2

FUNCTIONS

We have already *used* several functions in the previous chapter—things such as `alert` and `print`—to order the machine to perform a specific operation. In this chapter, we will start *creating* our own functions, making it possible to extend the vocabulary that we have available. In a way, this resembles defining our own words inside a story we are writing to increase our expressiveness. Although such a thing is considered rather bad style in prose, in programming it is indispensable.

The Anatomy of a Function Definition

In its most basic form, a function definition looks like this:

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

square(12);
→ 144
```

Here, `square` is the name of the function. `x` is the name of its (first and only) argument. `return x * x;` is the body of the function.

The keyword `function` is always used when creating a new function. When it is followed by a variable name, the new function will be stored under this name. After the name comes a list of argument names and finally the body of the function. Unlike those around the body of `while` loops or `if` statements, the braces around a function body are obligatory.

The keyword `return`, followed by an expression, is used to determine the value the function returns. When control comes across a `return` statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. A `return` statement without an expression after it will cause the function to return `undefined`.

A body can, of course, have more than one statement in it. Here is a function for computing powers (with positive, integer exponents):

```
function power(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}

power(2, 10);
→ 1024
```

The arguments to a function behave like variables—but ones that are given a value by the *caller* of the function, not the function itself. The function is free to give them a new value though, just like normal variables.

Definition Order

Even though function definitions occur as statements between the rest of the program, they are not part of the same timeline. In the following example, the first statement can call the `future` function, even though its definition comes later:

```
print("The future says: ", future());

function future() {
  return "We STILL have no flying cars.";
}
```

What is happening is that the computer looks up all function definitions, and stores the associated functions, *before* it starts executing the rest of the program. The nice thing about this is that we do not have to think about the order in which we define and use our functions—they are all allowed to call each other, regardless of which one is defined first.

Local Variables

A very important property of functions is that the variables created inside of them are *local* to the function. This means, for example, that the result variable in the power example will be newly created every time the function is called and will no longer exist after the function returns. In fact, if power were to call itself, that call would cause a *new*, distinct result variable to be created and used by the inner call and would leave the variable in the outer call untouched.

This “localness” of variables applies only to the arguments of the function and those variables that are declared with the var keyword inside the function. It is possible to access *global* (nonlocal) variables inside a function, as long as you haven’t declared a local variable with the same name.

The following code demonstrates this. It defines (and calls) two functions that both change the value of the variable x. The first one does not declare the variable as local and thus changes the global variable defined at the start of the example. The second does declare it and ends up changing only the local variable.

```
var x = "A";

function setVarToB() {
  x = "B";
}
setVarToB();
x;
→ "B";

function setVarToC() {
  var x;
  x = "C";
}
setVarToC();
x;
→ "B";
```

As an aside, note that these functions contain no return statements, because they are called for their side effects, not to create a value. The actual return value of such functions is undefined.

Nested Scope

In JavaScript, it is not enough to simply distinguish between *global* and *local* variables. In fact, there can be any number of stacked (or nested) variable scopes. Functions defined inside other functions can refer to the local variables in their parent function, functions defined inside those inner functions can refer to variables in both their parent and their grandparent functions, and so on.

Take a look at this example. It defines a function that takes the absolute (positive) value of `number` and multiplies that by `factor`.

```
function multiplyAbsolute(number, factor) {
  function multiply(number) {
    return number * factor;
  }
  if (number < 0)
    return multiply(-number);
  else
    return multiply(number);
}
```

The example is intentionally confusing in order to demonstrate a subtlety—it contains two separate variables named `number`. When the body of the function `multiply` runs, it uses the same `factor` variable as the outer function but has its own `number` variable (created for the argument of that name). Thus, it multiplies its own argument by the `factor` passed to `multiplyAbsolute`.

What this comes down to is that the set of variables visible inside a function is determined by the place of that function in the program text. All variables that were defined "above" a function's definition are visible, which means both those in function bodies that enclose it and those at the top level of the program. This approach to variable visibility is called *lexical scoping*.

People who have experience with other programming languages might expect that a block of code (between braces) also produces a new local environment. Not in JavaScript. Functions are the only things that create a new scope. You are allowed to use free-standing blocks:

```
var something = 1;
{
  var something = 2;
  // Do stuff with variable something...
}
// Outside of the block again...
```

But the `something` inside the block refers to the same variable as the one outside the block. In fact, although blocks like this are allowed, they are only useful to group the body of an `if` statement or a loop. (Most people agree that this is a bit of a design blunder by the designers of JavaScript, and later versions of the language will add some way to define variables that stay inside blocks.)

The Stack

To understand how functions are called and how they return, it is useful to be aware of a thing called the *stack*. When a function is called, control is given to the body of that function. When that body returns, the code that called the function is resumed. Thus, while the body is running, the computer must remember the context from which the function was called so that it knows where to continue afterward. The place where this context is stored is the stack.

The reason that it is called a stack has to do with the fact that, as we saw, a function body can again call a function. Every time a function is called, another context has to be stored. One can visualize this as a stack of contexts. Every time a function is called, the current context is thrown on top of the stack. When a function returns, the context on top is taken off the stack and resumed.

This stack requires space in the computer's memory to be stored. When the stack grows too big, the computer will give up with a message like "out of stack space" or "too much recursion." The following code illustrates that—it asks the computer a really hard question, which causes an infinite back-and-forth between two functions. Or rather, it would be infinite, if we had an infinite stack. As it is, it will run out of space, or "blow the stack."

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
print(chicken() + " came first.");
```

Function Values

As I mentioned in the previous chapter, *everything* in JavaScript is a value, including functions. This means that the names of defined functions can be used like normal variables, and their content can be passed around and used in bigger expressions. The following example will call the function in variable `a`, unless that is a "false" value (like `null`), in which case it chooses and calls `b` instead.

```
var a = null;
function b() {return "B";}
(a || b)();
→ "B"
```

The bizarre-looking expression `(a || b)()` applies the "call without arguments" operation represented by `()` to the expression `(a || b)`. If that expression does not produce a function value, this will of course produce an error. But when it does, as in the example, the resulting value is called, and all is well.

When we simply need an unnamed function value, the `function` keyword can be used as an expression, like this:

```
var a = null;
(a || function(){return "B";})();
→ "B"
```

This produces the same effect as the previous example, except that this time no function named `b` is defined. The "nameless" (or "anonymous") function expression `function(){return "B";}` simply creates a function value. It is possible to specify arguments or multistatement bodies in such definitions as well.

In Chapter 5, the *first-class* nature of functions (which is the usual term used for the "functions are values" concept) will be further explored and used to write some very clever code.

Closure

The nature of the function stack, combined with the ability to treat functions as values, brings up an interesting question. What happens to local variables when the function call that created them is no longer on the stack? The following code illustrates this:

```
function createFunction() {
  var local = 100;
  return function(){return local;};
}
```

When `createFunction` is called, it creates a local variable and then returns a function that returns this local variable. The question of how to treat this situation is known as the "upwards Funarg problem," and many old programming languages simply forbid it. JavaScript, fortunately, is from a generation of languages that solve this problem by going out of their way to preserve the local variable as long as it is in any way reachable. Doing `createFunction()()` (creating the function and then calling it) results in the value `100` being returned, as hoped.

This feature is called *closure*, and a function that "closes over" some local variables is called *a closure*. This behavior not only frees you from having to worry about variables still being "alive" but also allows for some creative use of function values.

For example, the following function makes it possible to dynamically create function values that add a certain number to their argument:

```
function makeAdder(amount) {
  return function(number) {
    return number + amount;
  };
}

var addTwo = makeAdder(2);
addTwo(3);
→ 5
```

Optional Arguments

It turns out we can execute the following code:

```
alert("Hello", "Good Evening", "How do you do?", "Good-bye");
```

The function `alert` officially accepts only one argument. Yet when you call it like this, it does not complain. It simply ignores the other arguments and shows you `Hello`.

JavaScript is notoriously nonstrict about the amount of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing ones get the value `undefined`. The downside of this is that it is possible—even likely—that you'll accidentally pass the wrong number of arguments to functions, and no one will tell you about it.

The upside of this is that it can be used to have a function take "optional arguments." For example, this version of `power` can be called with only a single argument, in which case it behaves like `square`:

```
function power(number, exponent) {
  var result = 1;
  if (exponent === undefined)
    exponent = 2;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
}
```

In the next chapter, we will see a way in which a function body can get at the exact list of arguments that were passed to it. This can be useful, because it makes it possible to have a function accept any number of arguments. `print` makes use of this—the following prints `R2D2`:

```
print("R", 2, "D", 2);
```

Techniques

Now that we have a rather good idea of what JavaScript functions are and how they function, we will look at some considerations that come into play when designing and writing them.

Avoiding Repetition

The reason functions were invented is to reuse pieces of code. Programs typically need to perform the same operation (such as exponentiation) multiple times, and when you repeat the full code needed to perform the operation every time you need it, your program is going to be a lot longer.

Not only will it be longer, but it will also be more boring to read and more likely to contain errors. For example, the `power` function we defined does not work with negative exponents. If you find out that those are also needed, you'd have to update all the places where you take the power of a number and fix them. If you defined a function, all it takes is fixing the function, and all uses of it will suddenly work correctly.

When finding you need the same piece of code more than once and deciding to move it into a function, you need to determine how much of the code should go into the function and what the interface to the function should look like. For example, say we have some code to print a zero-padded number, like this:

```
var number = 5;
if (number < 10)
  print("0", number);
else
  print(number);
```

But it turns out we need to print padded numbers in other places as well. We now have several choices to make.

Do we make a function at all? The occurrences of the code might be in different projects, making it more work to share functions. Usually, the answer to this is "yes," regardless.

Does the function include the printing action, or does it just produce a zero-padded string? The best functions are those that perform a single, simple action, since they are easier to name (and thus easier to understand) and can be used in a wider variety of situations. So, write a `zeroPad` function, rather than a `printZeroPadded` function. `print(zeroPad(5))` is no harder to type than `printZeroPadded(5)`, after all.

How smart and versatile should the function be? We could write anything from a terribly simple "pad this number with a single zero" function to an involved formatted-output system that handles fractional numbers, rounding, and table layout. A good principle is to not add cleverness unless you are absolutely sure you are going to need it. It is tempting to fall into the trap of writing complicated "frameworks" for every little bit of functionality you need and never getting any actual work done. In this case, a second argument that specifies the width of the resulting number sounds like a useful, simple addition.

```
function zeroPad(number, width) {
  var string = String(Math.round(number));
  while (string.length < width)
    string = "0" + string;
  return string;
}
```

`Math.round` is a function that rounds a number; `String` is a function that converts its argument to a string.

Purity

"Purity," when applied to functions, is not about their lack of contaminants or their sexual behavior, but about whether they have side effects. *Pure functions* are the things that mathematicians mean when they say "function." They always return the same value when given the same arguments and do not have side effects.

The distinction between pure and nonpure functions is interesting mostly in terms of good code design and mental overhead. If a function is pure, a call to it can be mentally substituted by its result without changing the meaning of the code. When you are not sure that it is working correctly, you can test it by simply calling it and know that if it works in that context, it will work in any context. Nonpure functions might return different values based on all kinds of factors and have side effects that might be hard to test and think about.

Because pure functions are self-sufficient, they are likely to be useful and relevant in a wider range of situations than nonpure ones. Take the `zeroPad` function that we wrote earlier, for example. Had we written `printZeroPadded` instead, the function would have been useful only in situations where a `print` function had been defined and where we wanted to directly print our padded number. When defined as a pure function from a number to a string, the function depends on less context and is more generally applicable.

Of course, `zeroPad` solves a different problem than `print`, and no pure function is going to be able to do what `print` does, because it requires a side effect. In many cases, nonpure functions are precisely what you need. In other cases, a problem can be solved with a pure function, but the nonpure variant is much more convenient or efficient. Generally, when something can naturally be expressed as a pure function, write it that way. You'll thank yourself later. If not, don't feel dirty for writing nonpure functions.

Recursion

As mentioned earlier, it is valid for a function to call itself. A function that calls itself is called *recursive*. Recursion allows for some interesting function definitions. Look at this alternate implementation of `power`:

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
    return base * power(base, exponent - 1);
}
```

This is rather close to the way mathematicians define exponentiation, and conceptually it looks a lot nicer than the earlier version. It sort of loops, but there is no `while`, `for`, or even a local side effect to be seen. By calling itself, the function produces the same effect that was produced with a `for` loop before.

There is one important problem: In most JavaScript implementations, this second version is about 10 times slower than the first one. In JavaScript, running through a simple loop is a *lot* cheaper than calling a function multiple times. On top of that, using a sufficiently large exponent to this function might cause the stack to overflow.

The dilemma of speed versus elegance is an interesting one and is not limited to debates about recursion. In many situations, an elegant, intuitive, and often short solution can be replaced by a more convoluted but faster solution.

In the case of the earlier `power` function, the inelegant version is still sufficiently simple and easy to read. It does not make much sense to replace it with the recursive version. Often, though, the concepts a program is dealing with get so complex that giving up some efficiency in order to make the program more straightforward becomes an attractive choice.

The basic rule, which has been repeated by many programmers and with which I wholeheartedly agree, is to not worry about efficiency until your program is provably too slow. When it is, find out which parts are taking up the most time, and start exchanging elegance for efficiency in those parts.

Of course, the previous rule doesn't mean one should start ignoring performance altogether. In many cases, like the `power` function, not much simplicity is gained by the "elegant" approach. In other cases, an experienced programmer can see right away that a simple approach is never going to be fast enough.

The reason I am making a big deal out of this is that surprisingly many programmers focus fanatically on efficiency, even in the smallest details. The result is bigger, more complicated, and often less correct programs, which take longer to write than their more straightforward equivalents and often run only marginally faster.

Recursion is not always just a less-efficient alternative to looping. Some problems are much easier to solve with recursion than with loops. Most often these are problems that require exploring or processing several "branches," each of which might branch out again into more branches.

Consider this puzzle: By starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite amount of new numbers can be produced. How would you write a function that, given a number, tries to find a sequence of additions and multiplications that produce that number?

For example, the number 13 could be reached by first multiplying 1 by 3 and then adding 5 twice. The number 15 cannot be reached at all.

Here is the solution:

```
function findSequence(goal) {
  function find(start, history) {
    if (start == goal)
      return history;
    else if (start > goal)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
             find(start * 3, "(" + history + " * 3)");
  }
  return find(1, "1");
}

findSequence(24);
→ (((1 * 3) + 5) * 3)
```

Note that it doesn't necessarily find the *shortest* sequence of operations—it is satisfied when it finds any sequence at all.

How does it work? The inner `find` function, by calling itself in two different ways, explores both the possibility of adding 5 to the current number and of multiplying it by 3. When it finds the number, it returns the `history` string, which is used to record all the operators that were performed to get to this number. It also checks whether the current number is bigger than `goal`. If it is, we should stop exploring this branch, since it is not going to give us our number.

The use of the `||` operator in the example can be read as "return the solution found by adding 5 to `start`, and if that fails, return the solution found by multiplying `start` by 3." Equivalent (but more wordy) code would look like this:

```
else {
  var found = find(start + 5, "(" + history + " + 5)");
  if (found == null)
    found = find(start * 3, "(" + history + " * 3)");
  return found;
}
```