Eval

Glen Diener
Center for Computer Research in Music and Acoustics (CCRMA)
Dept. of Music
Stanford University
grd@ccrma.stanford.edu

Eval is a service provider for any app that writes ascii to the pasteboard. It can do two things: compile, load, and execute the selection as ObjC code, or interpret the selection as postScript. Eval manages a window, called the

"Transcript", with both a text view and a graphics view.
ObjC output or postscrip output can be directed to these windows.

Installing

Eval runs as a "service". To install it, copy the Eval.app directory into ~/Apps or /LocalApps (you must be be root to install in /LocalApps). Then logout and log back in again. The "Eval" option should now appear in the services menus of Edit, Terminal, Librarian, WriteNow, and many other Apps. Any application which can print ascii text to the pasteboard should show the "Eval" .

If Eval does not appear, try deleting the files ".applit" and ".cache" in ~/NeXT/services, then log back out and log back in. This will regenerate these files automatically.

Special note: If you remake the binary, and want to strip the executable, use strip -S. If you use strip without the -S option, Eval will not function correctly.

Running Eval

Eval is run by selecting text in your application, then selecting either the "ObjC" or the "PostScript" options in Eval's services menu. Assuming you have installed Eval, then try working through the following examples. In might be useful to "tear off" the Eval services menu to make things more convenient.

Select the following line of ObjC code, then execute the menu item "ObjC"

```
[Eval printf: "Hello world\n"] ;
```

You should see Eval's transcript window pop up, with the phrase "Hello World" written into the lower view.

Let's examine the above code."Eval" , the message receiver, is the name of a "prelinked" class which understands a number of useful class messages. One of these, printf:, works exactly like good old printf(), except that it always writes to the transcript's text view, concatenating its output to the contents of that view. You can clear the text in Eval's text view by pressing the "Clear Text" button. Clear it now, then evaluate the following ObjC code to see how printf: accepts a variable number of arguments, just as printf() does...

```
int i, j = 0 ;
```

```
for(i = 1 ; i <= 100 ; i++)
j += i ;
[Eval printf: "%s %d\n","The answer is ", j] ;
```

If you need to #import anything into the code, no problem. Just be sure you leave at least one blank line between the #import (#define, whatever), and the rest of your code (more on this later). Evaluate the following ObjC....

```
#import <sys/dir.h>
// Print a listing of the root directory.
// Note: 1 or more blank lines seperate #defines from code

int i,fileCount ;
    struct direct **filesList ;
    char *aPath = "/" ;
```

```
        fileCount = scandir(aPath,&filesList,NULL,NULL) ;
    for(i = 0 ; i < fileCount ; i++)
            [Eval    printf: "%s\n", (*filesList[i]).d_name] ;
```

Eval's upper view is used for displaying postscript graphics output. Select the following, then choose Eval's PostScript menu item...

```
 /Times-Roman findfont
20 scalefont setfont
10 10 moveto
(Hello world) show
```

You can see how Eval functions like a ready-to-use version of Yap.
Unlike Yap, however, Eval let's you generate postscript via ObjC, using Eval's ps: class message. Clear the graphics (using the Clear Graphics button), then evaluate the following as ObjC...

```
[Eval ps:
  "/Helvetica findfont "
  "50 scalefont setfont "
  "10 10 moveto "
  "(c'est quoi ca?) show "
] ;
```

As shown in the following example, Eval's ps: method can accept arguments,

just like its printf: method. The example also shows Eval's clearGraphics class method, which accomplishes programmatically what hitting the Clear Graphics button accomplushes manually.

```
int i ;
for(i = 1 ; i < 10 ; i++)
{ [Eval clearGraphics] ;
   [Eval ps:
     "/Helvetica findfont "
     "%d scalefont setfont "
     "%d %d   moveto "
     "(Hello world) show ",
      i * 10, i * 5, i * 10
   ] ;
```

```
    NXPing() ;
    usleep(500000) ; // sleep for half a second
}
```

If "Hello world" has become too big and is no longer completely visible in the Graphics view, use the scrollers to view it all (or resize the window!).

The safest way to do postscript is to use either Eval's ps: method or to just evaluate straight postscript using the Services/Eval/PostScript item. In both cases, the postscript sent to the window server is wrapped in error-handling code to catch errors, which will be reported in the text view. Both of these methods also wrap the postscript in a gsave/grestore pair, so if you want to build up state between calls to these methods, you can't use these methods.

The alternative is to braket ObjC code within [Eval startPS] and [Eval stopPS] messages. startPS takes care of "locking focus" on the graphics view's underlying NXImage. stopPS unlocks it composites the image to the graphics view, and flushes the window. The following example illustrates this technique...

```
#import <math.h>
#import <dpsclient/psops.h>

float x ;
#define PI 3.14159
[Eval clearGraphics] ;
[Eval ps:
 "0 100 moveto 250 100 lineto stroke"] ;
```

```
[Eval startPS] ;
PSmoveto(0.0,100.0) ;
for(x = 0.0 ; x <= 4.0 * PI ; x += 0.05)
   PSlineto(x * 20.0, sin(x) * 100.0 + 100.0) ;
 PSstroke() ;
[Eval stopPS] ;
```

How it works

When you select code and evaluate it using the ObjC button, the Eval app sandwiches your code with both a header and a footer. It creates a complete objC class definition with exactly one class method, called "doit". It then saves the definition in a temporary file in /tmp, invokes the objC compiler, then uses the objc_loadmodules call to dynamically link the result into the Eval app itself. Finally, the newly linked class is sent the "doit" message,

and your code is executed. When the message returns, all the temporary files are removed.

Actually, Eval attempts to cut your code into two sections, a "header" section, and a body section. The "header" section is placed at the beginning of the file Eval creates, and is not tectually part of the "doit" method. For more prescise details about how this works, see the file "Eval.m", and, in particular, the method

```
- evalObjC:(id) pboard userData: (const char *) uBuf error: (char **) msg ;
```

You can customize the compiler options using Eval's Preferences Panel. The panel also lets you add or remove link libraries searched during the dynamic linking. The transcript's text view's font and font size can be saved, and the size of the graphics view, in pixels, can also me specified here.

Eval Class Methods

Definitions of the following class methods are already present in the file which Eval creates, so the code you use can simply call them directly without having to #import any headers.

+ clearGraphics ;

    Clears the transcript's graphics view.

+ clearText ;

Clears the transcript's text view.

+ printf: (char *) format, ... ;

Formats and prints text to the transcript.

+ ps: (char *) format, ... ;

"Locks focus" on the transcript's graphic's view's NXImage object, formats and prints text to the postscript window server, wrapped inside a NXHandler and bracketed by a gsave/grestore pair., then unlocks focus, composites the NXImage to the graphics view, and flushes the offscreen window.

+ startPS ;

"Locks focus" on the transcript's graphic's view's NXImage object.
Must be followed, eventually, by a stopPS message. Any PostScript
rendered after startPS will show up in the graphics view after stopPS
is sent.

+ stopPS ;

"Unlocks focus" on the transcript's graphic's view's NXImage object,
then composites the image to the graphics view and flushes the window.

Updated Thursday, Sept. 17th 1992 for 3.0