           :ioFileCreator "RSED"))
```

```
;format is roughly equivalent to a printf in C
(format t "~%~2@a: ~s  ~3@s   ~4@s~%" #\# 'type 'icon 'size)
(loop
 ;break on error or when afpItemNotFound is returned
 (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))))
```

→

```
 #:  type  icon   size   all the icons associated with ResEdit
 1: "APPL"    1    256
 2: "APPL"    2    512
 3: "APPL"    3   1024
 4: "APPL"    4     64
 5: "APPL"    5    128
 6: "APPL"    6    256
 7: "RSRC"    1    256
 8: "RSRC"    2    512
 9: "RSRC"    3   1024
10: "RSRC"    4     64
11: "RSRC"    5    128
12: "RSRC"    6    256
13: "paul"   -1    256   ← what's this? [2]
14: "rsrc"    1    256
15: "rsrc"    2    512
16: "rsrc"    3   1024
17: "rsrc"    4     64
18: "rsrc"    5    128
19: "rsrc"    6    256
20: "ssrc"    1    256   ← stationary document icons
21: "ssrc"    2    512
22: "ssrc"    3   1024
23: "ssrc"    4     64
24: "ssrc"    5    128
25: "ssrc"    6    256
-5012                                   ← afpItemNotFound
```

Closer inspection reveals that the `paul` icon isn't actually an icon at all. It's used by the Finder to store the file types which can be dragged and dropped onto the application. This code uses `PBDTGetIcon` to retrieve the raw icon data. It then prints it out as a text string.[3]

```
(%stack-block ((buf #$kLarge8BitIconSize))
  (rlet ((pb :DTPBRec
             :ioDTRefNum *DTDB-refNum*
             :ioTagInfo 0
             :ioDTBuffer buf
             :ioDTReqCount #$kLarge8BitIconSize
             :ioIconType -1
             :ioFileCreator "RSED"
```

[2] Notice the undocumented type of icon associated with `paul` files. It turns out that most applications have an entry for this mysterious `paul` icon in the desktop database, but none of them seem to have it in their `BNDL` resource. How odd.

[3]

```
                :ioFileType "paul"
                ))
        (when (zerop (#_PBDTGetIcon pb))
          (print (pref pb :DTPBRec.ioDTActCount))
          (%get-text buf (pref pb :DTPBRec.ioDTActCount)))))

→
 256
"rsrcRSRC****" ← This is actually a 256 character string.  The 244  trailing null characters aren't shown.
```

As you can see, it's pretty simple to access the ToolBox; no windows to create, no managers to initialize, no make-files to write. When code is evaluated, the resulting value is printed to the Listener window, a standard part of the MCL environment. The `print` function outputs to the Listener by default.

MCL played a pivotal role in the development of Save A BNDL. First, I used it to learn about the desktop database, including the existence of the undocumented `paul` icon. Then, I used it to figure out how to kill and restart the Finder using Apple Events and the Process Manager. Before a single line of C code was written, I had already solved most of the interesting problems.

"If MCL is totally awesome, why isn't everyone using it?"

Arguments against using MCL are usually along the lines of: "No one else is using MCL;" "LISP might be too slow;" "I already have a big investment in C;" "Why should I learn something new?" Do these objections sound familiar? They sound a lot like the complaints the PC community raised when the Macintosh first came out. They are all pretenses for avoiding change.

If you're still not sold on MCL, remember that I've kept this paper focused on the benefits of MCL directly related to Mac programming. There are also plenty of reasons to choose LISP over traditional programming languages. For a set of relevant articles, see the September 1991 issue of the Communications of the ACM. It has a special section on LISP and CLOS.

There's been a lot of talk about Object Oriented Dynamically Linked languages (OODLs) being the wave of the future. Object oriented languages have already changed the face of Mac development. Dynamic languages have the potential to completely revolutionize it. MCL is not some promise of the future. It's here now, it works, and you can benefit from it.

References and Suggested Reading

Apple Computer, "Macintosh Common LISP 2.0 Reference - Draft." 1991, Apple Computer.
Comes with MCL (available from APDA). I haven't seen the final version yet. The 2.0b1 draft has plenty of errors and omissions. MCL's Apropos tool goes a long way toward making up for the deficiencies.

Card, Orson Scott, "Ender's Game." 1991, Tom Doherty and Associates.
Very entertaining. Even people who don't normally like science fiction will enjoy this one.

Engber, Michael S., *The Sound Manager with LISP*. MacTutor, March 1991, pp. 84-89.
An informative and well written article, if I must say so myself. It covers the basics of ToolBox access and illustrates them by using the Sound Manager. It was written in the days of MACL 1.32, so parts are dated.

Keene, Sonya E., "Object-Oriented Programming in Common LISP." 1989, Addison Wesley.
A very complete and digestible introduction to CLOS. It can be read straight through (if you ignore that extended example on the lock class).

Kleiman, Ruben, The Power of Macintosh Common LISP, d e v e l o p, Winter 1991, pp. 85-113.
Covers the basic MCL environment and object system in detail. Lots of example code.

Norvig, Peter, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp" 1992, Morgan Kaufman Publishers, Inc.
Common Lisp from a to z, with lots of good examples and two chapters on efficiency issues.

Steele, Guy L., "Common LISP - The Language" 2nd edition. 1990, Digital Press.
Comprehensive, precise, essential, very dense. This is one of those books whose prerequisite is a solid

understanding of the subject it covers. Not a tutorial.

Sherwood T.K. and Wilcox F.C., "Sabotage of Gasoline Engines." 1946, Office of Scientific Research .
A definite must.

Wilensky, Robert, "Common LISPcraft." 1984, WW Norton and Company.
If you already know something about programming and you're looking for a book you can sit down, read, and come away with the impression, albeit mistaken, that you know something about LISP, this is it. It has good, readable, explanations of the fundamentals. Appendix A is a good Common LISP reference, although not as encyclopedic as Steele. When you outgrow Wilensky, you'll be ready for Steele.

Acknowledgments

I’d like to thank the following individuals for their input: Jorn Barger, Mark Chung, Martha Engber, Josh Golub, Dan Halabe, Alice Hartley, Mike Korcuska, Rich Lynch, David Moon, David Neves, Tamar Offer, Chris Riesbeck, Bill St. Clair, and Steve Strassmann.

The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting, part of The Arthur Andersen Worldwide Organization. The Institute receives additional funding from Ameritech (an Institute Partner), IBM, the Defense Advanced Research Projects Agency, the Air Force Office of Scientific Research, and the Office of Naval Research.

The `paul` icon is actually an array of `OSType`'s. Notice it contains an entry for files of type `****`. This is a wild card type you can specify in a `BNDL` resource if you want to accept files of any type. Other wild card types are: `fold` to accept folders, `disk` to accept disks, and `????` to accept applications. This last type isn't documented, but without it there's no way to only accept `APPL` files (recall the `APPL` entry in a `BNDL` specifies the application's icon).