

Callbacks in plain vanilla AppleScript

Gregory Lo <gregorylo@sympatico.ca>

Abstract

A common technique for creating generic subroutine libraries that can be adapted for use in different situations is to provide a mechanism to "call back" to the caller; however, it is not immediately obvious how this can be done with AppleScript. This paper discusses a simple technique for implementing a generic callback mechanism in plain vanilla AppleScript. Specifically, the technique allows one subroutine to invoke another specified in a variable, rather than one whose name is known when the subroutine was written.

Introduction

Reusability is very fashionable in the craft of programming software. Software developers develop and refine useful code and save them away for later reuse: code can be factored into reusable *subroutines*, and subroutines can be collected into *libraries*. Many programming languages do provide a library mechanism for managing and using the saved code, but this can be considered simple automation of the more general concept. We can define *library code* to be any code that has been stored away for the purpose of future reuse, regardless of the form of the repository in which it is stored or the manner in which it is retrieved — this includes code remembered in human memory, or a notebook or text file from which one can copy-and-paste code.

In order to reduce the amount of work required to integrate library code into the system of a software program, library code is often designed to be *generic*. Ideally, the code is written in such a way that the number of situations in which it will function as intended (or at least as promised) is maximized. The separation between the generic library code and the application of it means that it can be developed and maintained independently of any particular application.

Generic code requires parameters to be set to be of any use. This is the mechanism to "bind" the code to the situation in which it is to be used — which is a fancy way of saying that the code is customized for a particular application by giving it information. These parameters can be simple values or actions.

Callbacks in general

One technique for using an action as a parameter of generic code is to provide a *callback*. A callback is a *reference* to some other subroutine, and the term "callback" itself is used to help explain the purpose of taking the reference to a subroutine — the specified subroutine will be "called back" at some future time.

In many programming languages, the machine address to a subroutine's image in memory (also known as a *function pointer*) is used as the reference to a subroutine. The term "pointer" has the connotation of a machine address; the term "reference" is more general. Where one talks of function pointers, a reference is implied. This can help one approach literature and discussion of callbacks in programming languages where the "function pointer" is the more natural usage, and apply the knowledge to AppleScript (which has no capability of addressing machine memory).

AppleScript subroutines

Subroutines in AppleScript are actually handlers of script objects. Subroutine calls are analogous to passing messages to script objects; an object responds to a message by invoking the handling subroutine — a “handler” — indicated in the message. In AppleScript lingo, “handler” is synonymous with the term “subroutine.”

```
on SomeProcedure(x)
    display dialog "Hello, " & x
end SomeProcedure

my SomeProcedure("World!")
-- 'SomeProcedure' message is
-- sent to our self
```

Listing 1. Subroutine call is analogous to message passing

The executing script is itself also a script object with handlers of its own. In Listing 1, the use of the 'my' keyword causes the 'SomeProcedure' message to be sent to our self.

Handlers are an intrinsic type in AppleScript. They have their own class ('hand'), and references to specific handlers can be stored away in variables.

```
script Object1
    on Proc()
        display dialog "Object 1!"
    end Proc
end script

set x to Object1's Proc
-- x is «handler Proc»
-- class of x is 'hand'
```

Listing 2. We can store a reference to a script object's handler in a variable

Trying to call subroutines indirectly

References to handlers can be stored away in variables, but the method of

invoking this stored handler reference is not immediately apparent. Intuitively, we assume that using a variable's name wherever a value is expected should cause the variable's value to be retrieved (or substituted).

```
script Object1
    on Proc()
        display dialog "Right!"
    end Proc
end script

set y to 23
-- y now has value of 23
set z to y
-- z now has value 23

set x to Object1's Proc
x()
-- "Right!"
```

Listing 3. The value of y was used in the assignment to z, and the handler referenced by x was invoked successfully.

Listing 3 shows us that it is possible for AppleScript to obtain the handler to be called from the value of a variable. This is the result we expected, but we were invoking the handler ourselves, and we are ultimately interested in passing the handler reference to other pieces of code — other handlers — in the hopes that we will be called back in some other context. Let's try another experiment to see if this can be done.

```
script Object1
    on Proc()
        display dialog "Right!"
    end Proc
end script

on CallHandler (msg)
    msg();
end CallHandler

set x to Object1's Proc
my CallHandler (x)
-- «script» didn't understand
-- the msg message
-- ERROR!
```

Listing 4. Something didn't work right.

From listing 4, we can see that calling a handler is a special case for AppleScript. AppleScript insists upon treating the name of the handler as a literal value. Instead of taking the value of the parameter variable `msg` and using that for the name of the handler to call, AppleScript tried to call the handler named `msg` in the executing script! This is not what we want to happen if we are to pass the handler reference to another piece of code and expect to be called back. An even worse scenario is if there actually was a handler named `msg`; AppleScript would not complain, but the wrong handler would be invoked — the script's `msg` instead of `Object1's Proc` (the handler we had intended to be called).

```
script Object1
  on Proc()
    display dialog "Right!"
  end Proc
end script

on msg()
  display dialog "Wrong!"
end msg

on CallHandler (msg)
  msg();
end CallHandler

set x to Object1's Proc
my CallHandler (x)
-- "Wrong!"
-- No error from AppleScript
-- But, we didn't want this!
```

Listing 5. AppleScript tried to call a handler of Object1 named msg.

Although it looks like AppleScript has led us to a dead end (it won't let us use the handler reference we've stored away in any useful way), we can't give up just yet. There is another way to get AppleScript to do what we want... but, it involves a trick.

Here's where it gets tricky

Experimentation shows that AppleScript can be fooled into taking

the value of a variable for use as a handler reference... if the variable is explicitly declared as a global. Armed with this information, let's try again.

```
script Object1
  on Proc()
    display dialog "Right!"
  end Proc
end script

on msg()
  display dialog "Wrong!"
end msg

on CallHandler(msg)
  global theHandler
  set theHandler to msg
  theHandler()
end CallHandler

set x to Object1's Proc
my CallHandler (x)
-- "Right!"
-- Success!
```

Listing 6. A successful call to the handler "pointed to" (or, referenced) by x.

Callbacks and AppleScript lists

The use of callbacks seems ideal for handling lists of items. We can combine iteration of lists and callbacks to construct simple routines that will form the basis of a list-processing toolkit. Simplicity and modularity are desirable because they help to make library code maintainable and robust — the less any individual subroutine has to do, the more likely that it will perform as expected.

```
to ForEach(fn, l)
  global ForEach_fn_
  repeat with x in l
    set ForEach_fn_ to fn
    ForEach_fn_(x)
  end repeat
end ForEach
```

Listing 7. An extremely simple, reusable subroutine that will "visit" each and every item in a list.

The subroutine in listing 7 is extremely simple, and can be reused as-is for a variety of applications. Its one and only function is to visit each item in a list, and it is reasonable to assume that it will do exactly that when called. However, when it is applied, the behaviour must be customized to suit that application in particular. An example of this is shown in listing 8, below.

```
to VisitItem(x)
    display dialog (x as string)
end VisitItem

set L to { 1, "Hello", 3 }

ForEach( my VisitItem, L )
-- displays a dialog for
-- every item in list L
```

Listing 8. Visiting each item in a list.

Another useful tool we can put into our list processing toolkit is a subroutine that will “transform” a list. We can build on the simplicity of the ForEach() subroutine described above, and create a variant that will not only visit each item in a list, but creates a new list from the results. One implementation of just such a subroutine is described below in listing 9.

```
to Map(fn, l)
    global Map_fn_
    set y to {}
    repeat with x in l
        set Map_fn_ to fn
        set end of y to Map_fn_(x)
    end repeat
    return y
end Map
```

Listing 9. This subroutine results in a transformed version of the given list.

```
to MakeCell(x)
    return "<TD>" & x & "</TD>"
end MakeCell

set L to { ↵
    "Simpson", ↵
    "Hohmur", ↵
    "555-1212" }
```

```
set R to Map( my MakeCell, L )
-- R is now composed of:
-- "<TD>Simpson</TD>"
-- "<TD>Hohmur</TD>"
-- "<TD>555-1212</TD>"

to Itemize(x)
    return "Item: " &
end Itemize

set R to Map( ↵
    my MakeCell, ↵
    Map( my Itemize, ↵
        { 1, "Hello", 4 } ) )
-- R is now composed of:
-- "<TD>Item: 1</TD>"
-- "<TD>Item: Hello</TD>"
-- "<TD>Item: 4</TD>"
```

Listing 10. Applying a transformation to a list by visiting each of its items.

From the examples in listing 10, we can see that simple subroutines can be combined to perform more complex operations. The output of one function can be fed as input into another function. If we can define a set of basic operations (our toolkit) that are robust, we can combine them to write other subroutines (or subroutine calls) that are more powerful, yet still simple. Again, callbacks are used to customize these more powerful subroutines for specific applications.

Closely related to the Map() subroutine is another useful callback-related subroutine that filters items in a list. The subroutine takes a special kind of callback called a *predicate* — for our purposes, a predicate is a callback which says something about an item that is either true or false. Like an intelligent sieve, the filter blocks some items and allows others to pass through. The result is another list which is a subset of the original list, and whose items satisfy the filter.

```
to Filter(fn, l)
    global Filter_fn_
    set y to {}
    repeat with x in l
```

```

    set Filter_fn_ to fn
    if Filter_fn_(x) false then
        set end of y to x as item
    end if
end repeat
return y
end Filter

```

Listing 11. Here, a callback is used to determine the composition of a new list.

```

to IsEven(x)
    if x mod 2 = 0 then
        return true
    end if
    return false
end IsEven

to MakeID(x)
    return "ID: " & x
end MakeID

set L to { 10, 2, 3, 45, 142 }

set R to Filter( my IsEven, L )
-- R is { 10, 2, 142 }

set R to Map( ↵
    my MakeID,
    Filter( my IsEven, L ) )
-- R is now composed of:
-- "ID: 10"
-- "ID: 2"
-- "ID: 142"

```

Listing 12. Odd numbers have been filtered out from a list.

We need to look at one more primitive subroutine at least. This one takes a list and visits each item with a callback, just as Map() does, but it is slightly more powerful in that its result does not have to be a list. Listing 13 shows one implementation of this subroutine.

```

to FoldLeft(fn, a, l)
    global FoldLeft_fn_
    set y to a
    repeat with x in l
        set FoldLeft_fn_ to fn
        set y to FoldLeft_fn_(y,x)
    end repeat
    return y
end FoldLeft

```

Listing 13. This subroutine visits every item a list, but the result can be of any value type

The callback used in FoldLeft() takes two arguments (it is *dyadic*), instead of the single argument (*monadic*) callbacks required by ForEach(), Map(), and Filter(). Not only is every item in the input list visited, but also the result of the visit to the previous item is passed along to the callback.

To see how useful FoldLeft() is, we can try a few examples. Listing 14 shows that summation of a list of numbers can be simply defined with the FoldLeft() subroutine and a callback that adds two numbers. With this definition, we can also build a subroutine that composes other subroutines to give the sum of squares. Here, we also simplify our subroutines further by taking advantage of the fact that the last expression in an AppleScript handler is used to define its result.

```

to Add(a,b)
    a + b
end Add

to Sum(l)
    FoldLeft( my Add, 0, l )
end Sum

to Square(x)
    x * x
end Square

to ShiftAdd(a,b)
    (2 * a) + b
end ShiftAdd

to SumSquares(l)
    my Sum( Map( my Square, l ) )
end SumSquares

to BinValue(l)
    FoldLeft( my ShiftAdd, 0, l )
end BinValue

set s to Sum( { 1, 2, 3, 4 } )
-- s has the value 10
-- 1 + 2 + 3 + 4 = 10

set ss to SumSquares( { 1, 2 } )
-- ss has the value 5

```

```
-- 1 * 1 + 2 * 2 = 1 + 4 = 5

set b to BinValue( { 1, 0, 1 } )
-- b has the value 5
-- 2*(2*(2*(2*0)+1)+0)+1
-- = 2*(2*(0+1)+0)+1
-- = 2*(2*1)+1 = 4 + 1 = 5
```

Listing 14. Composing subroutine calls

Notice that a reading of the code of the `SumSquares()` subroutine is very close to the plain English definition of its operation — the sum of squares is the summation of a list of squared items. The simplicity of the code makes it easy to understand and maintain.

It doesn't take too long to discover that the `ForEach()`, `Map()`, and `Filter()` subroutines can be implemented as a combination of the `FoldLeft()` and helper subroutines. Listing 15 defines possible implementations of such alternatives. While our existing definitions of `ForEach()`, `Map()`, and `Filter()` are perfectly adequate (and arguably more efficient in practice), the example does reinforce the point that general subroutines like `FoldLeft()` can easily be adapted to perform more specific functions through the use of callbacks. The same `FoldLeft()` subroutine used in listing 15 to define `MapX()` was used in the examples of listing 14 to sum a list of numbers.

```
to Call1(a,b)
  global Call1_fn_
  set Call1_fn_ to a
  Call1_fn_(b)
end Call1

to ForEachX(fn, l)
  FoldLeft( my Call1, fn, l )
end ForEachX

to MapX_helper(a,b)
  set fn to f of a
  set bb to my Call1(fn,b)
  set dl to data of a
  { f:fn, data:dl & bb }
end MapX_helper

to MapX(fn, l)
```

```
data of ↵
  FoldLeft( ↵
    my MapX_helper, ↵
    { f:fn, data:{} }, l )
end MapX

to FilterX_helper(a,b)
  set fn to f of a
  if my Call1(fn,b) true then
    return a
  end if
  set dl to data of a
  { f:fn, data:dl & b }
end FilterX_helper

to FilterX(fn, l)
  data of ↵
  FoldLeft( ↵
    my FilterX_helper, ↵
    { f:fn, data:{} }, l )
end FilterX
```

Listing 15. Alternate definitions of `ForEach()`, `Map()`, and `Filter()`

Some examples in the form of droplets

To begin drawing our attention back towards more practical uses of callbacks, Listing 16 displays part of an example AppleScript droplet application. The logic for iterating over the list of filesystem objects dropped onto the droplet is hidden away in the `Filter()` subroutine call, which can be used and reused as needed — the same code provides infrastructure for the filtering operation in all calls. In the example, the callback used to filter the list is determined by the day of the week the droplet is activated.

```
on open(fl)
  set myDate to current date
  set myDay to weekday of myDate
  if myDay is Sunday then
    set theProc to my hasAA
  else
    set theProc to my hasBB
  end if
  set R to Filter( theProc, fl )
  choose from list R
end open

to hasAA(x)
```

```

set y to info for x
if folder of y is true then
    return false
end if
set ny to name of y
if ny contains "AA" then
    return true
end if
return false
end hasAA

to hasBB(x)
set y to info for x
if folder of y is true then
    return false
end if
set ny to name of y
if ny contains "BB" then
    return true
end if
return false
end hasBB

```

Listing 16. An example AppleScript droplet application

Even with the minimal set of callback-related tools we have already defined, we can start to write more useful scripts. The example droplet application shown in listing 17 assembles a hierarchical list of the files and folders dropped onto it, into a web page, and then places the result onto the clipboard.

```

property G: load script libraryloc
-- where libraryloc is the
-- location of a library script
-- with our list processing
-- subroutines

```

```

global oo

on open(fl)
    set oo to "
<html>
<head>
</head>
<body>
<ul>
"
    G's ForEach(my ProcessItem,
fl)
    set oo to oo & "
</ul>
</body>
</html>

```

```

"
    set the clipboard to oo
end open

to MakeSubitem(a, b)
set a1 to folder of a
set a2 to data of a
set x2 to (a1 as string) & b
return { ¬
    folder:a1, ¬
    data:a2 & (x2 as alias) }
end MakeSubitem

to AddItem(x)
try
set y to info for x
set ny to name of y
set oo to oo & "<li>" & ny
if folder of y is true then
set sl to data of ¬
    G's FoldLeft( ¬
        my MakeSubitem, { ¬
            folder:x, ¬
            data:{} }, ¬
            list folder x )
set oo to oo & "
<ul>
"
    G's ForEach( ¬
        my AddItem, sl)
    set oo to oo & "</ul>"
end if
set oo to oo & return
on error
-- ignore errors
end try
end AddItem

```

Listing 17. A droplet that places an HTML listing onto the clipboard of all files and folders dropped onto it

Conclusion

Code that is generic can be reused and maintained independently of application-specific code. Libraries of generic code whose robustness is certain can be assembled, making reuse of robust code easy; callbacks can be used to adapt the generic code to specific applications. AppleScript has facilities for creating libraries of generic code, and (with a little bit of help) for using callbacks to customize behaviour as desired.

Further reading

[0] Function pointers and callbacks (in C and C++)

<<http://www.function-pointer.org/>>

[1] Callbacks in C++ using template functors

<<http://www.tutok.sk/fastgl/callback.html>>

[2] The AppleScript Language Guide

<<http://developer.apple.com/techpubs/macosx/Carbon/interapplicationcomm/AppleScript/AppleScriptLangGuide/index.html>>

Some useful subroutines for processing lists with callbacks

Basic implementation

```
to ForEach(fn, l)
    global ForEach_fn_
    repeat with x in l
        set ForEach_fn_ to fn
        ForEach_fn_(x)
    end repeat
end ForEach

to Map(fn, l)
    global Map_fn_
    set y to {}
    repeat with x in l
        set Map_fn_ to fn
        set end of y to Map_fn_(x)
    end repeat
    return y
end Map

to Filter(fn, l)
    global Filter_fn_
    set y to {}
    repeat with x in l
        set Filter_fn_ to fn
        if Filter_fn_(x) false then
            set end of y to x
        end if
    end repeat
    return y
end Filter

to FoldLeft(fn, a, l)
    global FoldLeft_fn_
    set y to a
    repeat with x in l
        set FoldLeft_fn_ to fn
        set y to FoldLeft_fn_(y,x)
    end repeat
    return y
end FoldLeft
```

An alternative implementation

```
to Call1(fn,x)
    global Call1_fn_
    set Call1_fn_ to fn
    Call1_fn_(x)
end Call1

to Call2(fn,x1,x2)
    global Call1_fn_
    set Call2_fn_ to fn
    Call2_fn_(x1,x2)
end Call2

to FoldLeft(fn, a, l)
    set y to a
    repeat with x in l
        set y to my Call2( y, x )
    end repeat
    y
end FoldLeft

to ForEach(fn, l)
    FoldLeft( my Call1, fn, l )
end ForEach

to Map_helper(a,b)
    set fn to f of a
    set bb to my Call1(fn,b)
    set dl to data of a
    { f:fn, data:dl & bb }
end Map_helper

to Map(fn, l)
    data of ¬
        FoldLeft( ¬
            my Map_helper, ¬
            { f:fn, data:{} }, l )
end Map

to Filter_helper(a,b)
    set fn to f of a
    if my Call1(fn,b) true then
        return a
    end if
    set dl to data of a
    { f:fn, data:dl & b }
end Filter_helper

to Filter(fn, l)
    data of ¬
        FoldLeft( ¬
            my Filter_helper, ¬
            { f:fn, data:{} }, l )
end Filter
```