developerWorks®

Higher order functions

Using functions for such higher order purposes as arguments, function-generating functions, and anonymous functions

Jonathan Bartlett

March 31, 2005

Functions are the wonderful and powerful building blocks of computer programs. Functions allow you to break code down into simpler, more manageable steps. They also allow you to break programs into reusable parts -- parts that are both reusable within the program and in other programs as well. In this article, learn how to create new functions at runtime based on templates, how to create functions that are configurable at runtime using function parameters, and how the Scheme language can be a valuable tool with functions.

In computer programs, functions form the powerful building blocks that allow developers to break code down into simple, more easily managed steps, as well as let programmers break programs into reusable parts. As a nod to the wonderful function, in this article I'll demonstrate how to craft new template-based functions at runtime and explain how to build functions that can be configured at runtime using function parameters.

Examples in this article will use the Scheme programming language and C. An introduction to the Scheme programming language can be found in "Better programming through effective list handling" (developerWorks, January 2005) (see Related topics; links to other Scheme introductions and Scheme function references are provided here too).

Let's jump right in and start by creating some functions.

Creating anonymous functions

In Scheme, functions are created by default without names. The *lambda* special form creates a nameless function and then returns the value to the enclosing form or function call. The enclosing form may:

- Set a symbol to refer to the value (thus giving it a name).
- Store the value in a data structure for later use.
- Pass the value as a parameter to a function.

In most programming languages, the act of defining functions and naming them occur simultaneously. That these operations are separate actions in Scheme leads to a lot of confusion to new Scheme programmers. The Scheme method is actually very simple though, because Scheme treats functions just as it would other values.

Functions are created with `lambda`. Just like other values, these functions can be:

- passed as arguments,
- stored into variables,
- stored as a part of a larger data structure.

To make a Scheme function created with `lambda` act like a function from other languages, you need only to store it in a global variable -- this makes the function visible to other functions and accessible by name. Listing 1 shows an example of an anonymous function in Scheme that squares the number given to it.

Listing 1. A nameless function

```
(lambda (x)
   (* x x)
)
```

This code defines a function with one formal parameter, `x`; the function squares the parameter and returns the value. Remember, Scheme doesn't need explicit return values. Scheme simply returns the result of the last form evaluated in the function.

Let's continue by giving the function a name as in Listing 2.

Listing 2. Naming a function

```
(define square
   (lambda (x)
      (* x x)))

(display (square 3))
(newline)
(display (square 4))
(newline)
```

This is the simplest and most common way of dealing with functions -- giving them a name and then using them in computation later on. However, there is no rule in Scheme that forces you to give a function a name before using it.

In Scheme, the head of a list in a program must result in a function or a special form, but that doesn't mean it has to be the *name* of a function. Because the `lambda` special form returns functions, you can actually use a `lambda` function definition directly in a function call.

For example, instead of writing `(square 4)`, you can also write `( (lambda (x) (* x x)) 4)`. This is a two-element list in which the first element is the definition of the function and the second element is a number. The head of this list is the function definition itself. When it runs, it first evaluates everything within the list. The `lambda` form evaluates to a function definition which, because it is the head of the list, is called on the second element. The return value of this form is the square of the number 4.

Defining and calling functions immediately rather than naming them or passing them as values is an interesting concept, but not extremely useful. However, it does illustrate the idea of functions as values and shows how anonymous functions work.

Now let's look at functions as function arguments.

Functions as function arguments

A practical use of nameless functions is that of building functions to pass as arguments to other functions. The best example of this is the Scheme built-in function `map`.

The `map` function takes two arguments: a function of one variable and a list. It then applies that function to each element of the list and returns a new list made up of the results. For example, using the square function, you can take a list of numbers and square them (as shown in Listing 3).

Listing 3. Passing functions as arguments to functions in Scheme

```
;Our function
(define square (lambda (x) (* x x)))

;Our data
(define my-test-data '(1 2 3 4 5 6))

;Create a new list as the squares of my-test-data
(define list-of-squares (map square my-test-data))

;Display results
(display list-of-squares)
(newline)
```

This is a pretty concise program compared with other languages. For example, in C the equivalent code would look like this code in Listing 4.

Listing 4. Passing functions as arguments to functions in C

```
int square(int x)
{
   return x * x;
}

int main()
{
   int my_test_data[6] = {1, 2, 3, 4, 5, 6};;
   int list_of_squares[6];
   int i;

   /* Note that this would only work if C had a function
    * called map which worked with integer functions.
    * It doesn't, so this is largely theoretical.  Also remember
    * that C doesn't know the size of the arrays, so you have to
    * pass it explicitly.  Also, list_of_squares has to be passed
    * as a parameter in C.
    */
   map(square, my_test_data, list_of_squares, 6);

   printf("(");
   for(i=0; i<6; i++)
   {
      printf("%d ", list_of_squares[i]);
   }
   printf(")\n");
}
```

Of course, in C, unless you use a type-tag mechanism like the one mentioned in the "Better programming through effective list handling" article, you will need a different function for every type of value used with `map`.

In addition to being more concise than C, coding such a function in Scheme has an additional advantage -- you don't need to name the function `square`. Because functions can be created and passed just like any other value in Scheme, the core of the code can be modified so that you don't even have to give your squaring function a name. You can just define it and pass it to `map`. In place of the variable name `square` you will instead have the definition of the function written as a `lambda` form. Listing 5 shows the code.

Listing 5. Passing anonymous functions as values

```
(define list-of-squares (map (lambda (x) (* x x)) '(1 2 3 4 5 6)))
(display list-of-squares)
(newline)
```

So why do it this way? Well, anonymous functions have several advantages over named functions when the function is only used once:

- The program namespace isn't polluted with a lot of functions that are only used once.
- The code for the function is located in the exact spot where it is used, therefore programmers don't have to go hunting for the code of tiny, one-use functions.
- The function is clearly associated with the function to which it's being passed, therefore it is obvious to other programmers in what context it is being used. If a one-use function is written separate from the calling function, if the calling function is later removed, it won't be obvious that the one-use function would need to be removed as well.

In short, using anonymous functions makes the code concise and easy to follow and makes it obvious to other programmers when a function is specialized for a single instance or specific use.

Also, as seen from the `map` function, passing functions as arguments allows you more control over what happens to your data downstream. In this instance, the `map` function iterated through each value in the list and then *used your function* to manage further processing of that list value. This allows function writers and users great flexibility -- you can create a function that allows the users of the function to plug in extra functionality by passing functions as arguments.

In addition to `map`, there are several other generic list-processing functions that take functions as parameters. These functions, or ones like them, are the backbone of any data-processing application. Using these generic functions takes a lot of the repetitiveness out of programming; otherwise you are left to code loops after loops that all essentially work the same way. Not only do you save time, you also have fewer opportunities to introduce bugs.

Following is a list of the most common generic list-processing functions:

- `map` takes a list and generates a new list by applying a user-supplied function to each list member.
- `filter` takes a list and generates a new list containing the members of the old list that match a user-specified condition.
- `reduce` takes a list and combines the values into a single value. This combining procedure can be summation, min-value, max-value, etc. In order to use this correctly, you need to use *closures* (which will be discussed in the section Building functions at runtime).

There are several other generic list-processing functions, but these are the basic ones that are easiest to learn and most widely used. Detailed descriptions of these functions are available through the links found in the Related topics section.

Now that you've examined how to pass functions as arguments to other functions, let's look at how to write functions of your own that take functions as parameters.

Using functions as arguments

New programmers often have trouble seeing places where passing functions as arguments would be useful. In many programs, there will be one or two instances of what is basically the same algorithm coded multiple times but with slight variations. These are perfect candidates for functions as arguments.

What you can do is code the basic structure of the algorithm into one function. Then, you can take the pieces that vary and add them back in as function parameters. This allows a great amount of customization within the function. You can also add extra function parameters for future expansion. Combining functions like this removes redundancy from your programs, and by extension, reduces the errors that come from re-coding the same algorithm over and over within a program.

Suppose you have an order-processing algorithm composed of several parts that:

- Processes each line of the order and adds up the total.
- Calculates shipping on the order.
- Validates the credit line of the purchaser.
- If successful, charges the order, sends an order confirmation, and records it in the database.

Now let's say that different customers have different types of shipping to calculate, have their credit line calculated differently, and are charged differently for each order. For example, shipping might be calculated through a different service provider depending on the client. The credit line might be checked through your own business on some customers or through the credit-card company on others. Order charging might vary depending on whether the client was normally billed, charged through their credit card, or performs automatic withdrawal. Different customers may have different processing needs for each of these stages.

One way to code such a function would be to simply hardcode all of the possible pathways in your order-processing algorithm directly. Then the call to this function would include a list of flags indicating which style of processing was requested. However, as the number of possibilities gets larger, the order-processing algorithm would become unwieldy.

Another way to code the function is to have these stages handled by independent functions passed to the algorithm. This way, the order-processing algorithm needs to have only the general flow of the algorithm coded directly. The specifics of each major stage would be handled by functions passed in. Such an algorithm would need to have parameters for the shipping-calculation function, the credit-validation function, and the order-charging function.

Listing 6 demonstrates a function in Scheme that encodes such an algorithm.

Listing 6. Using functions as arguments in order processing

```
(define process-order
   ;;The function has one data record for the order and three functions as parameters
   (lambda (order ship-calc-func credit-validate-func charge-order-func)
      (let
         (
            ;;Here you execute functions to get the fields
            ;;from the order that you will need for later
            ;;processing
            (order-lines (get-order-lines order))
            (origin (get-origin-address order))
            (destination (get-destination-address order))
            (delivery-time (get-delivery-time order))
            (customer-email (get-customer-email order))

            ;;These values will be calculated when you
            ;;process the order
            (weight 0.0)
            (subtotal 0.0)
            (total 0.0)
            (shipping 0.0)
            (tax 0.0))
         ;;Iterate through each order line, running a function on each line
         ;;to add the weight and price to the weight and price of the order
         (for-each
            ;;Note that anonymous functions created within a function
            ;;have full access to the enclosing function's variables
            (lambda (order-line)
               (set! weight (+ weight (get-weight order-line)))
               (set! subtotal (+ total (get-price order-line))))
            order-lines)

         ;;This uses your shipping calculation function that was passed as a parameter.
         ;;Remember, this could be any number of things, such as calculating from
         ;;a shipping provider's remote database, calculating based on static
         ;;customer-specific tables, or any other method you wish.
         (set! shipping (ship-calc-func weight origin destination delivery-time))

         ;;For this exercise, tax will be considered fairly uniform, so you didn't pass
         ;;a tax-calculating function
         (set! tax (calculate-tax destination subtotal))

         ;;Now record the total
         (set! total (+ subtotal shipping tax))

         ;;Validate the user's credit line, according to the validation parameter
         (if (credit-validate-func total)
            (begin
               ;;Charge the order according to the validation parameter
               (charge-order-func total)
               (send-confirmation customer-email order-lines shipping tax total)
               (mark-as-charged order))
            (begin
               ;;Cancel the order
               (send-failure-message customer-email)
               (mark-as-failed order))))))
```

Parameters that are functions are passed just like any other parameter, the only difference being their use within the program. This technique allows you to have the algorithm code the general flow of control but still have the specifics of the processing parameterized. Many programmers who do not know this technique often have lots of flags that control the processing and that's not a bad idea when you have a few simple variations in processing. But it starts to get cumbersome as the number of differences grows.

It is difficult to determine at what point passing functions as parameters is more beneficial than having special cases and/or flags control the processing in a function. However, there are some guidelines:

- When the options are few and specific, special-casing is often the better method.
- When the options are so closely tied to the algorithm that it takes several functions using most of the algorithm's local variables to generate the desired options, special-casing is probably the better method.
- When each option needs completely different sets of variables to determine the answer, special-casing is probably the better method.
- When the options are many or if new options are anticipated, functions as parameters is often the better method.
- When the options are very clear and logically separate from the code, functions as parameters is often the better method.

The C language can also have functions passed as parameters, but writing a parameter specification for a function in C is a little confusing. C is statically typed -- that is, types are checked at compile-time -- while Scheme is not. Therefore, declaring a variable or parameter as holding a function requires that you also specify what types of parameters that function takes.

For example, take a look at how you might code the `map` function mentioned earlier. That function needed to take as a parameter a function of one integer that also returns an integer. The prototype of your `square` function looked like this: `int square(int x)`. Therefore, to declare a variable that holds a pointer to such a function, you would write the following code:

Listing 7. Variable declaration for a function pointer in C

```
/* This creates a variable called "the_function" which you can store
 * pointers to functions
 */
int (*the_function)(int);
```

The first `int` indicates that the return value is an integer. Then in parenthesis, you have the pointer notation and the name of the variable. In another set of parentheses after the function name is a comma-separated list of the types of arguments the function takes.

Assigning and using functions is actually very straightforward because *all* functions in C are actually function pointers.

Listing 8. Operations on function pointers in C

```
int main()
{
   /* Declare function pointer */
   int (*the_function)(int);

   /* assign function pointer */
   the_function = square;

   /* call function */
   printf("The square of 2 is %d\n", the_function(2));

   return 0;
}
```

Using this knowledge about function pointers in C, you can easily implement a version of Scheme's `map` function in C. However, your version will be much more limited because it will be handling only integers and integer arrays instead of *any* type of data and linked lists, as it would in Scheme.

Listing 9. Implementation of the map function in C

```
void map(int (*mapping_function)(int), int src[], int dst[], int count)
{
   int i;
   for(i = 0; i < count; i++)
   {
      dst[i] = mapping_function(src[i]);
   }
}
```

For an even more general approach, you could use the type-tag mechanism mentioned in "Better programming through effective list handling" and have the mapping function, the source array, and the destination array all be of type `data`.

When third-party libraries are used, functions passed to the library as arguments are usually called *callback functions*. This is because they are used to "call back" into the programmer's custom code from the library. This allows the library writers to write very generic functions which can then call back to the programmer's code for more specific processing.

With callback functions, the library can be extended by the programmer without having to directly modify the source code. For example, most GUI toolbox API's use callback functions for event handling. Callback functions are passed to the GUI API, which are then stored until the user triggers them with events such as mouse clicks and button pushes. The GUI API certainly doesn't know what actions the program needs to take in response to these events so it simply uses the callback functions provided by the calling program to direct the course of action.

Now, having covered functions that can be passed as parameters, I'm going to go a step further and discuss functions that can be *created* according to a programmer-defined template.

Building functions at runtime

In Scheme, `lambda` creates a function. However, functions in Scheme are "bigger" concepts than in many other languages.

To understand the difference, it is first necessary to understand how Scheme views variables. In Scheme, variables are grouped together in *environments*, of which there are two kinds -- the local environment and the global environment. The global environment consists of all of the global variables of a function; Scheme handles the global environment essentially the same as most other programming languages.

In addition to the global environment, Scheme has numerous local environments. A local environment is the collection of local variables and formal parameters that are visible at a given point in the program. But the local environment is more than just the names of the variables -- it's also the storage/values to which they refer. Unlike local environments in other programming languages, local environments in Scheme can be saved for later use.

So how do you save a local environment; what does that have to do with functions? In Scheme, when a function is defined, if it is defined within a local environment, that local environment permanently attaches itself to the function definition. Therefore the function has access to all of the variables that were active and in scope when the function was created, even if the function that created those variables has since returned. Let's look at some examples of functions with attached local environments.

Listing 10. Examples of functions declared within local environments

```
;Environment test is a function taking one argument.  It will then return
;a function which remembers that argument
(define make-echo-function
   (lambda (x)
      ;The local environment now consists of "x"

      ;Create a function which takes no parameters and return it
      (lambda ()
         ;This function returns the x defined in the enclosing
         ;local environment
         x)))

(define echo1 (make-echo-function 1))

(display (echo1)) ;should display 1
(newline)

(define echo2 (make-echo-function 2))

(display (echo2)) ;should display 2
(newline)

(display (echo1)) ;should display 1
```

```
(newline)
```

First notice that `echo1` and `echo2` are *functions*. Now `echo1` always gives back 1 even though it is not using any global variables and is not being passed any parameters. That's because when you create a function, the local environment is saved. In this case, the local environment consists of `x`. Therefore, unlike functions in languages like C, the variables created on each call of `make-echo-function` survive even past the point where `make-echo-function` ends! As long as the function returned from `make-echo-function` survives, the local environment will survive too.

In fact, when you define `echo2`, you now have two different local environments for `make-echo-function` that remain active. The variable names are the same, but the values in those variables are different. The computer knows which value to use because the environment is tied to the function when it is created by `lambda`. This strange fusion of environments and functions is called a *closure*.

Let's look at a slightly more complicated closure. You will create a function that starts at a given number and then, every time it is called, returns one number higher.

Listing 11. Number counter program to illustrate function-generating functions

```
(define make-counter
   (lambda (curval)
      (lambda ()
         (set! curval (+ curval 1))
         curval)))

(define my-counter (make-counter 0))

(display (my-counter)) ;writes 1
(newline)

(display (my-counter)) ;writes 2
(newline)

(define my-other-counter (make-counter 25))

(display (my-other-counter)) ;writes 26
(newline)

(display (my-counter)) ;writes 3
(newline)

(display (my-other-counter)) ;writes 27
(newline)
```

These types of functions can be useful for creating functions that maintain state from call to call. Combined with function parameters, this is a powerful technique for programming. You can make the functions passed as parameters even more powerful because Scheme closures allow you to ship pieces of state (the local environment) with the function. In programming languages such as C, any state that might be used in a function passed as an argument would either need to exist in a global variable or would need to be passed explicitly.

The point of having a function parameter within a procedure is to extend it beyond what was originally envisioned by the programmer. But if you have to explicitly pass all data you want used in your function, then you have already defeated its purpose. (I will discuss a solution to this problem later, but even then the solution isn't nearly as elegant as using closures in Scheme to attach a local environment to a function.)

When you put a function definition within an environment, what you are doing in essence is creating a template for a function. Every time the `lambda` form is evaluated, the template is used to create a new function using the current local environment to fill in the unknown slots.

Callback functions make closures quite handy. It is rare for the person who writes the API to know anything about the type of data that you are handling. Therefore, the writer doesn't know what kind of data needs to be passed to your callback function to make it work.

With closures, you can pack all of the data you want directly into a callback function. Let's say for instance, that an API wants a callback function of no parameters when it builds a button that will be called every time that button is pushed. That might seem problematic, especially if you have multiple buttons sharing a callback function. If the API isn't passing any data, how will you know which button was pushed?

With closures, you can define your callback function within an environment containing any information needed to make the callback effective. The API which calls receives and calls the callback doesn't need to know about any of it -- it just looks like a normal function to the API.

You can also use closures to create functions that create specialized versions of other functions. Let's say you have a function called `filter-evens` that takes a list and returns all even numbers from that list. The function might look like this:

Listing 12. Filtering even numbers from a list

```
(define filter-evens
   (lambda (the-list)
      (let
         (
            ;This will hold the filtered elements
            (new-list '()))
         ;Cycle through the elements
         (for-each
            (lambda (entry)
               ;if it's even, add it to new-list
               (if (even? entry)
                  (set! new-list (cons entry new-list))))
            the-list)
         ;return new-list (note that this will be in
         ;reverse order that the items were in the
         ;original list)
         new-list)))

(display (filter-evens '(1 2 3 4 5 6 7 8 9))) ;should display (8 6 4 2)
(newline)
```

Notice first that the function you define and pass to `for-each` is used as a closure. `new-list` is defined in the enclosing environment. `new-list` is going to be a different value each time `filter-evens` is called, but you will need that same value for each iteration of `for-each` in order to build the list.

Filtering even numbers is a nice exercise, but if you want to filter for something else -- like odd numbers, numbers below 20, or even filter on non-numeric lists (such as all addresses in the state of Oklahoma) -- you will basically have to re-code the algorithm even though each instance of the filtering algorithm would be largely similar. So, based on the criteria you discussed previously, the code that does the filtering selection would present a great opportunity to make a function parameter.

You can break out the filtering selection into its own function. This function will return true if the element is in the list and false otherwise. It can then be made a parameter to the function to allow for alternative filters. Here is how it would look:

Listing 13. A generic filter function

```
(define filter
   (lambda (filter-func the-list)
      (let
         (
            ;This will hold the filtered elements
            (new-list '()))
         ;Cycle through the elements
         (for-each
            (lambda (entry)
               ;if it matches the filter function
               ;add it to the result list
               (if (filter-func entry)
                  (set! new-list (cons entry new-list))))
            the-list)
         ;return new-list (note that this will be in
         ;reverse order that the items were in the
         ;original list)
         new-list)))

(define number-list '(1 2 3 4 5 6 7 8 9))

(display (filter even? number-list)) ;should display (8 6 4 2)
(newline)

(display (filter odd? number-list)) ;should display (9 7 5 3 1)
(newline)

(display (filter (lambda (x) (< x 5)) number-list)) ;should display (4 3 2 1)
(newline)
```

You now have a pretty useful function. Sometimes though, it may be useful to pre-package a filter function so you don't have to write it out every time you use it. This is especially true if the function parameter is a long lambda expression. In order to help create new, specialized filtering functions, you can define a new function that will generate new filtering functions. Let's call this function `make-filter` (Listing 14):

Listing 14. A filter-generating function

```
(define make-filter
   (lambda (comparison)
      (lambda (the-list)
         (filter comparison the-list))))

(define filter-evens (make-filter even?))
(define filter-odds (make-filter odd?))
(define filter-under20 (make-filter (lambda (x) (< x 20))))
```

Although these examples are quite trivial, when you have more complicated filtering procedures, `make-filter` can save a lot of time and errors.

As mentioned, in the C programming language closures are not directly supported. C-language functions have no data associated with them when they are defined. You can have static variables, but you can only have one value in the static variable at a time. With closures, each closure has its own environment attached, allowing the programmer to have multiple instances of the function each with its own local environment.

Under the hood, Scheme closures are a structure containing two pointers -- one to the function itself and one to the environment. That allows you to simulate a closure in C by having a struct with two pointers -- one a function pointer and the other a void pointer. You use a void pointer so that you don't have to worry about what the environment looks like. You give the programmer the freedom to choose.

The function pointer is declared in a very general way in order to support a large number of functions without casting. The closure structure looks like this:

Listing 15. Closure definition in C

```
typedef void * (*generic_function)(void *, ...);
typedef struct {
   generic_function function;
   void *environment;
} closure;
```

The `generic_function` type was created in order to have the most generic calling interface to be used in a wide variety of situations. It also makes it easy to typecast other function types to be used within the closure. The environment is a void pointer, again for ease of use in casting. The parts of the program that create and consume the environment are responsible for knowing it's layout. The `closure` struct is only responsible for holding the pointer. Some APIs for closures define an entire marshalling system for creating and using closures (see the Related topics section for a link), but this is all that's necessary for a basic implementation.

Let's look at a C version of your counter program to see how all of this works together:

Listing 16. Counter closure in C

```
#include <stdio.h>
#include <stdlib.h>

/* Closure definitions */
typedef void *(*generic_function)(void *p, ...);
typedef struct {
    generic_function function;
   void *environment;
} closure;

int nextval(void *environment);

/* This is the function that creates the closure */
closure make_counter(int startval)
{
   closure c;
   /* The data is very simple.  You are just storing an int,
      so all you need is a pointer to an int.
    */
   int *value = malloc(sizeof(int));
   *value = startval;

   /* Setup the closure */
   c.function = (generic_function)nextval;
   c.environment = value;

   /* Return the closure */
   return c;
}

/* This is the function that is used for the closure */
int nextval(void *environment)
{
   /* convert the environment data back into the form used
    * by your functions
    */
   int *value = environment;

   /* Increment */
   (*value)++;

   /* Return the result */
   return (*value);
}

int main()
{
   /* Create the two closures */
   closure my_counter = make_counter(2);
   closure my_other_counter = make_counter(3);

   /* Run the closures */
   printf("The next value is %d\n", ((generic_function)my_counter.function)
                                  (my_counter.environment))
   printf("The next value is %d\n", ((generic_function)my_other_counter.function)
                                  (my_other_counter.environment));
   printf("The next value is %d\n", ((generic_function)my_counter.function)
                                  (my_counter.environment));

   return 0;
}
```

We're missing some clean-up here, but if you use garbage collection as explained in the article "Inside Memory Management" (see Related topics), it won't be an issue.

Many API's perform closures differently than described here. Instead of defining a closure structure which contains both the function pointer and the environment pointer, whenever a closure is needed the function pointer and the environment pointer are passed separately in the function call. Either way, faking closures in C gets more difficult to manage as the environments being passed become more complex. The Scheme language is set up to do all the work for you, so why not take advantage of that?

Next, let's take a look at the relationship between closures in Scheme and objects in object-oriented languages.

Functions and object-oriented programming

Although it may not be immediately obvious, there is a direct relationship between closures in Scheme and objects in object-oriented languages. Think back to when you made your counter function. What components did you have? You had a function that created a set of local variables and then returned a single function (a closure) that acted on those variables (because of the environment). Let's look at the parts of object-oriented programming you have already here:

- The function that created the function operates exactly like a constructor.
- The local variables defined in the environment during the constructor behave exactly like instance/member variables of an object.
- The returned function behaves like a member function.

The only things missing are the ability to declare *multiple* member functions, destructors, and more object-oriented syntax. In fact, you can view an object as simply a set of functions defined over the same local environment. Or, hinting at a possible implementation, you could call them a *vector* of functions defined over the same local environment.

Let's see how your counter would look in an object-oriented language like C++.

Listing 17. The counter function rewritten as a class

```
class Counter
{
   private:
   int value;
   public:
   Counter(int initial_value)
   {
      value = initial_value;
   }

   int nextValue()
   {
      value++;
      return value;
   }
};
```

Of course, as mentioned earlier, in order to get real object-oriented programming, you need to be able to define a vector of functions over the same closure. So let's do that. You'll use your same counter code, adding a `setValue()` method to set the current value to whatever you choose.

Listing 18. Counter functions written as objects

```
(define make-counter
   (lambda (value)
      (vector
         (lambda ()
            (set! value (+ value 1))
            value)
         (lambda (new-value)
            (set! value new-value)
            value))))
(define nextValue (lambda (obj) (vector-ref obj 0)))
```

```
(define setValue (lambda (obj) (vector-ref obj 1)))

(define my-counter (make-counter 3))

(display ((nextValue my-counter))) ;displays 4
(newline)

((setValue my-counter) 25) ;now my-counter's value is 25

(display ((nextValue my-counter))) ;displays 26
(newline)
```

As can be seen in the `make-counter` function, in order to get a vector of functions defined over a local environment, you just defined a vector where each member was a function defined within the same environment. However, there is some difficulty in referring to the functions in the vector since vectors are referenced by position. Remembering the offset of each function within the vector would be a pain, especially if you had multiple classes. Calling `((vector-ref my-counter 0))` for example, would be horribly nonintuitive. Therefore, you defined some helper functions to look up those indexes for you.

In the program, there is what looks like an extra set of parentheses around your method calls. The extra parentheses is included because your helper functions only look up the function -- they don't actually call it. `nextValue` only looks up the nextvalue function on the object. After being looked up, the function still has to be called. The second set of parentheses actually performs the call. The two separate function calls in this implementation allow you to see more clearly that it is a two-step process.

Here is the program rewritten to use a one-step function call for object methods:

Listing 19. Counter functions written as objects with combined lookup/call steps

```
(define make-counter
   (lambda (value)
      (vector
         (lambda ()
            (set! value (+ value 1))
            value)
         (lambda (new-value)
            (set! value new-value)
            value))))
(define nextValue (lambda (obj) ((vector-ref obj 0))))
(define setValue (lambda (obj value) ((vector-ref obj 1) value)))

(define my-counter (make-counter 3))

(display (nextValue my-counter)) ;displays 4
(newline)

(setValue my-counter 25) ;now my-counter's value is 25

(display (nextValue my-counter)) ;displays 26
(newline)
```

Note that this mechanism also allows for single inheritance. Because the way you are looking up the function is based on the index into the array, inheritance depends on the functions in compatible classes and subclasses to have the corresponding functions in the same location. If you were doing multiple inheritance, several functions would each need to be at the same slot. Multiple inheritance (or even just Java-style interfaces), while possible, needs a different function-lookup mechanism and is much more difficult to implement.

Of the highest order

So, why go to all that trouble to define objects when there are good object-oriented languages and object-oriented extensions to Scheme already available? Well really, you shouldn't; however, I wanted to point out the basic equivalence between objects and closures. In fact, it's almost a rite of passage for a programmer to have built his or her own object system using closures while learning Scheme.

Why have both closures and objects when they are both somewhat equivalent?

- For small systems, closures are almost always easier to handle than objects. The amount of code that goes into creating a class for an object dwarfs the code needed to make an anonymous function on a local environment.
- However, when you need to define several functions (probably more than three or four) that work in tandem on a local environment, objects often work better.

Now that you know how both systems work, if they language you are working in is missing one or the other, you can always use what you have to simulate what you don't have.

Related topics

- Teach Yourself Scheme in Fixnum Days is a great online tutorial of the Scheme language.
- The Function Pointer Tutorials site has a great discussion on using function pointers in C/C++.
- SRFI 1 contains a whole load of useful functions for lists, many of which takes function parameters to extend their processing (SRFI's are like RFC's for Scheme).
- Inside memory management (developerWorks, November 2004) details how memory management works, how to manage memory manually, how to manage memory semi-manually using referencing counting or pooling, and how to manage memory automatically using garbage collection.
- Charming Python: Functional programming in Python, Part 1 (developerWorks, March 2001) introduces some fundamental concepts of functional programming and shows how to implement functional programming techniques in Python.
- Charming Python: Functional programming in Python, Part 2 (developerWorks, April 2001) covers closures in Python.
- Charming Python: Functional programming in Python, Part 3 (developerWorks, June 2001) covers currying (named after the logician Haskell Curry, this allows the return value of functions to themselves be functions but with the returned functions "narrowed") and other high order functions.
- Innovate your next Linux development project with IBM trial software, available for download directly from developerWorks.