

# Writing Self-Modifying Perl

Spencer Tipping

December 27, 2010

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>I</b>	<b>The Basics</b>	<b>4</b>
<b>2</b>	<b>A Big Quine</b>	<b>5</b>
2.1	A basic quine . . . . .	5
2.2	Reducing duplication . . . . .	6
2.3	Using eval . . . . .	6
<b>3</b>	<b>Building the interface</b>	<b>8</b>
3.1	Using an editor . . . . .	9
<b>4</b>	<b>Namespaces</b>	<b>12</b>
4.1	Handling functions more usefully . . . . .	12
4.2	Catching attribute creation . . . . .	13
4.3	Putting it all together . . . . .	15
4.4	Separating bootstrap code . . . . .	17
<b>5</b>	<b>Serialization</b>	<b>21</b>
5.1	Fixing the EOF markers . . . . .	22
5.2	Verifying serialization . . . . .	22
5.2.1	Implementing the Fowler-Noll Vo hash . . . . .	22
5.2.2	Fixing EOF markers again . . . . .	24
5.2.3	Implementing the state function . . . . .	24
5.2.4	Implementing the verify function . . . . .	24
5.3	Save logic . . . . .	25
5.4	code::main fixes . . . . .	25
5.5	Final result . . . . .	26
<b>6</b>	<b>Adding a REPL</b>	<b>30</b>
6.1	The data data type . . . . .	30
6.2	Setting up the default action . . . . .	31
6.3	Making the script executable . . . . .	31
6.4	The shell function . . . . .	31

6.5	Taking it to the max: tab-completion . . . . .	32
6.6	Final result . . . . .	33
<b>7</b>	<b>Some improvements</b>	<b>39</b>
7.1	Useful functions . . . . .	39
7.2	Making some functions internal . . . . .	40
7.3	Separate attributes for data types . . . . .	42
7.3.1	Factoring externalization . . . . .	43
7.4	Abstracting %data . . . . .	44
7.4.1	Dynamic execution . . . . .	45
7.5	Final result . . . . .	45
<b>II</b>	<b>The Fun Stuff</b>	<b>52</b>
<b>8</b>	<b>eval backtraces</b>	<b>53</b>
<b>9</b>	<b>Archiving state</b>	<b>55</b>
9.1	Saving state . . . . .	55
9.2	Loading state . . . . .	56
9.3	The hypothetically function . . . . .	56
<b>10</b>	<b>Cloning and inheritance</b>	<b>58</b>
10.1	Tracking inheritability . . . . .	58
10.2	Extensions to serialize . . . . .	60
10.3	The update-from function . . . . .	62
10.4	Managing parents . . . . .	63
10.4.1	The parent:: namespace . . . . .	63
10.4.2	Uniqueness . . . . .	63
10.4.3	Updating ls and serialize . . . . .	64
10.4.4	An updated update-from function . . . . .	65
10.4.5	The update function . . . . .	66
10.5	clone and child . . . . .	66

# Chapter 1

## Introduction

I've gotten a lot of WTF's<sup>1</sup> about self-modifying Perl scripts. Rightfully so, too. There's no documentation (until now), the interface is opaque and not particularly portable, and they aren't even very human-readable when edited:

```
...
meta::define_form 'meta', sub {
    my ($name, $value) = @_;
    meta::eval_in($value, "meta::$name");
};
meta::meta('configure', <<'__25976e07665878d3fae18f050160343f');
# A function to configure transients. Transients can be used to store any number of
# different things, but one of the more common usages is type descriptors.
sub meta::configure {
    my ($datatype, %options) = @_;
    $transient{$_}{$datatype} = $options{$_} for keys %options;
}
__25976e07665878d3fae18f050160343f
...
```

Despite these shortcomings, though, I think they're fairly useful (this guide is a self-modifying Perl file, in fact). At the end, you'll have a script that is functionally equivalent to the object script, which I use as the prototype for all of the other ones.<sup>2</sup> The full source code for this guide and accompanying examples is available at <http://github.com/spencertipping/writing-self-modifying-perl>.

Proceed only with fortitude, determination, and Perl v5.10.

---

<sup>1</sup>[http://www.osnews.com/story/19266/WTFs\\_m](http://www.osnews.com/story/19266/WTFs_m)

<sup>2</sup>See <http://github.com/spencertipping/perl-objects> for the full source.

**Part I**

**The Basics**

## Chapter 2

# A Big Quine

At the core of things, a self-modifying Perl script is just a big quine.<sup>1</sup> There are only two real differences:

1. Self-modifying Perl scripts print into their own files rather than to standard output.
2. They print modified versions of themselves, not the original source.

If we're going to write such a script, it's good to start with a simple quine.

### 2.1 A basic quine

Some languages make quine-writing easier than others. Perl actually makes it very simple. Here's one:

Listing 2.1 examples/quine

```
1 my $code = <<'EOF';  
2 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;  
3 EOF  
4 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
```

The logic is fairly straightforward, though it may not look like it. We're quoting a bunch of stuff using `<<'EOF'`,<sup>2</sup> and storing that into a string. We then put the quoted content outside of the heredoc to let it execute. The duplication is necessary; we want to quote the content and then run it.<sup>3</sup> The key is this line:

```
print 'my $code = <<'EOF\';', "\n", $code, "EOF\n"; print $code;
```

This code prints the setup to define a new variable `$code` and prints its existing content after that.

---

<sup>1</sup>A “quine” being a program that prints its own source.

<sup>2</sup>The single-quoted heredoc form doesn't do any interpolation inside the document, which is ideal since we don't want to worry about escaping stuff.

<sup>3</sup>Later on I'll use `eval` to reduce the amount of duplication.

## 2.2 Reducing duplication

We don't want to write everything in our quine twice. Rather, we want to store most stuff just once and have a quine that scales well. The easiest way to do this is to use a hash to store the state, and serialize each key of the hash in the self-printing code. So instead of creating `$code`, we'll create `%data`:

Listing 2.2 examples/quine-with-data

```
1 my %data;
2 $data{code} = <<'EOF';
3 print 'my %data;', "\n";
4 print '$data{', $_, '}' = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
5 print $data{code};
6 EOF
7 print 'my %data;', "\n";
8 print '$data{', $_, '}' = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
9 print $data{code};
```

This is a good start. Here's how to add attributes without duplication:

Listing 2.3 examples/quine-with-data-and-foo

```
1 my %data;
2 $data{foo} = <<'EOF';
3 a string
4 EOF
5 $data{code} = <<'EOF';
6 print 'my %data;', "\n";
7 print '$data{', $_, '}' = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
8 print $data{code};
9 EOF
10 print 'my %data;', "\n";
11 print '$data{', $_, '}' = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
12 print $data{code};
```

## 2.3 Using eval

The business about duplicating `$data{code}` is easily remedied by just evaling `$data{code}` at the end. This requires the `eval` section to be duplicated, but it's smaller than `$data{code}`. Here's the quine with that transformation:<sup>4</sup>

Listing 2.4 examples/quine-with-data-and-eval

```
1 my %data;
2 $data{foo} = <<'EOF';
```

---

<sup>4</sup>Note that these quines might not actually print themselves identically due to hash-key ordering. This is fine; all of the keys are printed before we use them.

```

3  a string
4  EOF
5  $data{code} = <<'EOF';
6  print 'my %data;', "\n";
7  print '$data{', $_, '}' = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
8  print $data{bootstrap};
9  EOF
10 $data{bootstrap} = <<'EOF';
11 eval $data{code};
12 EOF
13 eval $data{code};

```

The advantage of this approach is that all we'll ever have to duplicate is `eval $data{code}` and `my %data;`, which is fairly trivial. It's important that you understand what's going on here, since this idea is integral to everything going forward.



## Chapter 3

# Building the interface

Now that we've got attribute storage working, let's build a command-line interface so that we don't have to edit these files by hand anymore. There are a couple of things that need to happen. First, we need to get these scripts to overwrite themselves instead of printing to standard output. Second, we need a way to get and set entries in %data. Starting with the quine from the last section, here's one way to go about it:

Listing 3.1 examples/cli-basic

```
1 my %data;
2 $data{cat} = <<'EOF';
3 sub cat {
4     print join "\n", @data{@_};
5 }
6 EOF
7 $data{set} = <<'EOF';
8 sub set {
9     $data{$_[0]} = join '', <STDIN>;
10 }
11 EOF
12 $data{code} = <<'EOF';
13 # Eval functions into existence:
14 eval $data{cat};
15 eval $data{set};
16
17 # Run specified command:
18 my $command = shift @ARGV;
19 &$command(@ARGV);
20
21 # Save new state:
22 open my $fh, '>', $0;
23 print $fh 'my %data;', "\n";
```

```

24 print $fh '$data{', $_, '}' = <<'EOF\';', "\n$data{$_}EOF\n" for keys %data;
25 print $fh $data{bootstrap};
26 close $fh;
27 EOF
28 $data{bootstrap} = <<'EOF';
29 eval $data{code};
30 EOF
31 eval $data{code};

```

Now we can modify its state:

```

$ perl examples/cli-basic cat cat
sub cat {
    print join "\n", @data{@_};
}
$ perl examples/cli-basic set foo
bar
^D
$ perl examples/cli-basic cat foo
bar
$

```

Not bad for a first implementation. This is a very minimal self-modifying Perl file, though it's useless at this point. It also has some fairly serious deficiencies (other than being useless). I'll cover the serious problems later on, but first let's address the usability.

### 3.1 Using an editor

The first thing that would help this script be more useful is a function that let you edit things with a real text editor. Fortunately this isn't difficult:

```

$ cp examples/cli-basic temp
$ perl temp set edit
sub edit {
    my $filename = '/tmp/' . rand();
    open my $file, '>', $filename;
    print $file $data{$_[0]};
    close $file;

    system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);

    open my $file, '<', $filename;
    $data{$_[0]} = join ' ', <$file>;
    close $file;
}

```

```
^D
$
```

It won't work yet though. The reason is that we aren't evaling `edit` yet; we need to manually edit the code section and insert this line:

```
...
eval $data{cat};
eval $data{set};
eval $data{edit};          # <- insert this
...
```

Now you can invoke a text editor on any defined attribute:<sup>1</sup>

```
$ perl examples/cli-editor edit cat
# hack away
$
```

Here's the object at this point:

**Listing 3.2** examples/cli-editor

```
1 my %data;
2 $data{cat} = <<'EOF';
3 sub cat {
4     print join "\n", @data{@_};
5 }
6 EOF
7 $data{set} = <<'EOF';
8 sub set {
9     $data{$_[0]} = join ' ', <STDIN>;
10 }
11 EOF
12 $data{edit} = <<'EOF';
13 sub edit {
14     my $filename = '/tmp/' . rand();
15     open my $file, '>', $filename;
16     print $file $data{$_[0]};
17     close $file;
18
19     system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
20
21     open my $file, '<', $filename;
22     $data{$_[0]} = join ' ', <$file>;
23     close $file;
24 }
```

---

<sup>1</sup>Don't modify bootstrap or break the print code though! This will possibly nuke your object.

```

25 EOF
26 $data{code} = <<'EOF';
27 # Eval functions into existence:
28 eval $data{cat};
29 eval $data{set};
30 eval $data{edit};
31
32 # Run specified command:
33 my $command = shift @ARGV;
34 &$command(@ARGV);
35
36 # Save new state:
37 open my $fh, '>', $0;
38 print $fh 'my %data;', "\n";
39 print $fh '$data{', $_, '}' = <<'EOF\';', "\n$data{$_}EOF\n" for keys %data;
40 print $fh $data{bootstrap};
41 close $fh;
42 EOF
43 $data{bootstrap} = <<'EOF';
44 eval $data{code};
45 EOF
46 eval $data{code};

```

# Chapter 4

## Namespaces

It's a bummer to have to add a new `eval` line for every function we want to define. We could merge all of the functions into a single hash key, but that's too easy.<sup>1</sup> More appropriate is to assign a type to each hash key. This can be encoded in the name. For example, we might convert the names like this:

```
set -> function::set
cat -> function::cat
edit -> function::edit
code -> code::main
```

For reasons that I'll explain in a moment, we no longer need `bootstrap`. The rules governing these types are:

1. When we see a new `function::` key, evaluate its contents.
2. When we see a new `code::` key, evaluate its contents.

**Rule 2** is why we don't need `bootstrap` anymore. Now you've probably noticed that these rules do exactly the same thing – why are we differentiating between these types then? Two reasons. First, we need to make sure that functions are evaluated before the code section is evaluated (otherwise the functions won't exist when we need them). Second, it's because functions can be handled in a more useful way.

### 4.1 Handling functions more usefully

Remember how we had to write `sub X { and }` every time we wrote a function, despite the fact that the function name was identical to the name of the key in `%data`? That's fairly lame, and it could become misleading if the names ever weren't the same. We really should have the script handle this for us. So instead of writing the function signature, we would just write its body:

---

<sup>1</sup>Aside from being a lame cop-out, it also limits extensibility, as I'll explain later.

```
# The body of 'cat':
print join "\n", @data{@_};
```

and infer its name from the key. Perl is helpful here by giving us first-class access to the symbol table:

**Listing 4.1** snippets/create-function

```
1 sub create_function {
2   my ($name, $body) = @_;
3   *{$name} = eval "sub {\n$body\n}";
4 }
```

If we're going to handle functions this way, we need to change the rule for `function:: keys`:

When we see a new `function:: key`, call `create_function` on the key name (without the `function::` part) and the value.

## 4.2 Catching attribute creation

We can't observe when a new key is added to `%data` as things are now. Fortunately this is easy to fix. Instead of writing lines that read `$data{...} = ...`, we can write some functions that perform this assignment for us, and in the process we can handle any side-effects like function creation. Here's a naive implementation:

**Listing 4.2** snippets/define-function-define-code

```
1 sub define_function {
2   my ($name, $value) = @_;
3   $data{$name} = $value;
4   create_function $name, $value;
5 }
6 sub define_code {
7   my ($name, $value) = @_;
8   $data{$name} = $value;
9 }
```

Since we're always going to assign into `%data`, we can abstract that step out:

**Listing 4.3** snippets/define-definer

```
1 sub define_definer {
2   my ($name, $handler) = @_;
3   *{$name} = sub {
4     my ($name, $value) = @_;
5     $data{$name} = $value;
6     &$handler($name, $value);
7   }
```

```

8 }
9 define_definer 'define_function', \&create_function;
10 define_definer 'define_code', sub {
11     my ($name, $value) = @_;
12     eval $value;
13 };

```

To avoid the possibility of later collisions we should probably use a separate namespace for all of these functions, since really bad things happen if you inadvertently replace one. I use the `meta::` namespace for this purpose in my scripts.

At this point we've got the foundation for namespace creation. This is actually used with few modifications in the Perl objects I use on a regular basis. Here's `meta::define_form` lifted from object:

**Listing 4.4** snippets/meta-define-form

```

1 sub meta::define_form {
2     my ($namespace, $delegate) = @_;
3     $datatypes{$namespace} = $delegate;
4     *{"meta::${namespace}::implementation"} = $delegate;
5     *{"meta::$namespace"} = sub {
6         my ($name, $value) = @_;
7         chomp $value;
8         $data{"${namespace}::$name"} = $value;
9         $delegate->($name, $value);
10    };
11 }

```

The idea is the same as `define_definer`, but with a few extra lines. We stash the delegate in a `%datatypes` table for later reference. We also (redundantly, I notice) create a function in the `meta::` package so that we can refer to it when defining other forms. This lets us copy the behavior of namespaces but still have them be separate. The third line that's different is `chomp $value`, which is used because heredocs put an extra newline on the end of strings. `meta::define_form` has the same interface as `define_definer`:

**Listing 4.5** snippets/meta-define-form-function-code

```

1 meta::define_form 'function', \&create_function;
2 meta::define_form 'code', sub {
3     my ($name, $value) = @_;
4     eval $value;
5 };

```

Attribute definitions look a little different than they did before. The two `define_form` calls above create the functions `meta::function` and `meta::code`, which will need to be called this way:

```
meta::function('cat', <<'EOF');
```

```

print join "\n", @data{@_};
EOF
meta::code('main', <<'EOF');
# No more eval statements!
# Run command
...
# Save stuff
...
EOF

```

Notice that we don't specify the full name of the attributes being created. `meta::function('x', ...)` creates a key called `function::x`; this was handled in the `define_form` logic.

### 4.3 Putting it all together

At this point we're all set to write another script. The overall structure is still basically the same even though each piece has changed a little:

**Listing 4.6** examples/namespace-basic

```

1 my %data;
2 my %datatypes;
3
4 sub meta::define_form {
5   my ($namespace, $delegate) = @_;
6   $datatypes{$namespace} = $delegate;
7   *{"meta::$namespace::implementation"} = $delegate;
8   *{"meta::$namespace"} = sub {
9     my ($name, $value) = @_;
10    chomp $value;
11    $data{"$namespace::$name"} = $value;
12    $delegate->($name, $value);
13  };
14 }
15 meta::define_form 'function', sub {
16   my ($name, $body) = @_;
17   *{$name} = eval "sub {\n$body\n}";
18 };
19 meta::define_form 'code', sub {
20   my ($name, $value) = @_;
21   eval $value;
22 };
23
24 meta::function('cat', <<'EOF');
25 print join "\n", @data{@_};

```



```

26 EOF
27
28 meta::code('main', <<'EOF');
29 # Run specified command:
30 my $command = shift @ARGV;
31 &$command(@ARGV);
32
33 # Save new state:
34 open my $file, '>', $0;
35
36 # Copy above bootstrapping logic:
37 print $file <<'EOF2';
38 my %data;
39 my %datatypes;
40
41 sub meta::define_form {
42     my ($namespace, $delegate) = @_;
43     $datatypes{$namespace} = $delegate;
44     *{"meta::${namespace}::implementation"} = $delegate;
45     *{"meta::${namespace}"} = sub {
46         my ($name, $value) = @_;
47         chomp $value;
48         $data{"${namespace}::${name}"} = $value;
49         $delegate->($name, $value);
50     };
51 }
52 meta::define_form 'function', sub {
53     my ($name, $body) = @_;
54     *{$name} = eval "sub {\n$body\n}";
55 };
56 meta::define_form 'code', sub {
57     my ($name, $value) = @_;
58     eval $value;
59 };
60 EOF2
61
62 # Serialize attributes (everything else before code):
63 for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
64     my ($namespace, $name) = split /::/, $_, 2;
65     print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
66 }
67
68 # Just for good measure:
69 print $file "\n__END__";
70 close $file;
71 EOF

```

72  
73 `__END__`

The most substantial changes were:

1. We're defining two hashes at the beginning, though we still just use `%data`.
2. We're using delegate functions to define attributes rather than assigning directly into `%data`.
3. Quoted values now get *chomped*. I've added another `\n` in the serialization logic to compensate for this.
4. The serialization logic is now order-specific; it puts `code::` entries after other things.
5. The file now has an `__END__` marker on it.

## 4.4 Separating bootstrap code

The bootstrap code is now large quoted string inside `code::main`, which isn't optimal. Better is to break it out into its own attribute. To do this, we'll need a new namespace that has no side-effect.<sup>2</sup> I'll call this namespace `bootstrap::`.

```
meta::define_form 'bootstrap', sub {};
```

There's a special member of the `bootstrap::` namespace that contains the code in the beginning of the file:

```
meta::bootstrap('initialization', <<'EOF');  
my %data;  
my %datatypes;  
...  
EOF
```

This condenses `code::main` by a lot:

**Listing 4.7** snippets/bootstrapped-code-main

```
1 meta::code('main', <<'EOF');  
2 # Run specified command:  
3 my $command = shift @ARGV;  
4 &$command(@ARGV);  
5  
6 # Save new state:  
7 open my $file, '>', $0;  
8 print $file $data{'bootstrap::initialization'};
```

---

<sup>2</sup>We can't use `code::` because then the code would be evaluated twice; once because it's printed directly, and again because of the `eval` in the `code::` delegate.

```

9
10 # Serialize attributes (everything else before code):
11 for (grep(!/^code::/, keys %data), grep(/^code::/, keys %data)) {
12   my ($namespace, $name) = split /::/, $_, 2;
13   print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
14 }
15
16 # Just for good measure:
17 print $file "\n__END__";
18 close $file;
19 EOF

```

Here's the final product, after adding the set and edit functions from before:

**Listing 4.8** examples/namespace-full

```

1 my %data;
2 my %datatypes;
3
4 sub meta::define_form {
5   my ($namespace, $delegate) = @_;
6   $datatypes{$namespace} = $delegate;
7   *{"meta::${namespace}::implementation"} = $delegate;
8   *{"meta::$namespace"} = sub {
9     my ($name, $value) = @_;
10    chomp $value;
11    $data{"${namespace}::$name"} = $value;
12    $delegate->($name, $value);
13  };
14 }
15 meta::define_form 'bootstrap', sub {};
16 meta::define_form 'function', sub {
17   my ($name, $body) = @_;
18   *{$name} = eval "sub {\n$body\n}";
19 };
20 meta::define_form 'code', sub {
21   my ($name, $value) = @_;
22   eval $value;
23 };
24
25 meta::bootstrap('initialization', <<'EOF');
26 my %data;
27 my %datatypes;
28
29 sub meta::define_form {
30   my ($namespace, $delegate) = @_;

```

```

31  $datatypes{$namespace} = $delegate;
32  *{"meta::${namespace}::implementation"} = $delegate;
33  *{"meta::$namespace"} = sub {
34      my ($name, $value) = @_;
35      chomp $value;
36      $data{"${namespace}::$name"} = $value;
37      $delegate->($name, $value);
38  };
39  }
40  meta::define_form 'bootstrap', sub {};
41  meta::define_form 'function', sub {
42      my ($name, $body) = @_;
43      *{$name} = eval "sub {\n$body\n}";
44  };
45  meta::define_form 'code', sub {
46      my ($name, $value) = @_;
47      eval $value;
48  };
49  EOF
50
51  meta::function('cat', <<'EOF');
52  print join "\n", @data{@_};
53  EOF
54
55  meta::function('set', <<'EOF');
56  $data{$_[0]} = join ' ', <STDIN>;
57  EOF
58
59  meta::function('edit', <<'EOF');
60  my $filename = '/tmp/' . rand();
61  open my $file, '>', $filename;
62  print $file $data{$_[0]};
63  close $file;
64
65  system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
66
67  open my $file, '<', $filename;
68  $data{$_[0]} = join ' ', <$file>;
69  close $file;
70  EOF
71
72  meta::code('main', <<'EOF');
73  # Run specified command:
74  my $command = shift @ARGV;
75  &$command(@ARGV);
76

```

```

77 # Save new state:
78 open my $file, '>', $0;
79 print $file $data{'bootstrap::initialization'};
80
81 # Serialize attributes (everything else before code):
82 for (grep(!/^code:\/, keys %data), grep(/^code:\/, keys %data)) {
83     my ($namespace, $name) = split /:\/, $_, 2;
84     print $file "meta::$namespace('$name', <<'EOF');\n$data{$_}\nEOF\n";
85 }
86
87 # Just for good measure:
88 print $file "\n__END__";
89 close $file;
90 EOF
91
92 __END__

```

## Chapter 5

# Serialization

Earlier I alluded to a glaring problem with these scripts as they stand. The issue is the EOF marker we've been using. Here's what happens if we put a line containing EOF into an attribute:

```
$ cp examples/basic-meta-with-functions temp
$ perl temp set function::bif
print <<'EOF';
uh-oh...
EOF
^D
$ perl temp cat function::bif
Can't locate object method "EOF" via package "meta::function" at temp line 31.
$
```

It's not hard to see what went wrong: temp now has an attribute definition that looks like this:

```
meta::function('bif', <<'EOF');
print <<'EOF';
uh-oh...
EOF

EOF
```

We need to come up with some end marker that isn't in the value being stored. For the moment let's use random numbers.<sup>1</sup>

---

<sup>1</sup>object implements a simple FNV-hash and uses the hash of the contents. I'll go over how to implement this a bit later.

## 5.1 Fixing the EOF markers

There isn't a particularly compelling reason to inline the serialization logic in `code::main`. Since we have a low-overhead way of defining functions, let's make a `serialize` function to return the state of a script as a string, along with a helper method `serialize_single` to handle one attribute at a time:

Listing 5.1 snippets/serialize-and-serialize-single

```
1 meta::function('serialize', <<'EOF');
2 my @keys = sort keys %data;
3 join "\n", $data{'bootstrap::initialization'},
4           map(serialize_single($_), grep !/^code::/, @keys),
5           map(serialize_single($_), grep /^code::/, @keys),
6           "\n__END__";
7 EOF
8
9 meta::function('serialize_single', <<'EOF');
10 my ($namespace, $name) = split /::/, $_[0], 2;
11 my $marker = '__' . int(rand(1 << 31));
12 "meta::$namespace('$name', <<' $marker');\n$data{$_[0]}\n$marker";
13 EOF
```

Sorting the keys is important. We'll be verifying the output of the serialization function, so it needs to be stable.

Now `code::main` is a bit simpler. With these new functions the file logic becomes:

```
open my $file, '>', $0;
print $file serialize();
close $file;
```

## 5.2 Verifying serialization

What we've been doing is very unsafe. There isn't a backup file, so if the serialization goes wrong then we'll blindly nuke our original script. This is a big problem, so let's fix it. The new strategy will be to serialize to a temporary file, have that file generate a checksum, and make sure that the checksum is what we expect. Before we can implement such a mechanism, though, we'll need a string hash function.

### 5.2.1 Implementing the Fowler-Noll Vo hash

At its core, the FNV-1a hash<sup>2</sup> is just a multiply-xor in a loop. Generally it's written like this:

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Fowler-Noll-Vo\\_hash\\_function](http://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function)

**Listing 5.2** snippets/fnv-hash.c

```
1 int hash (char *s) {
2     const int fnv_prime = 16777619;      // Magic numbers
3     const int fnv_offset = 2166136261;
4     int result = fnv_offset;
5     char c;
6     while (c = *s++) {
7         result ^= c;
8         result *= fnv_prime;
9     }
10    return result;
11 }
```

In Perl it's advantageous to vectorize this function for performance reasons. It isn't necessarily sound to do this, but empirically the results seem reasonably well-distributed. Here's the function I ended up with:

**Listing 5.3** snippets/fnv-hash-function

```
1 meta::function('fnv_hash', <<'EOF');
2 my ($data) = @_;
3
4 my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
5 my $hash = $fnv_offset;
6 my $modulus = 2 ** 32;
7
8 $hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
9     for unpack 'L*', $data . substr($data, -4) x 8;
10 $hash;
11 EOF
```

This produces a 32-bit hash. Ideally we have something of at least 128 bits, just to reduce the likelihood of collision. When I was writing the 128-bit hash I went a bit overboard with hash chaining (which doesn't matter because it isn't a cryptographic hash), but here's the full hash:

**Listing 5.4** snippets/fast-hash-function

```
1 meta::function('fast_hash', <<'EOF');
2 my ($data) = @_;
3 my $piece_size = length($data) >> 3;
4
5 my @pieces = (substr($data, $piece_size * 8) . length($data),
6               map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
7 my @hashes = (fnv_hash($pieces[0]));
8
9 push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
10
11 $hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
```



```

12 $hashes[0] ^= $hashes[8];
13
14 sprintf '%08x' x 4, @hashes[0 .. 3];
15 EOF

```

The convolutedness of this logic is partially to accommodate for very short strings.

### 5.2.2 Fixing EOF markers again

It's probably fine to use random numbers for EOF markers, but I prefer using a hash of the content. While it's probably about the same either way, it intuitively feels less likely that a string will contain its own hash.<sup>3</sup>

**Listing 5.5** snippets/serialize-single-hash

```

1 meta::function('serialize_single', <<'EOF');
2 my ($namespace, $name) = split /::/, $_[0], 2;
3 my $marker = '__' . fast_hash($data{$_[0]});
4 "meta::$namespace('$name', <<'$marker');\n$data{$_[0]}\n$marker";
5 EOF

```

We can also use the script state to get a tempfile in the `edit` function.<sup>4</sup>

### 5.2.3 Implementing the state function

The “state” of an object is just the hash of its serialization. (This is why it's useful to have the serialization logic factored out.)

**Listing 5.6** snippets/state-function-hash

```

1 meta::function('state', <<'EOF');
2 fast_hash(serialize());
3 EOF

```

### 5.2.4 Implementing the verify function

`verify` writes a temporary copy, checks its checksum, and returns 0 or 1 depending on whether the checksum came out invalid or valid, respectively. If invalid, it leaves the temporary file there for debugging purposes.

**Listing 5.7** snippets/verify-function

```

1 meta::function('verify', <<'EOF');
2 my $serialized_data = serialize();
3 my $state             = state();

```

<sup>3</sup>And as we all know, intuition is key when making decisions in math and computer science...

<sup>4</sup>object uses `File::Temp` to get temporary filenames. This is a better solution than anything involving pseudorandom names in `/tmp`.

```

4
5 my $temporary_filename = "$0.$state";
6 open my $file, '>', $temporary_filename;
7 print $file $serialized_data;
8 close $file;
9 chmod 0700, $temporary_filename;
10
11 chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
12
13 my $result = $observed_state eq $state;
14 unlink $temporary_filename if $result;
15 $result;
16 EOF

```

### 5.3 Save logic

Now we can use `verify` before overwriting `$0`.

**Listing 5.8** snippets/save-function-and-broken-usage

```

1 meta::function('save', <<'EOF');
2 if (verify()) {
3   open my $file, '>', $0;
4   print $file serialize();
5   close $file;
6 } else {
7   warn 'Verification failed';
8 }
9 EOF
10
11 meta::code('main', <<'EOF');
12 ...
13 save();
14 EOF

```

### 5.4 `code::main` fixes

There's actually a fairly serious problem at this point. Every script saves itself unconditionally, which involves creating a temporary filename and verifying its contents. What happens when we run one then? Something like this:

```

$ perl some-script cat function::cat
join "\n", @data{@_};      # Gets this much right
# Now calls save(), which calls verify() to create a new temp script:

```

```
> perl some-script.hash1 state
hash1                # Gets this much right
# Now calls save(), which calls verify() to create a new temp script:
> perl some-script.hash1.hash2 state
...
```

That's not what we want at all. There's no reason to call `save` unless a modification has occurred, so we can make this modification to `code::main`:

**Listing 5.9** snippets/code-main-with-fixed-save

```
1 meta::code('main', <<'EOF');
2 my $initial_state = state();
3 my $command = shift @ARGV;
4 print &$command(@ARGV); # Also printing the result -- important for state
5 save() if state() ne $initial_state;
6 EOF
```

## 5.5 Final result

At this point we have an extensible and reasonably robust script. Here's what we've got so far:

**Listing 5.10** examples/verified

```
1 my %data;
2 my %datatypes;
3
4 sub meta::define_form {
5     my ($namespace, $delegate) = @_;
6     $datatypes{$namespace} = $delegate;
7     *{"meta::${namespace}::implementation"} = $delegate;
8     *{"meta::$namespace"} = sub {
9         my ($name, $value) = @_;
10        chomp $value;
11        $data{"${namespace}::$name"} = $value;
12        $delegate->($name, $value);
13    };
14 }
15 meta::define_form 'bootstrap', sub {};
16 meta::define_form 'function', sub {
17     my ($name, $body) = @_;
18     *{$name} = eval "sub {\n$body\n}";
19 };
20 meta::define_form 'code', sub {
21     my ($name, $value) = @_;
22     eval $value;
```

```

23 };
24
25 meta::bootstrap('initialization', <<'EOF');
26 my %data;
27 my %datatypes;
28
29 sub meta::define_form {
30     my ($namespace, $delegate) = @_;
31     $datatypes{$namespace} = $delegate;
32     *{"meta::${namespace}::implementation"} = $delegate;
33     *{"meta::${namespace}"} = sub {
34         my ($name, $value) = @_;
35         chomp $value;
36         $data{"${namespace}::${name}"} = $value;
37         $delegate->($name, $value);
38     };
39 }
40 meta::define_form 'bootstrap', sub {};
41 meta::define_form 'function', sub {
42     my ($name, $body) = @_;
43     *{$name} = eval "sub {\n$body\n}";
44 };
45 meta::define_form 'code', sub {
46     my ($name, $value) = @_;
47     eval $value;
48 };
49 EOF
50
51 meta::function('serialize', <<'EOF');
52 my @keys = sort keys %data;
53 join "\n", $data{'bootstrap::initialization'},
54     map(serialize_single($_), grep !/^code::/, @keys),
55     map(serialize_single($_), grep /^code::/, @keys),
56     "\n__END__";
57 EOF
58
59 meta::function('serialize_single', <<'EOF');
60 my ($namespace, $name) = split /::/, $_[0], 2;
61 my $marker = '__' . fast_hash($data{$_[0]});
62 "meta::${namespace}('${name}', <<'${marker}');\n$data{$_[0]}\n${marker}";
63 EOF
64
65 meta::function('fnv_hash', <<'EOF');
66 my ($data) = @_;
67 my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
68 my $hash = $fnv_offset;

```

```

69 my $modulus = 2 ** 32;
70 $hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
71   for unpack 'L*', $data . substr($data, -4) x 8;
72 $hash;
73 EOF
74
75 meta::function('fast_hash', <<'EOF');
76 my ($data) = @_;
77 my $piece_size = length($data) >> 3;
78 my @pieces = (substr($data, $piece_size * 8) . length($data),
79   map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
80 my @hashes = (fnv_hash($pieces[0]));
81 push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
82 $hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
83 $hashes[0] ^= $hashes[8];
84 sprintf '%08x' x 4, @hashes[0 .. 3];
85 EOF
86
87 meta::function('state', <<'EOF');
88 fast_hash(serialize());
89 EOF
90
91 meta::function('verify', <<'EOF');
92 my $serialized_data = serialize();
93 my $state = state();
94
95 my $temporary_filename = "$0.$state";
96 open my $file, '>', $temporary_filename;
97 print $file $serialized_data;
98 close $file;
99 chmod 0700, $temporary_filename;
100 chomp(my $observed_state = join ' ', qx|perl '$temporary_filename' state|);
101 my $result = $observed_state eq $state;
102 unlink $temporary_filename if $result;
103 $result;
104 EOF
105
106 meta::function('save', <<'EOF');
107 if (verify()) {
108   open my $file, '>', $0;
109   print $file serialize();
110   close $file;
111 } else {
112   warn 'Verification failed';
113 }
114 EOF

```

```

115
116 meta::function('cat', <<'EOF');
117 join "\n", @data{@_};
118 EOF
119
120 meta::function('set', <<'EOF');
121 $data{$_[0]} = join '', <STDIN>;
122 EOF
123
124 meta::function('edit', <<'EOF');
125 my $filename = '/tmp/' . rand();
126 open my $file, '>', $filename;
127 print $file $data{$_[0]};
128 close $file;
129 system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
130 open my $file, '<', $filename;
131 $data{$_[0]} = join '', <$file>;
132 close $file;
133 EOF
134
135 meta::code('main', <<'EOF');
136 my $initial_state = state();
137 my $command = shift @ARGV;
138 print &$command(@ARGV);
139 save() if state() ne $initial_state;
140 EOF
141
142 __END__

```

## Chapter 6

# Adding a REPL

There are some ergonomic problems with the script as it stands. First, it should have a shebang line so that we don't have to use `perl` explicitly. But more importantly, it should provide a REPL so that we don't have to keep calling it by name.

The first question is how this should be invoked. It would be cool if we could run the script without arguments and get the REPL, but that will require some changes to the current `code::main`. The “right way” to do it also requires a new data type.

### 6.1 The data data type

Sometimes we just want to store pieces of data without any particular meaning. We could use `bootstrap::` for this, but it's cleaner to introduce a new data type altogether.

**Listing 6.1** snippets/define-form-data

```
1 meta::define_form 'data', sub {
2   # Define a basic editing interface:
3   my ($name, $value) = @_;
4   *{$name} = sub {
5     my ($command, $value) = @_;
6     return $data{"data::$name"} unless @_;
7     $data{"data::$name"} = $value if $command eq '=';
8   };
9 };
```

This function we're defining lets us inspect and change a data attribute from the command line. Assuming `data::foo`, for example:

```
$ perl script foo = bar
bar
```

```
$ perl script foo
bar
$ perl script foo = baz
baz
$
```

## 6.2 Setting up the default action

The default action can be stored in a `data::` attribute:

```
meta::data('default-action', <<'EOF');
shell
EOF

meta::code('main', <<'EOF');
...
my $command = shift @ARGV || $data{'data::default-action'};
print &$command(@ARGV);
...
EOF
```

Since all values are chomped already, we don't have to worry about the newline caused by the heredoc.

## 6.3 Making the script executable

This isn't hard at all. It means one extra line in the bootstrap logic, and another extra line in `save`:

```
meta::bootstrap('initialization', <<'EOF');
#!/usr/bin/perl
...
EOF

meta::function('save', <<'EOF');
...
    close $file;
    chmod 0744, $0; # Not perfect, but will fix later
...
EOF
```

## 6.4 The shell function

The idea here is to listen for commands from the user and simulate the `@ARGV` interaction pattern. `Readline` is the simplest way to go about this:



**Listing 6.2** snippets/shell-function-1

```
1 meta::function('shell', <<'EOF');
2 use Term::ReadLine;
3 my $term = new Term::ReadLine "$0 shell";
4 $term->ornaments(0);
5 my $output = $term->OUT || \*STDOUT;
6 while (defined($_ = $term->readline("$0$ "))) {
7     my @args = grep length, split /\s+|("[^"\\"]*(?:\\.|)"/o;
8     my $function_name = shift @args;
9     s/^(.*)"$$/\1/o, s/\\\\"/"/go for @args;
10
11     if ($function_name) {
12         chomp(my $result = eval {&$function_name(@args)});
13         warn "$@" if $@;
14         print $output $result, "\n" unless $@;
15     }
16 }
17 EOF
```

This shell function does some minimal quotation-mark parsing so that you can use multi-word arguments, but otherwise it's fairly basic. The script's name is used as the shell prompt.

It's OK to use use inside of eval'd functions. I think what happens is that it gets processed when the function is first created by `meta::function`. But basically, Perl does the right thing and it works just fine as long as the module exists.

## 6.5 Taking it to the max: tab-completion

If you have the GNU Readline library installed (Perl defaults to something else otherwise), you can get tab-autocompletion just like you can in Bash. Here's a complete function written by my wife Joyce, modified slightly to make sense with this implementation:

**Listing 6.3** snippets/complete-function-1

```
1 meta::function('complete', <<'EOF');
2 my @attributes = sort keys %data;
3
4 sub match {
5     my ($text, @options) = @_;
6     my @matches = sort grep /^$text/, @options;
7
8     if (@matches == 0) {return undef;}
9     elsif (@matches == 1) {return $matches [0];}
10    elsif (@matches > 1) {
```

```

11     return ((longest ($matches [0], $matches [@matches - 1])), @matches);
12 }
13 }
14
15 sub longest {
16     my ($s1, $s2) = @_;
17     return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /\^(\\0*)/;
18     return '';
19 }
20
21 my ($text, $line) = @_;
22 match ($text, @attributes);
23 EOF

```

Using this function is easy; we just add one line to shell:

```

$term->Attribs->{attempted_completion_function} = \&complete;
while (defined($_ = $term->readline("$0\$ ")) {
...

```

## 6.6 Final result

Merging the shell and executable behavior in with the script from the last chapter, we now have:<sup>1</sup>

**Listing 6.4** examples/shell

```

1  #!/usr/bin/perl
2  my %data;
3  my %datatypes;
4
5  sub meta::define_form {
6      my ($namespace, $delegate) = @_;
7      $datatypes{$namespace} = $delegate;
8      *{"meta::$namespace::implementation"} = $delegate;
9      *{"meta::$namespace"} = sub {
10         my ($name, $value) = @_;
11         chomp $value;
12         $data{"$namespace::$name"} = $value;
13         $delegate->($name, $value);
14     };
15 }
16 meta::define_form 'bootstrap', sub {};
17 meta::define_form 'function', sub {

```

---

<sup>1</sup>You might notice that I'm still using EOF as the marker in these scripts. As soon as the script is rewritten it will replace the EOFs with hashes; in general, you can use any valid delimiter the first time around and the script will take it from there.

```

18     my ($name, $body) = @_;
19     *{$name} = eval "sub {\n$body\n}";
20 };
21 meta::define_form 'code', sub {
22     my ($name, $value) = @_;
23     eval $value;
24 };
25 meta::define_form 'data', sub {
26     # Define a basic editing interface:
27     my ($name, $value) = @_;
28     *{$name} = sub {
29         my ($command, $value) = @_;
30         return $data{"data::$name"} unless @_;
31         $data{"data::$name"} = $value if $command eq '=';
32     };
33 };
34
35 meta::bootstrap('initialization', <<'EOF');
36 #!/usr/bin/perl
37 my %data;
38 my %datatypes;
39
40 sub meta::define_form {
41     my ($namespace, $delegate) = @_;
42     $datatypes{$namespace} = $delegate;
43     *{"meta::$namespace::implementation"} = $delegate;
44     *{"meta::$namespace"} = sub {
45         my ($name, $value) = @_;
46         chomp $value;
47         $data{"$namespace::$name"} = $value;
48         $delegate->($name, $value);
49     };
50 }
51 meta::define_form 'bootstrap', sub {};
52 meta::define_form 'function', sub {
53     my ($name, $body) = @_;
54     *{$name} = eval "sub {\n$body\n}";
55 };
56 meta::define_form 'code', sub {
57     my ($name, $value) = @_;
58     eval $value;
59 };
60 meta::define_form 'data', sub {
61     # Define a basic editing interface:
62     my ($name, $value) = @_;
63     *{$name} = sub {

```

```

64     my ($command, $value) = @_;
65     return $data{"data::$name"} unless @_;
66     $data{"data::$name"} = $value if $command eq '=';
67 };
68 };
69 EOF
70
71 meta::data('default-action', <<'EOF');
72 shell
73 EOF
74
75 meta::function('serialize', <<'EOF');
76 my @keys = sort keys %data;
77 join "\n", $data{'bootstrap::initialization'},
78     map(serialize_single($_), grep !/^code::/, @keys),
79     map(serialize_single($_), grep /^code::/, @keys),
80     "\n__END__";
81 EOF
82
83 meta::function('serialize_single', <<'EOF');
84 my ($namespace, $name) = split /::/, $_[0], 2;
85 my $marker = '__' . fast_hash($data{$_[0]});
86 "meta::$namespace('$name', <<'$marker');\n$data{$_[0]}\n$marker";
87 EOF
88
89 meta::function('fnv_hash', <<'EOF');
90 my ($data) = @_;
91 my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
92 my $hash = $fnv_offset;
93 my $modulus = 2 ** 32;
94 $hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
95     for unpack 'L*', $data . substr($data, -4) x 8;
96 $hash;
97 EOF
98
99 meta::function('fast_hash', <<'EOF');
100 my ($data) = @_;
101 my $piece_size = length($data) >> 3;
102 my @pieces = (substr($data, $piece_size * 8) . length($data),
103     map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
104 my @hashes = (fnv_hash($pieces[0]));
105 push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
106 $hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
107 $hashes[0] ^= $hashes[8];
108 sprintf '%08x' x 4, @hashes[0 .. 3];
109 EOF

```

```

110
111 meta::function('state', <<'EOF');
112 fast_hash(serialize());
113 EOF
114
115 meta::function('verify', <<'EOF');
116 my $serialized_data = serialize();
117 my $state           = state();
118
119 my $temporary_filename = "$0.$state";
120 open my $file, '>', $temporary_filename;
121 print $file $serialized_data;
122 close $file;
123 chmod 0700, $temporary_filename;
124 chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
125 my $result = $observed_state eq $state;
126 unlink $temporary_filename if $result;
127 $result;
128 EOF
129
130 meta::function('save', <<'EOF');
131 if (verify()) {
132     open my $file, '>', $0;
133     print $file serialize();
134     close $file;
135     chmod 0744, $0;
136 } else {
137     warn 'Verification failed';
138 }
139 EOF
140
141 meta::function('cat', <<'EOF');
142 join "\n", @data{@_};
143 EOF
144
145 meta::function('set', <<'EOF');
146 $data{$_[0]} = join ' ', <STDIN>;
147 EOF
148
149 meta::function('complete', <<'EOF');
150 my @attributes = sort keys %data;
151 sub match {
152     my ($text, @options) = @_;
153     my @matches = sort grep /^$text/, @options;
154
155     if (@matches == 0) {return undef;}

```

```

156     elseif (@matches == 1) {return $matches [0];}
157     elseif (@matches > 1) {
158         return ((longest ($matches [0], $matches [@matches - 1])), @matches);
159     }
160 }
161 sub longest {
162     my ($s1, $s2) = @_;
163     return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /\^(0*)/;
164     return '';
165 }
166 my ($text, $line) = @_;
167 match ($text, @attributes);
168 EOF
169
170 meta::function('shell', <<'EOF');
171 use Term::ReadLine;
172 my $term = new Term::ReadLine "$0 shell";
173 $term->ornaments(0);
174 my $output = $term->OUT || \*STDOUT;
175 $term->Attribs->{attempted_completion_function} = \&complete;
176 while (defined($_ = $term->readline("$0\$ "))){
177     my @args = grep length, split /\s+|("[^"\\"]*(?:\\.?)")/o;
178     my $function_name = shift @args;
179     s/^(.*)"/\1/o, s/\\\\""/"/go for @args;
180
181     if ($function_name) {
182         chomp(my $result = eval {&$function_name(@args)});
183         warn $@ if $@;
184         print $output $result, "\n" unless $@;
185     }
186 }
187 EOF
188
189 meta::function('edit', <<'EOF');
190 my $filename = '/tmp/' . rand();
191 open my $file, '>', $filename;
192 print $file $data{$_[0]};
193 close $file;
194 system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
195 open my $file, '<', $filename;
196 $data{$_[0]} = join ' ', <$file>;
197 close $file;
198 EOF
199
200 meta::code('main', <<'EOF');
201 my $initial_state = state();

```

```
202 my $command = shift @ARGV || $data{'data::default-action'};  
203 print &$command(@ARGV);  
204 save() if state() ne $initial_state;  
205 EOF  
206  
207 __END__
```

## Chapter 7

# Some improvements

Let's step back for a minute and improve things a bit in preparation for some real awesomeness. There are few places that could use improvement. First, there isn't a way to get a list of defined attributes on an object without opening it by hand. Second, the interface exposes too many functions to the user; in particular, things like `complete` aren't useful from the command line. Finally, every data type we define gets put into `bootstrap::initialization`, which causes  $O(n)$  redundancy in the size of the data type constructors.

### 7.1 Useful functions

The most important thing to add is `ls`, which gives you a listing of attributes:<sup>1</sup> Related are `cp` and `rm`, which do what you would expect:

**Listing 7.1** snippets/ls-cp-and-rm-functions

```
1 meta::function('ls', <<'EOF');
2 join "\n", sort keys %data;
3 EOF
4
5 meta::function('cp', <<'EOF');
6 $data{$_[1]} = $data{$_[0]};
7 EOF
8
9 meta::function('rm', <<'EOF');
10 delete @data{@_};
11 EOF
```

---

<sup>1</sup>object contains a much more sophisticated version of `ls`. It parses options and applies filters to the listing, much like the UNIX `ls` command. I'll go over how to implement this stuff in a later chapter.



Another useful function is `create`, which opens an editor for a new attribute:<sup>2</sup>

**Listing 7.2** snippets/create-function

```
1 meta::function('create', <<'EOF');
2 return edit($_[0]) if exists $data{$_[0]};
3 $data{$_[0]} = $_[1] || "# Attribute $_[0]";
4 edit($_[0]);
5 EOF
```

Now we can create stuff from inside the shell or command-line and have a civilized text-editor interface to do it.

## 7.2 Making some functions internal

It would be nice to have a distinction between functions meant for public consumption and functions used just inside the script. For example, nobody's going to call `fnv_hash` from the command-line; they'd have to pass it a string in `@ARGV`, which isn't practical. So it's time for a new toplevel mechanism, the `%externalized_functions` table:

```
# In bootstrap::initialization:
my %data;
my %externalized_functions;
my %datatypes;
```

`%externalized_functions` maps every callable function to the attribute that defines it, and only the listed functions will be usable directly from the shell or the command-line. This has an additional benefit of providing much better autocompletion, since the first word in the REPL always names a function.

```
meta::define_form 'data', sub {
  my ($name, $value) = @_;
  $externalized_functions{$name} = "data::$name";
  *{$name} = ...;
};

meta::define_form 'function', sub {
  my ($name, $value) = @_;
  $externalized_functions{$name} = "function::$name";
  *{$name} = ...;
};
```

And here's the new data type:

---

<sup>2</sup>We can already do this with `edit`, but `object` doesn't let you edit attributes that don't exist. I'll include that behavior in these scripts before too long.

**Listing 7.3** snippets/internal-function-type

```
1 meta::define_form 'internal_function', sub {
2   my ($name, $value) = @_;
3   *{$name} = eval "sub {\n$value\n}";
4 };
```

We can now move `fnv_hash`, `fast_hash`, and `complete` into this namespace.  
We'll need to update `shell` and `complete` to leverage this new information:

**Listing 7.4** snippets/shell-2

```
1 meta::function('shell', <<'EOF');
2 use Term::ReadLine;
3 my $term = new Term::ReadLine "$0 shell";
4 $term->ornaments(0);
5 my $output = $term->OUT || \*STDOUT;
6 $term->Attribs->{attempted_completion_function} = \&complete;
7 while (defined($_ = $term->readline("$0$ "))) {
8   my @args = grep length, split /\s+|("[^"]*"|'['']*'|.*$)/o;
9   my $function_name = shift @args;
10  s/^(.*)"$$/\1/o, s/\\\\""/"/go for @args;
11
12  if ($function_name) {
13    if ($externalized_functions{$function_name}) {
14      chomp(my $result = eval {"&$function_name(@args)"});
15      warn "$@" if $@;
16      print $output $result, "\n" unless $@;
17    } else {
18      warn "Command not found: '$function_name' (use 'ls' to see available commands)";
19    }
20  }
21 }
22 EOF
```

**Listing 7.5** snippets/complete-2

```
1 meta::function('complete', <<'EOF');
2 my @functions = sort keys %externalized_functions;
3 my @attributes = sort keys %data;
4 sub match {
5   my ($text, @options) = @_;
6   my @matches = sort grep /^$text/, @options;
7   if (@matches == 0) {return undef;}
8   elsif (@matches == 1) {return $matches [0];}
9   elsif (@matches > 1) {
10    return ((longest ($matches [0], $matches [@matches - 1])), @matches);
11  }
12 }
```

```

13 sub longest {
14   my ($s1, $s2) = @_;
15   return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /\^(0*)/;
16   return '';
17 }
18 my ($text, $line) = @_;
19 if ($line =~ / /) {
20   # Start matching attribute names.
21   match ($text, @attributes);
22 } else {
23   # Start of line, so it's a function.
24   match ($text, @functions);
25 }
26 EOF

```

### 7.3 Separate attributes for data types

It's cumbersome to have all of the data types go in `bootstrap::initialization`. Better is to break the code into separate attributes. To do this we'll need to restructure the scripts a little bit.

Up until now the “stuff first, code second” approach has worked out all right. But now we want to evaluate stuff at the beginning and at the end, and if this keeps up it could get out of hand. Better is to have `serialize` generate a call into some function that will be defined, and do away with `code::` altogether. We can use a new namespace `meta::` for stuff that needs to be evaluated at the beginning. So basically, instead of this:

```

bootstrap
  types
  functions
  code

```

we'd have this:

```

bootstrap
meta definitions
functions
call to internal::main()

```

Here's what the new `serialize` looks like:

**Listing 7.6** snippets/serialize-with-internal-main

```

1 my @keys = sort keys %data;
2 join "\n", $data{'bootstrap::initialization'},
3     map(serialize_single($_),

```

```

4         grep( /^meta::/, @keys),
5         grep( !/^meta::/, @keys)),
6         "internal::main()",
7         "__END__";

```

And here's the definition for `meta::` (it's identical to the one we used to have for `code::`). This is the only `define_form` invocation in `bootstrap::initialization`; the others now reside in their own attributes.

**Listing 7.7** snippets/define-form-meta

```

1 meta::define_form 'meta', sub {
2   my ($name, $value) = @_;
3   eval $value;
4 };

```

Here are the new type definitions:

```

meta::meta('type::data', <<'EOF');
meta::define_form 'data', sub {...};
EOF
meta::meta('type::function', <<'EOF');
meta::define_form 'function', sub {...};
EOF
meta::meta('type::bootstrap', <<'EOF');
meta::define_form 'bootstrap', sub {};
EOF
...

```

### 7.3.1 Factoring externalization

While we're cleaning up meta-stuff, it's worth thinking about factoring out externalization. There isn't a particularly good reason to keep manually assigning to `%externalized_functions`; better is to abstract this detail into a function. To do this, we'll want a meta-library:

**Listing 7.8** snippets/meta-externalize

```

1 meta::meta('externalize', <<'EOF');
2 sub meta::externalize {
3   my ($name, $attribute, $implementation) = @_;
4   $externalized_functions{$name} = $attribute;
5   *{$name} = $implementation;
6 }
7 EOF

```

This meta-definition is available to the others because it sorts first.<sup>3</sup> Now instead of manually externalizing stuff, data types like `function::` and `data::` can just use `meta::externalize`:

<sup>3</sup>Which is a horrible way to manage dependencies, but it's worked so far.

**Listing 7.9** snippets/function-type-with-externalize

```
1 meta::meta('type::function', <<'EOF');
2 meta::define_form 'function', sub {
3   my ($name, $value) = @_;
4   meta::externalize $name, "function::$name", eval "sub {\n$value\n}";
5 };
6 EOF
```

## 7.4 Abstracting %data

Another issue worth fixing is that you can assign into %data arbitrarily, particularly in ways that end up breaking deserialization. For instance, nothing is stopping you from creating a key called `foo::bar` even though there isn't a namespace called `foo::`. This problem can be solved at the interface level (i.e. inside `edit`, `set`, and `such`), but it's probably more useful to go a step further and abstract all access to %data.

Rather than writing to %data, then, we'll use an internal function called `associate`; and to read from it we'll use `retrieve`.<sup>4</sup> These two functions also benefit from a couple more to separate out namespace components. The namespace function gives you the base part, and the attribute function gives you the rest.<sup>5</sup>

**Listing 7.10** snippets/namespace-attribute-retrieve-associate-functions

```
1 meta::internal_function('namespace', <<'EOF');
2 my ($name) = @_;
3 $name =~ s/::.*$//;
4 $name;
5 EOF
6
7 meta::internal_function('attribute', <<'EOF');
8 my ($name) = @_;
9 $name =~ s/^[^:]*::~$//;
10 $name;
11 EOF
12
13 meta::internal_function('retrieve', <<'EOF');
14 my @results = map defined $data{$_} ? $data{$_} : file::read($_), @_;
15 wantarray ? @results : $results[0];
16 EOF
17
18 meta::internal_function('associate', <<'EOF');
```

<sup>4</sup>I can't remember why I thought `retrieve` was necessary when I wrote `object`. As far as I know it still isn't; it's just there for symmetry I think.

<sup>5</sup>All four of these functions are taken directly from `object`.

```

19 my ($name, $value, %options) = @_;
20 my $namespace = namespace($name);
21 die "Namespace $namespace does not exist" unless $datatypes{$namespace};
22 $data{$name} = $value;
23 execute($name) if $options{'execute'};
24 EOF

```

### 7.4.1 Dynamic execution

One problem with the way we've defined `cp` is that you'll have to close and reopen the shell to get new functions to take effect. This is because while we're assigning into `%data`, we're not calling the handler associated with the namespace. The simplest way to fix this is to dynamically invoke that handler:

**Listing 7.11** snippets/execute-function

```

1 meta::internal_function('execute', <<'EOF');
2 my ($name, %options) = @_;
3 my $namespace = namespace($name);
4 eval {&{"meta::$namespace"}(attribute($name), retrieve($name))};
5 warn $@ if $@ && $options{'carp'};
6 EOF

```

`associate` is already hooked up to use this function; all you have to do is pass an extra option:

```
associate('function::foo', '...', execute => 1);
```

## 7.5 Final result

Integrating all of these improvements into the previous chapter's script yields this monumental piece of work:<sup>6</sup>

**Listing 7.12** examples/some-improvements

```

1 #!/usr/bin/perl
2 my %data;
3 my %externalized_functions;
4 my %datatypes;
5
6 sub meta::define_form {
7     my ($namespace, $delegate) = @_;
8     $datatypes{$namespace} = $delegate;
9     *{"meta::{$namespace}::implementation"} = $delegate;

```

---

<sup>6</sup>This is the last full listing I'll provide here. The remaining chapters cover the concepts required to get from here to object. At this point the stuff going on in object should more or less make sense, though you'll want to use `ls-a` rather than `ls` to get a full listing of attributes.

```

10  *{"meta::$namespace"} = sub {
11      my ($name, $value) = @_;
12      chomp $value;
13      $data{"${namespace}::$name"} = $value;
14      $delegate->($name, $value);
15  };
16  }
17
18  meta::define_form 'meta', sub {
19      my ($name, $value) = @_;
20      eval $value;
21  };
22
23  meta::meta('externalize', <<'EOF');
24  sub meta::externalize {
25      my ($name, $attribute, $implementation) = @_;
26      $externalized_functions{$name} = $attribute;
27      *{$name} = $implementation;
28  }
29  EOF
30
31  meta::meta('type::bootstrap', <<'EOF');
32  meta::define_form 'bootstrap', sub {};
33  EOF
34
35  meta::meta('type::function', <<'EOF');
36  meta::define_form 'function', sub {
37      my ($name, $body) = @_;
38      meta::externalize $name, "function::$name", eval "sub {\n$body\n}";
39  };
40  EOF
41
42  meta::meta('type::internal_function', <<'EOF');
43  meta::define_form 'internal_function', sub {
44      my ($name, $value) = @_;
45      *{$name} = eval "sub {\n$value\n}";
46  };
47  EOF
48
49  meta::meta('type::data', <<'EOF');
50  meta::define_form 'data', sub {
51      # Define a basic editing interface:
52      my ($name, $value) = @_;
53      meta::externalize $name, "data::$name", sub {
54          my ($command, $value) = @_;
55          return $data{"data::$name"} unless @_;

```

```

56     $data{"data::$name"} = $value if $command eq '=';
57     };
58 };
59 EOF
60
61 meta::bootstrap('initialization', <<'EOF');
62 #!/usr/bin/perl
63 my %data;
64 my %externalized_functions;
65 my %datatypes;
66
67 sub meta::define_form {
68     my ($namespace, $delegate) = @_;
69     $datatypes{$namespace} = $delegate;
70     *{"meta::{$namespace}::implementation"} = $delegate;
71     *{"meta::$namespace"} = sub {
72         my ($name, $value) = @_;
73         chomp $value;
74         $data{"{$namespace}::$name"} = $value;
75         $delegate->($name, $value);
76     };
77 }
78
79 meta::define_form 'meta', sub {
80     my ($name, $value) = @_;
81     eval $value;
82 };
83 EOF
84
85 meta::data('default-action', <<'EOF');
86 shell
87 EOF
88
89 meta::internal_function('namespace', <<'EOF');
90 my ($name) = @_;
91 $name =~ s/::.*$//;
92 $name;
93 EOF
94
95 meta::internal_function('attribute', <<'EOF');
96 my ($name) = @_;
97 $name =~ s/^[^:]*::~//;
98 $name;
99 EOF
100
101 meta::internal_function('retrieve', <<'EOF');

```



```

102 my @results = map defined $data{$_} ? $data{$_} : file::read($_), @_;
103 wantarray ? @results : $results[0];
104 EOF
105
106 meta::internal_function('associate', <<'EOF');
107 my ($name, $value, %options) = @_;
108 my $namespace = namespace($name);
109 die "Namespace $namespace does not exist" unless $datatypes{$namespace};
110 $data{$name} = $value;
111 execute($name) if %options{'execute'};
112 EOF
113
114 meta::internal_function('execute', <<'EOF');
115 my ($name, %options) = @_;
116 my $namespace = namespace($name);
117 eval {&{"meta::$namespace"}(attribute($name), retrieve($name))};
118 warn $@ if $@ && %options{'carp'};
119 EOF
120
121 meta::function('serialize', <<'EOF');
122 my @keys = sort keys %data;
123 join "\n", $data{'bootstrap::initialization'},
124         map(serialize_single($_,
125             grep(/^meta:\/, @keys),
126             grep(!/^meta:\/, @keys)),
127         "internal::main();",
128         "__END__";
129 EOF
130
131 meta::function('serialize_single', <<'EOF');
132 my ($namespace, $name) = split /::/, $_[0], 2;
133 my $marker = '__' . fast_hash($data{$_[0]});
134 "meta::$namespace('$name', <<'$marker');\n$data{$_[0]}\n$marker";
135 EOF
136
137 meta::function('fnv_hash', <<'EOF');
138 my ($data) = @_;
139 my ($fnv_prime, $fnv_offset) = (16777619, 2166136261);
140 my $hash = $fnv_offset;
141 my $modulus = 2 ** 32;
142 $hash = ($hash ^ ($_ & 0xffff) ^ ($_ >> 16)) * $fnv_prime % $modulus
143 for unpack 'L*', $data . substr($data, -4) x 8;
144 $hash;
145 EOF
146
147 meta::function('fast_hash', <<'EOF');

```

```

148 my ($data)      = @_;
149 my $piece_size = length($data) >> 3;
150 my @pieces      = (substr($data, $piece_size * 8) . length($data),
151                    map(substr($data, $piece_size * $_, $piece_size), 0 .. 7));
152 my @hashes      = (fnv_hash($pieces[0]));
153 push @hashes, fnv_hash($pieces[$_ + 1] . $hashes[$_]) for 0 .. 7;
154 $hashes[$_] ^= $hashes[$_ + 4] >> 16 | ($hashes[$_ + 4] & 0xffff) << 16 for 0 .. 3;
155 $hashes[0] ^= $hashes[8];
156 sprintf '%08x' x 4, @hashes[0 .. 3];
157 EOF
158
159 meta::function('state', <<'EOF');
160 fast_hash(serialize());
161 EOF
162
163 meta::function('verify', <<'EOF');
164 my $serialized_data = serialize();
165 my $state           = state();
166
167 my $temporary_filename = "$0.$state";
168 open my $file, '>', $temporary_filename;
169 print $file $serialized_data;
170 close $file;
171 chmod 0700, $temporary_filename;
172 chomp(my $observed_state = join '', qx|perl '$temporary_filename' state|);
173 my $result = $observed_state eq $state;
174 unlink $temporary_filename if $result;
175 $result;
176 EOF
177
178 meta::function('save', <<'EOF');
179 if (verify()) {
180     open my $file, '>', $0;
181     print $file serialize();
182     close $file;
183     chmod 0744, $0;
184 } else {
185     warn 'Verification failed';
186 }
187 EOF
188
189 meta::function('ls', <<'EOF');
190 join "\n", sort keys %data;
191 EOF
192
193 meta::function('cp', <<'EOF');

```

```

194 associate($_[1], retrieve($_[0]));
195 EOF
196
197 meta::function('rm', <<'EOF');
198 delete @data{@_};
199 EOF
200
201 meta::function('cat', <<'EOF');
202 join "\n", @data{@_};
203 EOF
204
205 meta::function('create', <<'EOF');
206 return edit($_[0]) if exists $data{$_[0]};
207 associate($_[0], $_[1] || "# Attribute $_[0]");
208 edit($_[0]);
209 EOF
210
211 meta::function('set', <<'EOF');
212 $data{$_[0]} = join '', <STDIN>;
213 EOF
214
215 meta::function('complete', <<'EOF');
216 my @functions = sort keys %externalized_functions;
217 my @attributes = sort keys %data;
218 sub match {
219     my ($text, @options) = @_;
220     my @matches = sort grep /^$text/, @options;
221     if (@matches == 0) {return undef;}
222     elsif (@matches == 1) {return $matches [0];}
223     elsif (@matches > 1) {
224         return ((longest ($matches [0], $matches [@matches - 1])), @matches);
225     }
226 }
227 sub longest {
228     my ($s1, $s2) = @_;
229     return substr ($s1, 0, length $1) if ($s1 ^ $s2) =~ /^(\0*)/;
230     return '';
231 }
232 my ($text, $line) = @_;
233 if ($line =~ / /) {
234     # Start matching attribute names.
235     match ($text, @attributes);
236 } else {
237     # Start of line, so it's a function.
238     match ($text, @functions);
239 }

```

```

240 EOF
241
242 meta::internal_function('shell', <<'EOF');
243 use Term::ReadLine;
244 my $term = new Term::ReadLine "$0 shell";
245 $term->ornaments(0);
246 my $output = $term->OUT || \*STDOUT;
247 $term->Attribs->{attempted_completion_function} = \&complete;
248 while (defined($_ = $term->readline("$0$ "))) {
249     my @args = grep length, split /\s+|("[^"\\"]*(?:\\.)?")/o;
250     my $function_name = shift @args;
251     s/^(.*)"$\1/o, s/\\\\""/"/go for @args;
252
253     if ($function_name) {
254         if ($externalized_functions{$function_name}) {
255             chomp(my $result = eval {"&$function_name(@args)"});
256             warn "$@" if $@;
257             print $output $result, "\n" unless $@;
258         } else {
259             warn "Command not found: '$function_name' (use 'ls' to see available commands)";
260         }
261     }
262 }
263 EOF
264
265 meta::function('edit', <<'EOF');
266 my $filename = '/tmp/' . rand();
267 open my $file, '>', $filename;
268 print $file retrieve($_[0]);
269 close $file;
270 system($ENV{EDITOR} || $ENV{VISUAL} || '/usr/bin/nano', $filename);
271 open my $file, '<', $filename;
272 associate($_[0], join '', <$file>);
273 close $file;
274 EOF
275
276 meta::internal_function('internal::main', <<'EOF');
277 my $initial_state = state();
278 my $command = shift @ARGV || retrieve('data::default-action');
279 print &$command(@ARGV);
280 save() if state() ne $initial_state;
281 EOF
282
283 internal::main();
284
285 __END__

```

**Part II**

**The Fun Stuff**

## Chapter 8

# eval backtraces

Our script is fairly awesome so far. It prevents us from creating attributes in namespaces that don't exist, since that would cause incorrect serialization, it verifies before it saves, etc. But there's one problem. Take a look at the error messages we get:

```
$ perl examples/some-improvements
examples/some-improvements$ create foo::bar
Namespace foo does not exist at (eval 9) line 4.
examples/some-improvements$
```

If there's a problem in some attribute, we have no information about the location of the error other than "eval *n*" and the line number relative to that. `object` solves this problem:

```
$ object
object$ create foo::bar
[error] Namespace foo does not exist at internal_function::associate line 4.
object$
```

The key is to wrap `eval` in such a way that we can later resolve the meaningless numbers into useful locations. And to do this, we're going to need to modify the bootstrap code again.

```
my %data;
my %externalized_functions;
my %datatypes;
my %locations;      # Maps eval-numbers to attribute names
```

There's a beautiful hack to handle the `eval` processing. Watch this (also in `bootstrap::initialization`):<sup>1</sup>

---

<sup>1</sup>It actually doesn't have to be inside the bootstrap code, but it doesn't change often and is useful to have around, so I decided to put it there to save time.

**Listing 8.1** snippets/meta-eval-in

```
1 sub meta::eval_in {
2   my ($what, $where) = @_;
3   # Obtain next eval-number and alias it to the designated location
4   @locations{eval('__FILE__') =~ /\(eval \(\d+\)\)/} = ($where); # <- step 1
5   my $result = eval $what; # <- step 2
6   $@ =~ s/\(eval \(\d+\)\)/$where/ if $@;
7   warn $@ if $@;
8   $result;
9 }
```

By evaluating `__FILE__`, we get the current eval number. So the next one will be whatever we eval next. This means that in the shell sessions above, `%locations` contains a mapping from 9 to `internal_function::associate`. Here's the function that converts an eval index into an attribute name:

**Listing 8.2** snippets/translate-backtrace-function

```
1 meta::internal_function('translate_backtrace', <<'EOF');
2 my ($trace) = @_;
3 $trace =~ s/\(eval \(\d+\)\)/$locations{$1 - 1}/g;
4 $trace;
5 EOF
```

Notice that we're subtracting one. The eval number that triggered the error will be one greater than the one we stored.<sup>2</sup>

Now that we have this mechanism, we can go back and convert eval calls into `meta::eval_in`:

**Listing 8.3** snippets/using-eval-in

```
1 meta::define_form 'function', sub {
2   my ($name, $value) = @_;
3   meta::externalize $name, "function::$name",
4     meta::eval_in("sub {\n$value\n}", "function::$name");
5 };
6
7 meta::define_form 'internal_function', sub {
8   my ($name, $value) = @_;
9   *{$name} =
10     meta::eval_in("sub {\n$value\n}", "internal_function::$name");
11 };
```

---

<sup>2</sup>Good API design would resolve this ahead-of-time rather than at lookup time. I haven't gotten around to changing it yet though.

## Chapter 9

# Archiving state

Suppose you're about to do something risky with a script and you want to take a snapshot that you can restore to. You could copy into another file, but that's a brute-force approach and it requires you to exit the script's shell. Better is to have some kind of internal state management, and that's where explicit states come into play.

Remember that `%data` is just a variable; we can do all of the usual things with it. We can store a state by doing a partial serialization into an attribute, and we can restore from that state by evaling that attribute. To do this we're going to need another namespace.

**Listing 9.1** snippets/state-type

```
1 meta::meta('type::state', <<'EOF');
2 # No action when a state is defined
3 meta::define_form 'state', \&meta::bootstrap::implementation;
4 EOF
```

### 9.1 Saving state

It's tempting to think that this code would do what we want:

```
# Won't work:
associate("state::$_[0]", serialize());
```

Unfortunately, `serialize` generates three things that we don't want. These are the bootstrap section at the beginning, the call to `internal::main()` at the end, and any attribute in the `state::` namespace.<sup>1</sup> We'll need to write a separate function to serialize just what we want:

---

<sup>1</sup>If some states contained others, the script size would grow exponentially in the number of states.



**Listing 9.2** snippets/current-state-function

```
1 meta::function('current-state', <<'EOF');
2 my @valid_keys = grep ! /^state::/, sort keys %data;
3 my @ordered_keys = (grep(/^meta::/, @valid_keys), grep(! /^meta::/, @valid_keys));
4 join "\n", map serialize_single($_), @ordered_keys;
5 EOF
```

And here's a save-state function to automate the state creation process:

**Listing 9.3** snippets/save-state-function

```
1 meta::function('save-state', <<'EOF');
2 my ($state_name) = @_ ;
3 associate("state::$state_name", &{'current-state'}());
4 EOF
```

## 9.2 Loading state

This is not as straightforward as saving state. Because we're modifying %data live, we have to be careful about what happens in the event that something goes wrong. We also don't want to have stray %data elements or externalized functions. The easiest way to defend against errors is to save the current state before applying a new one. Here's the implementation of load-state:

**Listing 9.4** snippets/load-state-function

```
1 meta::function('load-state', <<'EOF');
2 my ($state_name) = @_ ;
3 my $state = retrieve("state::$state_name");
4 &{'save-state'}(' '); # Make a backup
5 delete $data{$_} for grep ! /^state::/, keys %data;
6 %externalized_functions = ();
7 eval($state); # Apply the new state
8 warn "$@" if $@;
9 verify(); # Make sure it worked
10 EOF
```

If the load failed for some reason, you can restore using load-state \_ . If it failed badly enough to bork your load-state function, then you have a problem.

## 9.3 The hypothetically function

Related to state management is a function called `hypothetically`, which lets you try something out and then revert. It's used internally to examine the

state of a modified copy without actually committing changes.<sup>2</sup> Here's how it's defined:

**Listing 9.5** snippets/hypothetically-function

```
1 meta::internal_function('hypothetically', <<'EOF');
2 my %data_backup = %data;
3 my ($side_effect) = @_;
4 my $return_value = eval {&$side_effect()};
5 %data = %data_backup;
6 die $@ if $@;
7 $return_value;
8 EOF
```

You can use it like this:

```
my $x = hypothetically(sub {
    associate('data::foo', '10');
    retrieve('data::foo');
});
my $y = retrieve('data::foo');
# now $x eq '10' and $y is undef
```

---

<sup>2</sup>This is covered in [chapter 10](#).

## Chapter 10

# Cloning and inheritance

This is probably the single coolest thing about self-modifying Perl programs. You’ve probably had this looming feeling that propagating updated versions of functions between scripts was going to be a complete nightmare. For a long time this was indeed the case; I had shell scripts that copied attributes out of one script and into another. Luckily I got tired of doing things that way and came up with the inheritance mechanism that’s used now.

Inheritance isn’t as simple as copying all of the attributes from one script into another. Certain namespaces like `data::` are script-specific, for instance. We’ll need to have some way to keep track of which namespaces should be inherited.

Another issue is getting attributes from one script into another. My first implementation of inheritance retrieved each attribute individually. It used `ls` and `cat` for the transfer, which involved  $O(n)$  runs of whichever script was being inherited from. Obviously it was really slow.  $O(n)$  runs of a function containing  $n$  functions means  $O(n^2)$  total time, and Perl isn’t blazingly fast at evaling functions. Later on I extended `serialize` to return a bundle of attributes that the child then eval’d.

### 10.1 Tracking inheritability

We’re going to need another toplevel field if we want to store data about data types. We can’t use `%data`, since we don’t really want to save it (whatever we’re storing would be regenerated automatically anyway). What we really need is a way to store transient information:

```
my %data;
my %externalized_functions;
my %datatypes;
my %transient;
```

`%transient` does nothing except store stuff while the script is running, and all of its information is discarded when the script exits. It's basically just a temporary workspace where we can stash stuff.

We can now use `%transient` to store things about data types. For convenience let's define `meta::configure` to do this for us:<sup>1</sup>

**Listing 10.1** snippets/meta-configure

```
1 meta::meta('configure', <<'EOF');
2 sub meta::configure {
3   my ($datatype, %options) = @_;
4   $transient{$_}{$datatype} = $options{$_} for keys %options;
5 }
6 EOF
```

Now we can add a configuration to each datatype we define:

```
meta::meta('type::function', <<'EOF');
meta::configure 'function', inherit => 1;
meta::define_form 'function', ...;
EOF

meta::meta('type::data', <<'EOF');
meta::configure 'data', inherit => 0;
meta::define_form 'data', ...;
EOF

meta::meta('type::internal_function', <<'EOF');
meta::configure 'internal_function', inherit => 1;
...
EOF

meta::meta('type::bootstrap', <<'EOF');
meta::configure 'bootstrap', inherit => 1;
...
EOF

meta::meta('type::state', <<'EOF');
meta::configure 'state', inherit => 0;
...
EOF
```

---

<sup>1</sup>For some reason I decided to store the keys in the odd order of `option-namespace` instead of the other way around. I'm still not sure why I did it this way, but it doesn't seem to cause problems.

## 10.2 Extensions to serialize

`serialize` needs to be able to give us a bundle of code to create just the attributes that should be inherited. While we're at it, it would also be nice if it handed us just the `meta::` attributes and then just the non-`meta::` attributes. This way we can make sure that the `meta::` attributes didn't break anything and bail out early if they did.

None of this is particularly challenging, but given that we're going to invoke `serialize` externally we should probably fix the `%options` stuff. (The last thing we want to write is something like `qx($script serialize partial 1 meta 1 inheritable 1)`). What we need is an adapter that turns command-line options into Perl hashes.<sup>2</sup> Here's a function that uses `Getopt`-style parsing:

Listing 10.2 snippets/separate-options-function

```
1 meta::internal_function('separate_options', <<'EOF');
2 # Things with one dash are short-form options, two dashes are long-form.
3 # Characters after short-form are combined; so -auv4 becomes -a -u -v -4.
4 # Also finds equivalences; so --foo=bar separates into $$options{'--foo'} eq 'bar'.
5 # Stops processing at the -- option, and removes it. Everything after that
6 # is considered to be an 'other' argument.
7
8 # The only form not supported by this function is the short-form with argument.
9 # To pass keyed arguments, you need to use long-form options.
10 my @parseable;
11 push @parseable, shift @_ until ! @_ or $_[0] eq '--';
12
13 my @singles = grep /^-[^-]/, @parseable;
14 my @longs   = grep /^--/, @parseable;
15 my @others  = grep ! /^-/, @parseable;
16 my @singles = map /-(.{2,})/ ? map("-$_", split(//, $_)) : $_, @singles;
17 my %options;
18 $options{$1} = $2 for grep /^([^=]+)=(.*)$/, @longs;
19 ++$options{$_} for grep ! /=/, @singles, @longs;
20
21 ({%options}, @others, @_);
22 EOF
```

The output of this function is a reference to a hash of any keyword arguments (where short-form arguments are treated as increments) followed by any non-switch arguments (either because they came after `--` or because they didn't start with a dash at all. For example, processing the arguments `-xy --z=foo bar` would yield `({-x => 1, -y => 1, --z => foo}, bar)`.<sup>3</sup>

Given the ability to pipe options into `serialize` on the command-line, we

---

<sup>2</sup>OK, I'm making a jump here. Later it will become clearer why it's good to do it this way.

<sup>3</sup>Given the similarity, I don't remember why I didn't just use `Getopt::Long` for this stuff. I think I must have been having a NIH day.

just need to have it support a reasonably flexible selection interface. We'll later need to have `ls` support the same options, so let's factor the key selector into its own function:

**Listing 10.3** snippets/select-keys-function

```

1 meta::internal_function('select_keys', <<'EOF');
2 my %options = @_;
3 my $criteria = $options{'--criteria'} ||
4     $options{'--namespace'} && "^$options{'--namespace'}::" || '.';
5 grep /$criteria/ && (! $options{'-i'} || $transient{inherit}{namespace($_)}) &&
6     (! $options{'-I'} || ! $transient{inherit}{namespace($_)}) &&
7     (! $options{'-S'} || ! /^state::/o) &&
8     (! $options{'-m'} || /^meta::/o) &&
9     (! $options{'-M'} || ! /^meta::/o), sort keys %data;
10 EOF

```

This function takes the `%options` hash output by `separate_options` as input and returns a list of keys into `%data`. The somewhat odd logical structure of the `grep` predicate is just implication: “if `$options{'-i'}` is set, then the key's namespace must be inheritable.”

The new `serialize` function is fairly simple; most of the heavy lifting is already done:

**Listing 10.4** snippets/serialize-final

```

1 meta::function('serialize', <<'EOF');
2 my ($options, @criteria) = separate_options(@_);
3 my $partial = $$options{'-p'};
4 my $criteria = join '|', @criteria;
5 my @attributes = map serialize_single($_,
6     select_keys(%$options, '-m' => 1, '--criteria' => $criteria),
7     select_keys(%$options, '-M' => 1, '--criteria' => $criteria));
8 my @final_array = @{$partial ? \@attributes :
9     [retrieve('bootstrap::initialization'),
10     @attributes,
11     'internal::main();', '',
12     '___END___']};
13 join "\n", @final_array;
14 EOF

```

`-m` and `-M` select `meta::` and `non-meta::` attributes, respectively. We also provide criteria if the user has selected any. They're joined together with a pipe symbol because that forms a disjunction inside a regular expression (and criteria are regexps for attribute names). We also, somewhat importantly, have a `-p` switch to produce a partial serialization. This leaves off the bootstrap code and the `internal::main()` call. The only difference between this and `current-state` is that `current-state` also leaves out `state::` attributes.<sup>4</sup>

<sup>4</sup>This version of `serialize` also will do that for you if you pass the `-S` option.

## 10.3 The update-from function

Here's where things start to get interesting. `update-from` handles the case where you have two scripts `base` and `child`, and you want `child` to inherit stuff from `base`.<sup>5</sup> Here's a basic implementation:

**Listing 10.5** snippets/update-from-function

```
1 meta::function('update-from', <<'EOF');
2 my ($options, @targets) = separate_options(@_);
3 my %options = %$options;
4 @targets or die 'Must specify at least one target to update from';
5 my $save_state = ! ($options{'-n'} || $options{'--no-save'});
6 my $force      = $options{'-f'} || $options{'--force'};
7
8 &{'save-state'}('before-update') if $save_state;
9 for my $target (@targets) {
10     eval qx($target serialize -ipm);
11     eval qx($target serialize -ipM);
12     reload();    # We're about to define this
13     unless (verify()) {
14         if ($force) {
15             warn 'Verification failed, but keeping new state';
16         } else {
17             warn "Verification failed after $target; reverting";
18             return &{'load-state'}('before-update') if $save_state;
19         }
20     }
21 }
22 EOF
```

If `child` has this function, you can update it this way (assuming you've set execute permissions):

```
$ ./child update-from ./base
$
```

If all goes well it will execute without failing. `base` will be run twice: once to grab inheritable `meta::`-attributes, again to grab inheritable other attributes.

The `reload` function just calls `execute` on each member of `%data`. It's there to make sure that the new attributes and the old attributes work well together. Here's the definition:

**Listing 10.6** snippets/reload-function

```
1 meta::function('reload', <<'EOF');
2 execute($_) for grep ! /^bootstrap::/, keys %data;
3 EOF
```

---

<sup>5</sup>"Stuff" is deliberately vague. Presumably we want to inherit every inheritable attribute though.

## 10.4 Managing parents

If you're using scripts as add-on modules, it gets tiring to issue `update-from` commands when you're using `n` modules. Further, you have to type out the whole path of the module, which might be in a separate directory. Finally, attributes that later get deleted or renamed in the add-on modules won't be cleaned up in the child script. The way we've got inheritance set up is woefully incomplete.

The missing piece is parent tracking. We need to keep track of two things:

1. Which scripts have I inherited from?
2. Which properties did I inherit from each one?

Fortunately this isn't too hard. We just need a new namespace and some new functions.

### 10.4.1 The `parent::` namespace

We can use the `parent::` namespace to answer both of the above questions. Each attribute will take its name from the path to a script (e.g. `parent::usr/bin/script1`), and its contents will be a newline-separated string of the attributes inherited from that script. Here's the type definition:

Listing 10.7 snippets/parent-type

```
1 meta::meta('type::parent', <<'EOF');
2 meta::configure 'parent', inherit => 1; # Transitive parents (explained below)
3 meta::define_form 'parent', \&meta::bootstrap::implementation;
4 EOF
```

Now we need to update `update-from` to populate this namespace. Before we do, though, we need to define uniqueness.

### 10.4.2 Uniqueness

Let's suppose you've got three objects `a`, `b`, and `c`, and each inherits from the previous one.<sup>6</sup> Then `b`'s parent is `a`, and `c`'s parent is `b`. It's important for `c` to know that `a` is also its parent. The reason is that otherwise you're forced to update `b` before `c`, since `c` inherits only from `b`.

It's ultimately for this reason that `parent::` attributes get inherited. Inheritance is certainly transitive (if `c` inherits from `b`, which inherits from `a`, then `c` inherits from `a`). There are, however, some logistical matters to be dealt with. The most important one has to do with ordering.

It shouldn't matter in which order the parents are inherited from. This is an interesting requirement, because it means that the things an object inherits

---

<sup>6</sup>I'm presupposing that inheritance works automatically even though we haven't defined it quite yet.



from each of its parents form disjoint sets. The only way to pull this off is if each object keeps track of how it's different from its own parents. The attributes that the child has but the parent doesn't are considered unique.

### 10.4.3 Updating ls and serialize

We need a way to ask `ls` for the list of unique attributes for an object, and then ask `serialize` to give us just those attributes. To do this, we'll add a `-u` option (and for symmetry a complement `-U`) to `select_keys`:

Listing 10.8 snippets/select-keys-final

```

1 meta::internal_function('select_keys', <<'EOF');
2 my %options = @_;
3 my %inherited = map {$_ => 1} split /\n/o, join "\n",
4     retrieve(grep /^parent::/o, sort keys %data)
5     if $options{'-u'} or $options{'-U'};
6 my $criteria = $options{'--criteria'} ||
7     $options{'--namespace'} && "^$options{'--namespace'}::" || '.';
8 grep /$criteria/ && (! $options{'-u'} || ! $inherited{$_}) &&
9     (! $options{'-U'} || $inherited{$_}) &&
10    (! $options{'-i'} || $transient{inherit}{namespace($_)}) &&
11    (! $options{'-I'} || ! $transient{inherit}{namespace($_)}) &&
12    (! $options{'-S'} || ! /^state::/o) &&
13    (! $options{'-m'} || /^meta::/o) &&
14    (! $options{'-M'} || ! /^meta::/o), sort keys %data;
15 EOF

```

The extra logic here searches through all of the `parent::` attributes to find properties that the parents also contain. If any parent contains one, then the property isn't unique.

`serialize` will get this new behavior automatically, since it just forwards its options to `select_keys`. But we haven't modified `ls` in a long time; it doesn't know anything about options. There are actually quite a few enhancements that we could make to `ls`, but for now let's keep it simple and change it only as necessary:

Listing 10.9 snippets/ls-with-options

```

1 meta::function('ls', <<'EOF');
2 my ($options, @criteria) = separate_options(@_);
3 join "\n", select_keys('--criteria' => join '|', @criteria, %$options);
4 EOF

```

Now you can say things like `./script ls -iu` to get a listing of attributes that are both inheritable and unique.

### 10.4.4 An updated update-from function

update-from now has three new responsibilities. One is to record the fact that we updated from another script, which involves creating a new parent:: attribute for each inheritance operation. Another is to ask each new parent which attributes it intends to define. The last thing it needs to do is clean up any attributes that some parent used to define but no longer does.<sup>7</sup> Here's the new implementation:

Listing 10.10 snippets/update-from-final

```
1 my ($options, @targets) = separate_options(@_);
2 my %options = %$options;
3 @targets or die 'Must specify at least one target to update from';
4 my $save_state = ! ($options{'-n'} || $options{'--no-save'});
5 my $no_parents = $options{'-P'} || $options{'--no-parent'} || $options{'--no-parents'};
6 my $force = $options{'-f'} || $options{'--force'};
7
8 &{'save-state'}('before-update') if $save_state;
9 for my $target (@targets) {
10     # The -a flag will become relevant once we add formatting to 'ls'
11     my $attributes = join ' ', qx($target ls -aiu);
12     warn "Skipping unreachable object $target" unless $attributes;
13     if ($attributes) {
14         # Remove keys that the parent used to define but doesn't anymore:
15         rm(split /\n/, retrieve("parent::$target")) if $data{"parent::$target"};
16         associate("parent::$target", $attributes) unless $no_parents;
17         eval qx($target serialize -ipmu);
18         eval qx($target serialize -ipMu);
19         warn $@ if $@;
20         reload();
21
22         if (verify()) {
23             print "Successfully updated from $_[0]. ",
24                 "Run 'load-state before-update' to undo this change.\n" if $save_state;
25         } elsif ($force) {
26             warn 'The object failed verification, but the failure state has been ' .
27                 'kept because --force was specified.';
28             warn 'At this point your object will not save properly, though backup ' .
29                 'copies will be created.';
30             print "Run 'load-state before-update' to undo the update and return to ",
31                 "a working state.\n" if $save_state;
32         } else {
33             warn 'Verification failed after the upgrade was complete.';
34             print "$0 has been reverted to its pre-upgrade state.\n" if $save_state;
35             print "If you want to upgrade and keep the failure state, then run ",
```

---

<sup>7</sup>Due to how update-from is structured, this step actually happens first.

```

36         "'update-from $target --force'." if $save_state;
37     return &{'load-state'}('before-update') if $save_state;
38 }
39 }
40 }

```

### 10.4.5 The update function

This new `update-from` function contains all of the logic to perform individual updates, but it still requires you to list the parent objects. There isn't any need to do this manually though, since if we look for `parent::` attributes we can get the list. That's what `update` does:

**Listing 10.11** snippets/update-function

```

1 meta::function('update', <<'EOF');
2 &{'update-from'}(@_, grep s/^parent:://o, sort keys %data);
3 EOF

```

## 10.5 clone and child

As things stand creating children from an object is a bit cumbersome. We have to manually copy the object and then run `update-from` once to get the parent to work out, which seems like too much work. Let's take care of copying first by creating a `clone` function:

**Listing 10.12** snippets/clone-function

```

1 meta::function('clone', <<'EOF');
2 open my $file, '>', $_[0];
3 print $file serialize();
4 close $file;
5 chmod 0700, $_[0];
6 EOF

```

Now you can clone an object as it exists at any given moment. More interesting, though, is the related `child` function, which creates an object already setup for inheritance:

**Listing 10.13** snippets/child-function

```

1 meta::function('child', <<'EOF');
2 my ($child_name) = @_;
3 clone($child_name);
4 qx($child_name update-from $0 -n);
5 EOF

```

object implements this function; so, for example, you could inherit from it like this:

```
$ /path/to/object child ./foo
$ ./foo update -n
$ ./foo ls -a parent::
parent::/path/to/object
$
```