The AutoLISP Platform for Computer Aided Design

Harold Carr   Robert Holt

Autodesk, Inc.
111 McInnis Parkway
San Rafael, California 94903
harold.carr@autodesk.com   rhh@autodesk.com

Introduction

Over fifteen years ago, Autodesk, then an unknown company headquartered in Marin, California, revolutionized the CAD industry with the introduction of a truly usable drafting package able to run effectively on personal computers. Within a few years, this same company sparked a second revolution with the introduction of the Lisp programming language as the system's primary customization tool. Since then, there have been dozens of books written, hundreds of courses offered, and thousands of technical columns devoted to empowering a wide spectrum of CAD professionals with the tools of AutoLISP.

Recently, Autodesk has revitalized its investment in Lisp technology with a new implementation of the AutoLISP system known as Visual Lisp. This environment satisfied many long-standing requests from the AutoLISP user base including a compiler, editor, and debugging environment. More significantly, the new system provides an unprecedented level of system integration in its role as both ActiveX client and server. This paper reviews the history of AutoLISP, its role in the success of AutoCAD in the market, new features introduced via Visual Lisp, and possible future architectural directions for this new generation of Lisp at Autodesk.

In the Beginning…

> "Lisp?!?! *Why the Heck did you pick the most arcane, obscure, and hopelessly-rooted-in-the-computer-science-department language in the world for an AutoCAD programming language?"* [1]

So, rhetorically, began one of the first internal Autodesk papers discussing the selection of Lisp as the primary macro language for AutoCAD. Obviously aware of the potential controversy that could follow such a choice, John Walker, (one of Autodesk's founders) made a series of cogent arguments that reinforced this decision which, in the view of many, played a major part in the success AutoCAD has enjoyed in the computer aided design market. Later in the same memo, he continued,

> *"No other major programming language can so easily manipulate the kinds of objects one works with in CAD. As opposed to numerical programming, CAD constantly works on collections of heterogeneous objects in variable sized groups. Lisp excels at this."*

AutoLISP traces its origins back to a particular dialect of LISP known as XLISP, authored by David Betz. Introducing it into AutoCAD required some basic changes such as implementing floating point as a fundamental type. The extent to which additional capabilities could be added, however, was constrained by the requirement at the time that the entire LISP system fit into a single 64K address space. The new system was partially introduced in 1985 as part of AutoCAD release 2.1, then more completely unveiled with release 2.16.

Command Performance

AutoCAD, at least at that stage of its development, was inherently command based. In other words, the AutoCAD user typically operated the system by entering a particular command (e.g., LINE, CIRCLE, TRIM, EXTEND) at a command line, then responding to a series of prompts which further qualified the desired operation.

One of the first and most fundamental extensions to the basic LISP engine was the introduction of a new function, "*command*", which allowed a LISP program to drive the command system. (Or, colloquially, to "*shove tokens down the command throat.")* Users could, for the first time, create abstractions specific to a particular problem domain building on the underlying base geometry set. The small excerpt below, for example, defines a routine which will draw a simple representation of a door. (see Figure 1.)

```
;;; Create a picture of a door
;;; given two points defining the lower left and upper right corners.
(defun door (lowerLeft upperRight / llx lly urx ury knobCtr knobRad)
  (setq llx (car lowerLeft)   ; decompose the corners
        lly (cadr lowerLeft)
        urx (car upperRight)
        ury (cadr upperRight)
        upperLeft (list llx ury)
        lowerRight (list urx lly))
  (setq knobCtr (list         ; compute a center for the door knob
                 (xpercent 0.9 llx urx)
                 (xpercent 0.4 lly ury)))
  (setq knobRad (* 0.05 (- urx llx)))
  (command "_LINE"      ; create the frame with the "LINE" command
           lowerLeft
           upperLeft
           upperRight
           lowerRight
           lowerLeft
           "")
  (command "_CIRCLE" knobCtr knobRad)) ; create the knob as a circle

;;;Interpolate between two bounds
(defun xpercent (frac lowerBnd upperBnd)
  (+ lowerBnd (* frac (- upperBnd lowerBnd))))
```

The primary role of the AutoCAD command environment also dictated a particular AutoLISP naming convention. Lisp functions whose names began with `C:` implicitly defined new AutoCAD commands that a user could enter at the command line. Thus, in the earlier example, instead of having to enter `(door '(0 0) '(4 8))` to invoke the function, the user could simply type "door".

The call to (door) might be wrapped as follows.

```
(defun C:door ( / lowerLeft upperRight)
      (setq lowerLeft (getpoint))
      (setq upperRight (getpoint))
      (door lowerLeft upperRight))
```

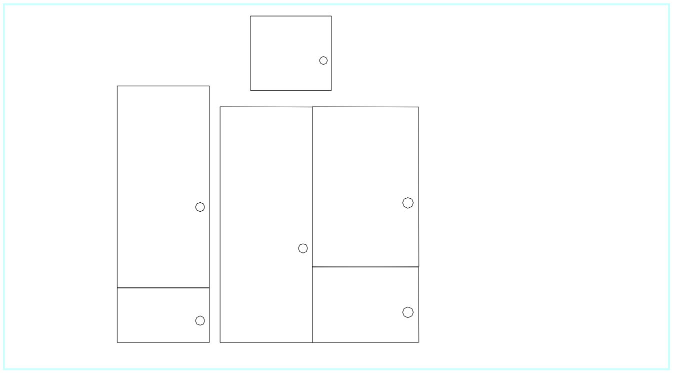


Figure 1 -- Some Sample AutoLISP Doors

Accessing CAD data as list structures

Besides creating geometry from an AutoLISP program by using AutoCAD command set and programmatically issuing sequences of commands, many applications need to interrogate the contents of a given drawing.

AutoCAD, at some level, can be thought of as an interactive editor for a collection of geometrical objects that can be composed to create a desired technical drawing. These objects can be as simple as a line segment or as complex as a full three-dimensional solid model. In order to gain access to these objects within an AutoCAD database from an AutoLISP program, a suite of functions was introduced which relied on the ability of any object within the database to express its state as a simple association list. The “key” for any of the pairs in the *alist* is always an integer, whose value dictated the interpretation of the “data” part of the pair. The association list of a circle entity within the system, for example, appears as

```
((-1 . <Entity name: d13d50>)      ; -1 indicates internal identifier for this circle object
  (0 . "CIRCLE")                   ; 0 indicates type of object
  (330 . <Entity name: d13cc8>)    ; 330 indicates internal identifier of the circle’s container
  (5 . "52")                       ; 5 indicates the “handle” of the object
  (100 . "AcDbEntity")             ; first 100 indicates the most ancestral class of the object
  (67 . 0)                         ; 67 indicates the circle’s color
  (8 . "0")                        ; 8 indicates the circle’s layer
  (100 . "AcDbCircle")             ; subsequent 100 indicates more derived class of object
  (10 3.0 5.0 0.0)                 ; 10 indicates center of circle
  (40 . 1.6)                       ; 40 indicates circle radius
  (210 0.0 0.0 1.0)                ; 210 indicates vector normal to plane of circle
)
```

Three functions provide access between the database and an AutoLISP program:

`(entget <entity>)` returns an association list reflecting the current state of the given entity

`(entmake <list>)` creates a new entity based on the definition of the given list and appends it to the database

`(entmod <list>)` changes a particular set of the properties of the entity referenced by the given list.

As a very simple example, the construct

```
(entmake (entget (entlast)))
```

creates a copy of the last entity in the database, and adds it to the model.

Synergy with DXF

The approach of expressing the state of the AutoCAD database as a set of key/value pairs, while natural from a Lisp perspective, actually originated in the “Drawing Exchange File” (DXF) standard. This format, intended to provide a publicly documented version of the content an AutoCAD drawing, consists of an ASCII file with pairs of lines reflecting the key/value pairs returned by the `entget` function.[1] DXF has emerged as one of the primary industry standards for sharing geometric data among various CAD systems. The similarities between this file structure, and the AutoLISP association list allowed a good deal of existing code, with some small modifications, to process both an external DXF file as well as a live database from within an AutoCAD session.

Extended Entity Data

While the core AutoCAD database was designed largely to contain basic geometric information about the entities within a CAD drawing, applications built on top of the AutoLISP system quickly found the need to record various types of problem-specific information in the database as well. Early architectural applications, for example, often represented a wall as a set of line entities for purposes of visual display, but may have actually known substantially more about the wall’s properties, such as its thickness, beam spacing, material characteristics, etc.

AutoLISP and the underlying AutoCAD database were enhanced to allow applications to associate domain-specific information with any given entity in the database. Developers would simply create an association list of their own which reflected the extended state of a particular entity, append this to the basic list of properties returned via `entget`, and submit the expanded list back to AutoCAD using `entmake` or `entmod`.

This application specific data, known as *extended entity data*, was then treated as a fundamental component of the object – it was copied when the entity was copied, filed out when the drawing was saved, restored when the drawing was opened, etc. When an application needed to retrieve the extended data associated with an object, it simply added an extra argument to the `entget` call to obtain a list including both the standard AutoCAD data as well as any application specific data.

User Interaction and AutoLISP

AutoLISP further extended the basic XLISP function set by including a set of primitives that allowed applications to interact with a user in a style consistent with the standard AutoCAD commands. A set of new functions, known as the `getXXX` complex, provided simple access to prompt for integers, reals, strings, angles, distances, etc., with a rudimentary level of data validation incorporated.

While conceptually simple, the significant feature of these functions is that they provide an entry in to the rather complex input acquisition logic of the AutoCAD editor. Even a request as basic as `(getdist`) which prompts a user for a scalar value, could be satisfied either by the user entering a number, or by the user identifying two points. In this latter case, each point could be specified

[1] The correspondence is not always one for one. Values that are not single numbers, e.g. a 3-dimensional point, are broken into three individual entries in the DXF file.

in several ways, including pointing on the screen, snapping to features of existing geometry, or simply typing in a polar or Cartesian coordinate.

The ultimate result of these extensions is that Lisp programmers were presented with an environment sufficiently rich to allow essentially seamless extension of the native AutoCAD command set, including the basic style of command interaction. AutoCAD had graduated from a CAD program into an application platform. And in spite of all the trepidation, practicing architects (who still secretly coveted paper and pencil) started to use and enjoy this “most arcane, obscure” language.

AutoLISP and ADS

As users and application developers began building increasingly powerful solutions on top of the AutoLISP environment, computation speed quickly surfaced as a critical barrier for many problems. AutoLISP was an interpreted system and was simply never designed to handle many of the numerically intensive solutions people wished to build with it.

In a clever, if not somewhat twisted attempt to quiet these complaints, a new C language API to AutoCAD was introduced known as the AutoCAD Development System (ADS). This new API was essentially a one-for-one mapping of the AutoLISP functions into C routines. The fundamental data structure in C became a singly linked list of “resbuf” structures which could in principle represent any Lisp s-expression. The “result buffer”, or `resbuf` was simply a union of all the native Lisp types including floats, integers, strings, points, etc., together with a simple forward linking structure.

The mapping from AutoLISP function to ADS equivalent was almost mechanical. `Entget`, for example had as its counterpart `struct resbuf *ads_entget().` Similarly, `entmake` led to `ads_entmake( struct resbuf *).`

This approach, while stylistically questionable from the C point of view, accomplished its main objective: creating a bridge from the AutoCAD/AutoLISP complex to a compiled application capable of performing significant computations at native machine speed.

Since that time, AutoCAD, while still supporting the ADS constructs, has moved away from the “resbuf” style of interaction to a fully object-based approach known as “ObjectARX™”.

The Extended Community

Since its introduction, there has been a strong support network of AutoLISP users that has played an important role in the widespread adoption of AutoLISP. In addition to an open newsgroup dedicated to AutoCAD customization issues, some of the leading CAD magazines feature a monthly column on various AutoLISP techniques. The author of one of these columns, “Hot Tip Harry”, has recently published a book detailing two hundred of the most popular AutoLISP routines.[2]

One non-scientific, but somewhat interesting measure of the level of interest in AutoLISP is that in a search of the general topic “lisp” in Amazon.com, over twenty percent of the retrieved publications were devoted to AutoLISP.

Large-scale Application Development

While the majority of AutoCAD users tend to rely on AutoLISP primarily as a powerful macro language, there is an important minority that authors substantial applications, either for in-house use, or for sale commercially completely on top of AutoLISP. Some sites we have contacted, for example, have over 800,000 lines of Lisp code that fully automates their design processes and drafting standards. With the introduction of Visual Lisp, such large scale applications should be even more effective in the future.

The Advent of Visual Lisp

Despite its popularity and widespread adoption by the AutoCAD community, AutoLISP clearly had its limitations. As mentioned earlier, execution speed proved to be one barrier that required some developers to migrate substantial portions of their application into C or C++. Another longstanding concern, especially for commercial developers, was the fact that there was not a particularly robust way to protect their intellectual property when deploying their applications.

Various attempts were made over the years to solve some of those shortcomings. One company in particular, Basis Software, invested several years in providing a full Lisp development environment which included the ability to produce compiled code. This product, known as Vital Lisp, addressed several of the deficiencies of AutoLISP while still offering an extremely high level of compatibility. Just over a year ago, Autodesk acquired this technology from Basis, and has been integrating the core of this system into AutoCAD as the Visual Lisp product.

Visual Lisp Architecture

The Visual Lisp Architecture consists of several subsystems: a virtual machine-based (VM) runtime system (RTS), a visual integrated development environment (IDE), a Lisp API to ObjectARX, a general ActiveX client/server subsystem (which includes access to the AutoCAD Automation API), and the interface by which AutoCAD utilizes Lisp services (i.e., accessing Lisp variables or calling Lisp functions from AutoCAD or other external ObjectARX applications).

Runtime System

The RTS is centered around a stack-based VM written in C++. The Lisp system code is written in Lisp Plus Plus (LPP), a lexically scoped Lisp1 [3]. The basic LPP types (numbers, symbols, pairs, strings, vectors, I/O streams, functions, etc.) are defined as C++ classes deriving from a single root (except for 31 bit immediate integers --- the only non-reference type). The garbage collector is a stop-and-copy collector with separate pages allocated for each type.

LPP system code is written in either LPP/C++ or LPP/Lisp. All LPP/Lisp code is compiled to VM instructions, either on-the-fly or to files. AutoLISP, a dynamically scoped Lisp1, is handled by an alternate front-end to the compiler. In this case, it compiles function entry/exit to alternate VM instructions which accomplish the dynamic binding protocol of function parameters, including error unwinds.

AutoCAD is enabled to support multiple documents in a single session. The RTS maintains a separate Lisp namespace for each document (i.e., drawing). Explicit communication between document namespaces is currently not allowed, although there is a global "blackboard" namespace accessible from all document namespaces. It is also possible to set the value of the same-named variable in all namespaces to a copy of its value in a given namespace. This setting will take place even on drawings opened after such variable value "propagation." There are also mechanisms for specifying that certain files or code be loaded into each namespace when a drawing is opened or created.

Integrated Development Environment

The optional (un)loadable IDE provides a complete set of tools for developing AutoCAD and/or ActiveX applications. It supports visual, expression-level stepping of source code, breakpoints, watch windows, backtrace, etc. The editor buffers provide console logging, parenthesis matching, match completion, find/replace, etc. Help is provided by online documentation and apropos features. The IDE itself is written in LPP's object system, which resembles CLOS.

Code developed via the IDE may be deployed in a number of formats: as text files to be loaded into AutoCAD (at which time it will be compiled while loading), as compiled VM code (with `*.fas` extension by convention), as a "project" file containing compiled code from multiple Lisp source files, and as a "VLX" application which will run in its own Lisp namespace. A VLX application may contain compiled Lisp from multiple source files and other resources such as text, DCL (AutoCAD's Dialog Control Language) code, and VBA code. This makes VLX the primary delivery vehicle for AutoCAD applications developed in Lisp.

VLX applications are the Visual Lisp equivalent of ObjectARX applications. A VLX application explicitly exports the names of functions (and only functions) to be made accessible from document namespaces. When a VLX application is associated with a document namespace (by "loading" it in the document), "envelope" functions, representing the VLX exported functions, are created in the document namespace.

The envelope functions copy argument and return values between the two namespaces. Although the heap is currently shared between all namespaces, the copying enables us to maintain a separate heap for each document and VLX namespace in the future. It also provides isolation between namespaces such that side-effects in one namespace will not be shared. For example, if

```
(vl-doc-export 'vlx-ex)

(setq *previous* nil)

(defun vlx-ex (x / retval)
  (setq retval (list (eq *previous* x) (equal *previous* x) x))
  (setq *previous* x)
  retval)
```

is compiled into a VLX application name `vlx-ex` and

```
(load "vlx-ex.vlx")
(setq test '(one 2 3.0 "four"))
(vlx-ex test)                      ; => (nil nil (ONE 2 3.0 "four"))
(vlx-ex test)                      ; => (nil T (ONE 2 3.0 "four"))
(eq (caddr (vlx-ex test)) test)    ; => nil
(equal (caddr (vlx-ex test)) test) ; => T
```

is executed in a document namespace then the results are as shown. Note that all functions exported by a VLX are automatically imported when loaded into a document namespace. (Also note that the forward slash, `/`, is AutoLISP notation for introducing local variables similar to `&aux` in Common Lisp.)

The envelope functions also set up an error handling context on entry to the VLX. Normally, when an error occurs in AutoLISP, an error message is generated, the `*error*` function is called with the message (a string) and then the stack is unwound and control returns to the read-eval-print-loop. The default error function simply prints the given message. The programmer can redefine `*error*` to handle user specified actions such as restoring global state. If an error occurs in the execution context of a VLX and the VLX namespace does *not* define `*error*`, then both the VLX stack and the document stack is unwound after the `*error*` function in the document namespace returns.

Additionally, the programmer can define `*error*` in VLX namespace. Now, if an error occurs inside the context of the VLX, this VLX error function is called with the system generated error string. The user defined VLX error function can either pass this string (or another string) to the document error function or it can return a value to the document continuation waiting to receive the result of the initial envelope entry into the VLX. In this case, only the portion of the stack representing the VLX execution context is unwound. In the previous case the document stack context is unwound also.

For example, if

```
(vl-doc-export 'vlx-er)
(setq *how* nil)
(defun vlx-er (how) (setq *how* how))
(setq *s1* 1)
(setq *s2* 2)
(vl-doc-export 'list-specials)
(defun list-specials () (list *s1* *s2*))
(vl-doc-export 'f0)
(defun f0 (x) (f1 x))
(defun f1 (*s1*) (f2 *s1*))
(defun f2 (*s2*) (list *s1* *s2* (/ 1 *s2*)))

(defun *error* (msg / specials)
  (setq specials (list *s1* *s2* msg))
  (cond ((eq *how* 'value)
         (vl-exit-with-value (cons 'value specials)))
        ((eq *how* 'error)
         (vl-exit-with-error (vl-princ-to-string
                               (cons 'error specials))))
        (t
         (print (cons t specials)))))
```

is compiled to a VLX named `vlx-er` and

```
(load "vlx-er.vlx")
(f0 1)            ; => (1 1 1)
(list-specials)   ; => (1 2)
(vlx-er 'value)
(setq a (f0 0))
a                 ; => (VALUE 0 0 "divide by zero")
(list-specials)   ; => (1 2)
(vlx-er 'error)
(setq a (f0 0))   ; *** ERROR: (ERROR 0 0 divide by zero)
a                 ; => (VALUE 0 0 "divide by zero")
(list-specials)   ; => (1 2)
(vlx-er nil)
(setq a (f0 0))   ; prints (T 0 0 "divide by zero")
a                 ; => "divide by zero"
(list-specials)   ; => (1 2)
```

is executed in a document namespace then one can see that `f0` returns a "value" list to document continuation or sends an error message to the document error function respectively. One can see that specials still retain their dynamic bindings in the VLX `*error*` function but that those bindings are undone properly. Note that if the VLX `*error*` function does not explicitly pass control to an error transfer form then the original error message is returned to the document continuation.

Document namespaces cannot export functions. A VLX can only access functions in its own namespace or functions in other VLX applications "loaded" into its namespace. However, it is possible for a VLX to reference and set the values of variables in the document namespace (although document namespace functions may not be invoked). This is done to give Visual LISP programmers the longstanding ability of ObjectARX application access to Lisp variables (used, for instance in the `CAL`culator ObjectARX application).

Given

```
(vl-doc-export 'dget)
(defun dget (var) (vl-doc-ref var))
(vl-doc-export 'dset)
(defun dset (var val) (vl-doc-set var val))
```

as `vlx-doc` then

```
(load "vlx-doc.vlx")
(setq a (list 'one 2 3.0))
(list (eq a (dget 'a)) (equal a (dget 'a))) ; => (nil T)
(setq b nil)
(dset 'b a)                                 ; => (one 2 3.0)
(list (eq a b) (equal a b))                 ; => (nil T)
```

results in the evaluations shown. Note that values are copied between the two namespaces.

Finally, ObjectARX applications may be loaded into document or VLX namespaces. Loading, in this case, means making their exported functions available in the namespace.

Visual LISP as an Active X Client

Visual LISP can serve as a controller for any application that exposes an `IDispatch` interface. This means that Visual LISP can control other Windows application (besides AutoCAD) such as Excel. It provides "bootstrap" functions to get or create an application object, including the application object for the current AutoCAD session. An application object typically provides "factory" methods such that other objects relevant to the given application may be created. For instance, the following code starts Excel, and prepares for entering data into the cells of a sheet in a workbook.

```
(setq *excel-cells* '())
(setq *row* nil)

(defun init-excel (/ excel-app wb-collection
                     workbook sheets sheet1)
  (setq excel-app (vlax-create-object "excel.application"))
  (setq wb-collection (vlax-get excel-app "Workbooks"))
  (setq workbook (vlax-invoke-method wb-collection "add"))
  (setq sheets (vlax-get workbook "Sheets"))
  (setq sheet1 (vlax-get-property sheets "item" 1))
  (setq *excel-cells* (vlax-get sheet1 "Cells"))
  (vlax-put excel-app "Visible" 1))

(defun write-row-column (row col x)
  (vlax-put-property
   *excel-cells* "Item" row col (vl-princ-to-string x)))
```

The next code gets the application object of the current AutoCAD session and uses that to obtain the modelspace of the current drawing document. Modelspace is a collection into which graphics objects may be placed.

```
(setq *model-space* nil)

(defun init-app ()
  (setq *model-space*
        (vla-get-modelspace
         (vla-get-activedocument
          (vlax-get-acad-object))))
  (write-row-column 1 1 "center x")
  (write-row-column 1 2 "center y")
  (write-row-column 1 3 "center z")
  (write-row-column 1 4 "radius")
  (setq *row* 2))
```

The methods and properties of an object may be accessed via generic Visual LISP functions (e.g., `vlax-invoke-method`) or through wrapper functions which may be automatically generated for a given application given its type library. The AutoCAD automation API is accessed via the wrapper functions which follow the convention of having `vla-` prepended to the names exposed in the type library. Thus, the `AddCircle` method of the `ModelSpace` class is available via the function `(vla-addcircle <modelspace-object> <center-point> <radius>)`.

The previous code wrote column headings in an Excel row. The following creates two concentric circles in modelspace and enters the center point and radius of each circle created into consecutive Excel rows.

```
(defun add-circle (px py pz radius)
  (write-row-column *row* 1 px)
  (write-row-column *row* 2 py)
  (write-row-column *row* 3 pz)
  (write-row-column *row* 4 radius)
  (setq *row* (+ *row* 1))
  (vla-addcircle *model-space*
                 (vlax-3d-point px py pz)
                 radius))

(defun demo ()
  (init-excel)
  (init-app)
  (add-circle 5 5 0 5)
  (add-circle 5 5 0 10))
```

Arguments to ActiveX methods are either simple types such as integers, doubles, booleans or strings, or aggregate types, such as safearrays, encapsulated in variants. Visual LISP provides an API to create safearrays and variants, to query or change their types, to set and get their values, and, in the case of safearrays, to query for dimension information.

For example, the code above used the `vlax-3d-point` utility function to create a variant containing a safearray of doubles. This can be seen in more detail as

```
(setq p (vlax-3d-point 2 3 4))  ; => #<variant 8197 ...>
(setq v (vlax-variant-value p)) ; => #<safearray...>
```

```
(vlax-safearray-get-element v 0); => 2.0
(vlax-safearray-get-element v 1); => 3.0
```

Visual LISP as an ActiveX Server

Visual LISP can function as an ActiveX server. This is useful when using VBA dialogs to control Visual LISP applications. For instance, a programmer can create dialogs using the VBA visual tools for layout. They can then have the actions invoke Visual LISP.

Any Visual LISP function may be invoked and any symbol may have its value referenced or set. Parameters and results are all variants. In the following example

```
Sub Hello()
    Set vla = CreateObject("VL.Application.1")
    Set vld = vla.ActiveDocument
    Set vl_read = vld.Functions.Item("read")
    Set vl_eval = vld.Functions.Item("eval")
    Set vl_hello = vl_read.funcall("(defun hello (x)(print x))")
    Set vl_hello = vl_eval.funcall(vl_hello)
    Set vl_hello = vld.Functions.Item("hello")
    ret = vl_hello.funcall("world")
    MsgBox ret, vbOKOnly
End Sub
```

we obtain access to Visual LISP's `read` and `eval` functions. We then use those to define a `hello` function, which we then call, displaying its return value in a message box.

As mentioned earlier, dialogs created with VBA may be delivered in the same file as the Visual LISP application by including that DVB file in a VLX application. The VBA code can then be accessed and launched from within the running application via

```
(vl-doc-export 'vba)
(defun vba ()
  (vl-vbarun "hello.dvb!Hello"))
```

Reactors and Lisp Data

ObjectARX "reactors" allow programmers to specify user callback functions to be invoked upon certain AutoCAD events such as drawing opening, entity creation or modification, etc. Visual LISP defines an API by which programmers can define reactors, associate them with AutoCAD events, specify Lisp callback functions, and specify data (i.e., cookies) to be passed to the callback functions when an event occurs. If the reactors are made persistent then they are saved with the drawing when it is written to a file.

In this example

```
(defun reactor-callback (notifier reactor other)
  (alert (vl-princ-to-string (list 'notifier notifier
                                   'reactor  reactor
                                   'other    other))))

(defun demo (cookie / line)
  (setq line (vla-addLine *model-space*
                          (vlax-3d-point '(1.0 1.0 0.0))
                          (vlax-3d-point '(4.0 4.0 0.0))))
```

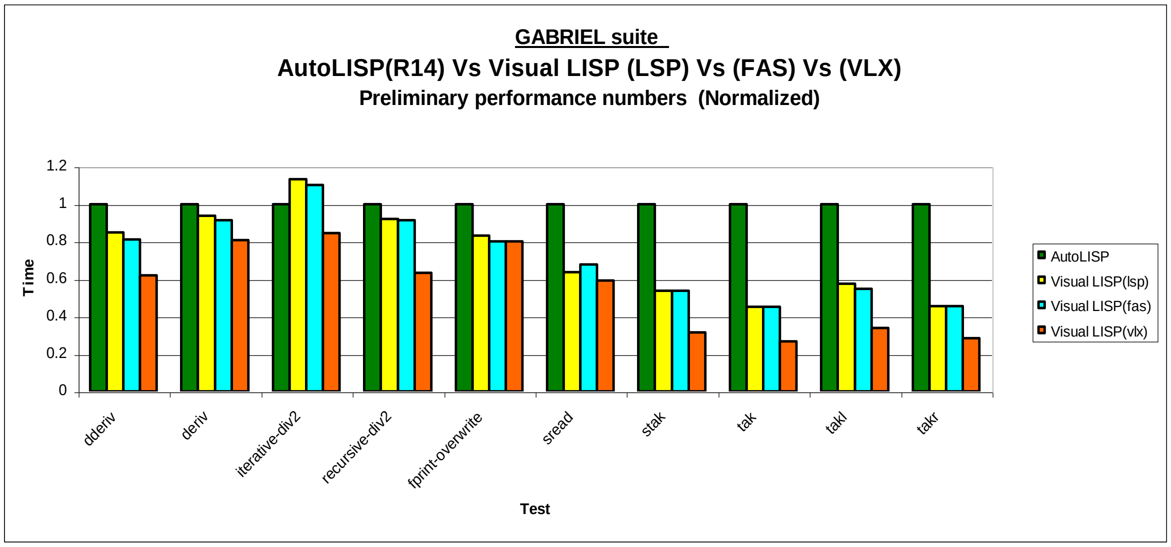
```
   (vlr-object-reactor
    (list line)
    cookie
    '((:vlr-modified . reactor-callback)))))

(demo "oreo")
```

a line is created then an object reactor is created to watch any modifications to that line. When a modification occurs the callback function will be called with the object which has been modified (notifier), the reactor object set up for watching (the one created in the call to `vlr-object-reactor`) and the Lisp data attached to the reactor at creation.

Similarly, programmers may attach arbitrary Lisp data to drawing or drawing entities. If Lisp data and/or persistent reactors are created by a VLX application then, when that drawing is loaded later, AutoCAD will attempt to automatically load the VLX application responsible for the data and/or reactors.

Performance

Visual LISP's design goals centered on ease-of-use and AutoLISP compatibility. However, Visual Lisp's VM does provide performance improvements compared to the R14 AutoLISP interpreter as shown in the following chart.



Migration Issues

Visual LISP approaches 100 percent AutoLISP compatibility. However, there are a few interesting incompatibilities. In AutoLISP, function definitions were stored and available to the programmer as lists. For instance, a definition

```
(defun foo (x)
  (+ x 1))
```

causes `foo` to evaluate to

```
((x) (+ x 1))
```

AutoLISP programmers have taken advantage of this implementation by augmenting function definitions at run time via code like:

```
(defun f (x)
  (print (list 'f x)))

(defun another-f (x)
  (print (list 'another-f x)))

(setq f (append f (cdr another-f)))
```

Since Visual LISP compiles all function definitions the value of the functions defined above will be an opaque type representing a compiled function. Therefore the call to `append` will fail.

Since this idiom was used so often, Visual LISP provides a `defun-q` form which still compiles the definition but allows access to the AutoLISP list representation of the definition.

AutoLISP had an evaluation model in which the function position of an application was evaluated first. The rest of the arguments were possibly evaluated from left-to-right depending on the result of the function position evaluation. The function evaluation could result in the value of one of AutoLISP's special forms (e.g., `setq`, `if`, etc.). In this case the rest of the arguments where evaluated according to the rules of the given special form. Therefore

```
((if (test) setq +) a 3)
```

would evaluate to and set `a` to `3` if `(test)` evaluated to non-`nil`. Otherwise it would evaluate to the result of adding the value of `a` (assuming it to be a number) to `3`.

Visual LISP first evaluates the arguments left-to-right then evaluates the function position, which can only be a variable whose value is a function or a `lambda` expression. So the above example would fail AutoLISP semantics on two accounts. First, `a` would be evaluated to its value, even if the function position results in `setq`. Second, if the function position evaluates to a special form, an error is raised.

Visual LISP does not attempt compatibility in this regard. A migration guide points this difference out along with others such as the inability to change the value of the variable `T` or integer rollover (AutoLISP and Visual LISP do not support bignums) edge conditions being different by one on the negative side.

Future Directions

We plan to advance Visual LISP along several fronts: the language itself, its relation to other languages, and the ability to implement custom objects in Lisp.

Visual LISP supports all the AutoLISP language functions and adds a number of functions derived from Common Lisp, such as `member-if`, `princ-to-string`, etc. We plan to enrich Visual LISP with types, syntax and functions from the Scheme dialect of Lisp. Scheme is a good fit with AutoLISP since they are both Lisp1 dialects and relatively small languages.

Java and COM have a thriving 3rd-party component market. We would like to leverage this market by providing the ability to use JavaBeans and/or COM components in the Visual LISP environment. This could be done by providing a JVM in AutoCAD and establishing a bridge between the Lisp VM and the JVM. With the presence of a JVM it would perhaps be possible to compile Visual LISP to JVM byte-code [4] making it possible to replace the Visual LISP VM with a JVM and further leverage work in Java Just-In-Time compilers. This would also allow components to be written in Lisp.

Although AutoCAD exposes its object model through its Automation API, this API only allows instantiations of given types using either VBA or Visual LISP. It is not possible to write custom objects which inherit and extend existing AutoCAD classes in either VBA or Visual LISP. This is only possible, at this time, in C++ via ObjectARX. In the future we plan to allows custom objects to be written in any of AutoCAD's application programming languages.

Acknowledgments

The authors would like to acknowledge Peter Petrov and Serguei Volkov as key architects and implementers of Visual LISP, Perceptual Engineering Inc. for their role in advancing Visual LISP in AutoCAD, and the entire Visual LISP team for making it a reality.

References

[1] John Walker. The Autodesk File, internal memo.
[2] Art Little, “Hot Tip Harry's Favorite 200 Lisp Routines for AutoCAD” Trade Paperback, 1997
[3] Richard Gabriel, Kent Piton. Technical Issues of Separation in Function Cells and Value Cells. In *Lisp and Symbolic Computation*, 1(1):81-101
[4] Per Bother. Compiling Dynamic Languages to the Java VM. In Proceeding of the 1998 Usenet Conference, New Orleans, June 1998.