

X shell

Spencer Tipping

February 23, 2014

Contents

I	Language reference	2
1	Expansion syntax	3
II	Bootstrap implementation	4
2	Self-replication	5
3	Data structures	7
4	Evaluator	12
5	Globals	17
6	Bootstrap unit tests	23
7	REPL	26

Part I

Language reference

Chapter 1

Expansion syntax

```
xh$ echo $foo                # simple variable expansion
xh$ echo $(echo hi)          # command output expansion
xh$ echo [$foo '0 @#]        # #words in first line of val of var foo
xh$ echo [{foo bar} "#]     # number of bytes in quoted string 'foo bar'

xh$ echo $foo[0 1]           # reserved for future use (don't write this)
xh$ echo $foo$bar            # reserved for future use (use ${foo}$bar)

xh$ echo $foo                # quote result with braces
xh$ echo $('foo)              # flatten into multiple lines (be careful!)
xh$ echo @$foo               # flatten into multiple words (one line)
xh$ echo $:foo               # multiple path components (one word)
xh$ echo $"foo               # multiple bytes (one path component)

xh$ echo ${foo}              # same as $foo
xh$ echo ${foo bar bif}     # reserved for future use

xh$ echo @${asdf asdf}      # expands into asdf adsf

xh$ echo $$foo               # $ is right-associative
xh$ echo ^$foo               # expand $foo within calling context
xh$ echo $('foo)             # result of running $('foo within current scope
xh$ $('foo                   # this works too
xh$ echo ^$('foo)            # result of running $('foo within calling scope
```

Part II

Bootstrap implementation

Chapter 2

Self-replication

Listing 2.1 boot/xh-header

```
1  #!/usr/bin/env perl
2  BEGIN {
3    print STDERR q{
4      NOTE: Development image
5
6      If you see this note after installing the shell, it's probably because
7      you're running a version that has not yet rebuilt itself (maybe you got the
8      wrong file from the Git repo?). You can do this, but it will be really
9      slow and may use a lot of memory. There are two ways to fix this:
10
11      1. Download the standard image from http://spencertipping.com/xh
12      2. Have this image recompile itself by running xh.recompile-in-place (this
13         will take some time because it stress-tests your Perl runtime)
14
15      Note also that bootstrapping requires Perl 5.14 or later, whereas running a
16      compiled image just requires Perl 5.10.
17
18    };
19  }
20
21  BEGIN {eval(our $xh_bootstrap = q{
22    # xh: the X shell | https://github.com/spencertipping/xh
23    # Copyright (C) 2014, Spencer Tipping
24    # Licensed under the terms of the MIT source code license
25
26    # For the benefit of HTML viewers (long story):
27    # <body style='display:none'>
28    # <script src='http://spencertipping.com/xh/page.js'></script>
29    use 5.014;
```

```

30 package xh;
31 our %modules;
32 our @module_ordering;
33
34 our %compilers = (pl => sub {
35     my $package = $_[0] =~ s/\./::/gr;
36     eval "{package ::$package;\n$_[1]\n}";
37     die "error compiling module $_[0]: $@" if $@;
38 });
39
40 sub defmodule {
41     my ($name, $code, @args) = @_;
42     chomp($modules{$name} = $code);
43     push @module_ordering, $name;
44     my ($base, $extension) = split /\.(?w+)$/, $name;
45     die "undefined module extension '$extension' for $name"
46         unless exists $compilers{$extension};
47     $compilers{$extension}->($base, $code, @args);
48 }
49
50 chomp($modules{bootstrap} = $::xh_bootstrap);
51 undef $::xh_bootstrap;

```

At this point we need a way to reproduce the image. Since the bootstrap code is already stored, we can just wrap it and each defined module into an appropriate BEGIN block.

Listing 2.2 boot/xh-header (continued)

```

1 sub image {
2     my @pieces = "#!/usr/bin/env perl";
3     push @pieces, "BEGIN {eval(our \$_xh_bootstrap = <<'_')}",
4         $modules{bootstrap},
5         '_';
6     push @pieces, "BEGIN {xh::defmodule('$_', <<'_')}",
7         $modules{$_},
8         '_ ' for @module_ordering;
9     push @pieces, "xh::main::main;\n__DATA__";
10    join "\n", @pieces;
11 }
12 }}

```

Chapter 3

Data structures

All values in `xh` have the same type, which provides a bunch of operations suited to different purposes. This implementation is based on strings and, as a result, has egregious performance appropriate only for bootstrapping the self-hosting compiler.

Listing 3.1 `modules/v.pl`

```
1 BEGIN {xh::defmodule('xh::v.pl', <<'_'')}
2 sub unbox;
3
4 sub parse_with_quoted {
5   my ($events_to_split, $split_sublists, $stake_zero_width, $s) = @_;
6   my @result;
7   my $current_item = '';
8   my $sublist_depth = 0;
9
10  for my $piece (split /\v|\s+|\v|\\\.|[\[\]\{\}]/, $s) {
11    next if !$stake_zero_width and !length $piece;
12    my $depth_before_piece = $sublist_depth;
13    $sublist_depth += $piece =~ /\[\{]/;
14    $sublist_depth -= $piece =~ /\[\]}/;
15
16    if ($split_sublists && !$sublist_depth != !$depth_before_piece) {
17      # Two possibilities. One is that we just closed an item, in which
18      # case we take the piece, concatenate it to the item, and continue.
19      # The other is that we just opened one, in which case we emit what we
20      # have and start a new item with the piece.
21      if ($sublist_depth) {
22        # Just opened one; kick out current item and start a new one.
23        push @result, $current_item if $stake_zero_width or
24                                     length $current_item;
25        $current_item = $piece;
```



```

26     } else {
27         # Just closed a list; concat and kick out the full item.
28         push @result, "$current_item$piece";
29         $current_item = '';
30     }
31 } elsif (!$sublist_depth && $piece =~ /$events_to_split/) {
32     # If the match produces a group, then treat it as a part of the next
33     # item. Otherwise throw it away.
34     push @result, $current_item if $take_zero_width or
35                                     length $current_item;
36     $current_item = $1;
37 } else {
38     $current_item .= $piece;
39 }
40 }
41
42 push @result, $current_item if length $current_item;
43 @result;
44 }
45
46 sub split_lines {parse_with_quoted '\v', 0, 1, @_}
47 sub split_words {parse_with_quoted '\s+', 0, 0, @_}
48 sub split_path {parse_with_quoted '(/)', 1, 0, @_}
49
50 sub parse_lines {map unbox($_), split_lines @_}
51 sub parse_words {map unbox($_), split_words @_}
52 sub parse_path {map unbox($_), split_path @_}
53
54 sub brace_balance {my $without_escapes = $_[0] =~ s/\.\./gr;
55                   length($without_escapes =~ s/^[{](/{)//gr) -
56                   length($without_escapes =~ s/[^\}]}]/gr)}
57
58 sub escape_braces_in {$_[0] =~ s/([\\\[ \] () {}])/\\$1/gr}
59
60 sub quote_as_multiple_lines {
61     return escape_braces_in $_[0] if brace_balance $_[0];
62     $_[0];
63 }
64
65 sub brace_wrap {"{" . quote_as_multiple_lines($_[0]) . "}" }
66
67 sub quote_as_line {parse_lines(@_) > 1 ? brace_wrap $_[0] : $_[0]}
68 sub quote_as_word {parse_words(@_) > 1 ? brace_wrap $_[0] : $_[0]}
69 sub quote_as_path {parse_path(@_) > 1 ? brace_wrap $_[0] : $_[0]}
70
71 sub quote_default {brace_wrap $_[0]}

```

```

72
73 sub split_by_interpolation {
74     # Splits a value into constant and interpolated pieces, where
75     # interpolated pieces always begin with $. Adjacent constant pieces may
76     # be split across items. Any active backslash-escapes will be placed on
77     # their own.
78
79     my @result;
80     my $current_item      = '';
81     my $sublist_depth     = 0;
82     my $blocker_count     = 0;      # number of open-braces
83     my $interpolating     = 0;
84     my $interpolating_depth = 0;
85
86     my $closed_something  = 0;
87     my $opened_something  = 0;
88
89     for my $piece (split /([\[\]\{\}\|\.\|\/\|$\|s+)/, $_[0]) {
90         $sublist_depth += $opened_something = $piece =~ /\[([{}$]/;
91         $sublist_depth -= $closed_something = $piece =~ /\][\]{}$]/;
92         $blocker_count += $piece eq '{';
93         $blocker_count -= $piece eq '}';
94
95         if (!$interpolating) {
96             # Not yet interpolating, but see if we can find a reason to change
97             # that.
98             if (!$blocker_count && $piece eq '$') {
99                 # Emit current item and start interpolating.
100                push @result, $current_item if length $current_item;
101                $current_item = $piece;
102                $interpolating = 1;
103                $interpolating_depth = $sublist_depth;
104            } elsif (!$blocker_count && $piece =~ /\[\[\|\/\|$\|s+/) {
105                # The backslash should be interpreted, so emit it as its own piece.
106                push @result, $current_item if length $current_item;
107                push @result, $piece;
108                $current_item = '';
109            } else {
110                # Collect the piece and continue.
111                $current_item .= $piece;
112            }
113        } else {
114            # Grab everything until:
115            #
116            # 1. We close the list in which the interpolation occurred.
117            # 2. We close a list to get back out to the interpolation depth.

```

```

118         # 3. We observe whitespace.
119         # 4. We observe a path separator.
120
121         if ($sublist_depth < $interpolating_depth
122             or $sublist_depth == $interpolating_depth
123             and $piece eq '/' || $piece =~ /^\\s/) {
124             # No longer interpolating because of what we just saw, so emit
125             # current item and start a new constant piece.
126             push @result, $current_item if length $current_item;
127             $current_item = $piece;
128             $interpolating = 0;
129         } elsif ($sublist_depth == $interpolating_depth
130             && $closed_something) {
131             push @result, "$current_item$piece";
132             $current_item = '';
133             $interpolating = 0;
134         } else {
135             # Still interpolating, so collect the piece.
136             $current_item .= $piece;
137         }
138     }
139 }
140
141 push @result, $current_item if length $current_item;
142 @result;
143 }
144
145 sub undo_backslash_escape {
146     return "\n" if $_[0] eq '\n';
147     return "\t" if $_[0] eq '\t';
148     return "\\" if $_[0] eq '\\\\';
149     substr $_[0], 1;
150 }
151
152 sub unbox {
153     my ($s) = @_;
154     my $depth = 0;
155     my $last_depth = 1;
156     for my $piece (grep length, split /(\\.|[\\[\]{}]), $s) {
157         $depth += $piece =~ /\[([{}])/;
158         $depth -= $piece =~ /\]([{}])/;
159         return $s if $last_depth <= 0;
160         $last_depth = $depth;
161     }
162     $s =~ s/^\\s*[\\[({}(.*[\\])])\\s*$/1/sgr;
163 }

```


Chapter 4

Evaluator

This bootstrap evaluator is totally cheesy, using Perl's stack and lots of recursion; beyond this, it is slow, allocates a lot of memory, and has absolutely no support for lazy values. Its only redeeming virtue is that it supports macroexpansion.

Listing 4.1 modules/e.pl

```
1 BEGIN {xh::defmodule('xh::e.pl', <<'_'')}
2 sub evaluate;
3 sub interpolate;
4 sub call;
5
6 sub interpolate_wrap {
7     my ($prefix, $unquoted) = @_;
8     return xh::v::quote_as_multiple_lines $unquoted if $prefix =~ /'$/;
9     return xh::v::quote_as_line           $unquoted if $prefix =~ /\@$/;
10    return xh::v::quote_as_word            $unquoted if $prefix =~ /:$/;
11    return xh::v::quote_as_path            $unquoted if $prefix =~ /"$/;
12    xh::v::quote_default $unquoted;
13 }
14
15 sub scope_index_for {
16     my ($carets) = $_[0] =~ /\^\$(\^*)/g;
17     -(1 + length $carets);
18 }
19
20 sub truncated_stack {
21     my ($stack, $index) = @_;
22     return $stack if $index == -1;
23     [@$stack[0 .. @$stack + $index]];
24 }
25
```

```

26 sub interpolate_dollar {
27   my ($binding_stack, $term) = @_;
28
29   # First things first: strip off any prefix operator, then interpolate the
30   # result. We do this because $ is right-associative.
31   my ($prefix, $rhs) = $term =~ /^(\$\^*[@'':.]?)(.*)$/sg;
32
33   # Do we have a compound form? If so, then we need to treat the whole
34   # thing as a unit.
35   if ($rhs =~ /^\(\/) {
36     # The exact semantics here are a little subtle. Because the RHS is just
37     # ()-boxed, it should be expanded within the current scope. The actual
38     # evaluation, however, might be happening within a parent scope; we'll
39     # know by looking at the $prefix to check for ^s.
40
41     my $interpolated_rhs = interpolate $binding_stack, $rhs;
42     my $index            = scope_index_for $prefix;
43     my $new_stack        = truncated_stack $binding_stack, $index;
44
45     return interpolate_wrap $prefix,
46                            evaluate $new_stack, $interpolated_rhs;
47   } elsif ($rhs =~ /^\[\/) {
48     # $[] is a way to call a series of functions on a value, just like
49     # Clojure's (-> x y z). Like $(), we always interpolate the terms of
50     # the [] list in the current environment; but any ^s you use (e.g.
51     # $^[ ]) cause the inner functions to be called from a parent scope.
52     # This can be relevant in certain pathological cases that you should
53     # probably never use.
54
55     my ($initial, @fns) = map {interpolate $binding_stack, $_}
56                            $rhs =~ /^\[([^\]]+)$/sg;
57     my $index            = scope_index_for $prefix;
58     my $calling_stack    = truncated_stack $binding_stack, $index;
59
60     # You can use paths as a curried notation within $[] interpolation. For
61     # example:
62     #
63     # > echo $[foo echo/hi]
64     # hi foo
65     #
66     # Lists also work, but there is no difference between () and [], which
67     # is a horrible oversight that should probably be addressed at some
68     # point.
69     $initial = call $calling_stack,
70                   (map {s/^\/\//r} $fns),
71                   $initial

```

```

72     for @fns;
73
74     return interpolate_wrap $prefix, $initial;
75 } elsif ($rhs =~ /\^\{\/) {
76     $rhs = xh::v::unbox $rhs;
77 } else {
78     # It's either a plain word or another $-term. Either way, go ahead and
79     # interpolate it so that it's ready for this operator.
80     $rhs = xh::v::unbox interpolate $binding_stack, $rhs;
81 }
82
83 # Try to unwrap any layers around the RHS. Any braces at this point mean
84 # that it's artificially quoted, or that the RHS is unusable.
85 while ($rhs =~ /\^\{\/) {
86     my $new_rhs = xh::v::unbox $rhs;
87     die "illegal interpolation: $rhs" if $new_rhs eq $rhs;
88     $rhs = $new_rhs;
89 }
90
91 my $index = scope_index_for $prefix;
92 interpolate_wrap $prefix,
93     $$binding_stack[$index]{$rhs}
94     // $$binding_stack[0]{$rhs}
95     // die "unbound var: $rhs (bound vars are ["
96         . join(' ', sort keys %{$$binding_stack[$index]})
97         . "] locally, ["
98         . join(' ', sort keys %{$$binding_stack[0]})
99         . "] globally)";
100 }
101
102 sub interpolate {
103     my ($binding_stack, $x) = @_;
104     join '', map {$_ =~ /\^\$/ ? interpolate_dollar $binding_stack, $_
105         : $_ =~ /\^\$/ ? xh::v::undo_backslash_escape $_
106         : $_ } xh::v::split_by_interpolation $x;
107 }
108
109 sub call {
110     my ($binding_stack, $f, @args) = @_;
111     my $fn = $$binding_stack[-1]{$f}
112         // $$binding_stack[0]{$f}
113         // die "unbound function: $f";
114
115     # Special case: if it's a builtin Perl sub, then just call that directly.
116     return &$fn($binding_stack, @args) if ref $fn eq 'CODE';
117

```

```

118   # Otherwise use xh calling convention.
119   push @$binding_stack,
120       { _ => join ' ', map xh::v::quote_default($_), @args };
121
122   my $result = eval {evaluate $binding_stack, $fn};
123   my $error = "$@ in $f "
124       . join(' ', map xh::v::quote_default($_), @args)
125       . ' at calling stack depth ' . @$binding_stack
126       . " with locals:\n"
127       . join("\n", map " $_ -> $$binding_stack[-1]{$_}",
128           sort keys %{$$binding_stack[-1]}) if $@;
129   pop @$binding_stack;
130   die $error if $error;
131   $result;
132 }
133
134 sub evaluate {
135     my ($binding_stack, $body) = @_ ;
136     my @statements           = xh::v::parse_lines $body;
137     my $result               = '';
138
139     for my $s (@statements) {
140         my $original = $s;
141
142         # Step 1: Do we have a macro? If so, macroexpand before calling
143         # anything. (NOTE: technically incorrect; macros should receive their
144         # arguments with whitespace intact)
145         my @words;
146         while ((@words = xh::v::parse_words $s)[0] =~ /^#/ ) {
147             $s = eval {call $binding_stack, @words};
148             die "$@ in @words (while macroexpanding $original)" if $@;
149         }
150
151         # Step 2: Interpolate the whole command once. Note that we can't wrap
152         # each word at this point, since that would block interpolation
153         # altogether.
154         my $new_s = eval {interpolate $binding_stack, $s};
155         die "$@ in $s (while interpolating from $original)" if $@;
156         $s = $new_s;
157
158         # If that killed our value, then we have nothing to do.
159         next unless @words = xh::v::parse_words $s;
160
161         # Step 3: See if the interpolation produced multiple lines. If so, we
162         # need to re-expand. Otherwise we can do a single function call.
163         if (xh::v::parse_lines($s) > 1) {

```



```

164     $result = evaluate $binding_stack, $s;
165 } else {
166     # Just one line, so continue normally. At this point we look up the
167     # function and call it. If it's Perl native, then we're set; we just
168     # call that on the newly-parsed arg list. Otherwise delegate to
169     # create a new call frame and locals.
170     $result = eval {call $binding_stack, @words};
171     die "$@ in $s (while evaluating $original)" if $@;
172 }
173 }
174 $result;
175 }
176 _

```

Chapter 5

Globals

At this point we have the evaluator logic, but `xh` code can't do anything because it has no way to create variable bindings. This is solved by defining the `def` function and list/hash accessors.

Listing 5.1 `modules/globals.pl`

```
1 BEGIN {xh::defmodule('xh::globals.pl', <<'_')}  
2 sub def {  
3     my ($binding_stack, %args) = @_;  
4     $$binding_stack[-1]{$_} = $args{$_} for keys %args;  
5     join ' ', keys %args;  
6 }  
7  
8 sub echo {  
9     my ($binding_stack, @args) = @_;  
10    join ' ', @args;  
11 }  
12  
13 sub comment      {''}  
14 sub print_from_xh {print STDERR join(' ', @_[1 .. $#_]), "\n"}  
15  
16 sub perl_eval {  
17     my $result = eval $_[1];  
18     die "$@ while evaluating $_[1]" if $@;  
19     $result;  
20 }  
21  
22 sub assert_eq_macro {  
23     my ($binding_stack, $lhs, $rhs) = @_;  
24  
25     # We should get the same result by evaluating the LHS and RHS; otherwise  
26     # expand into a print statement describing the error.
```

```

27 my $expanded_lhs = xh::e::interpolate $binding_stack, $lhs;
28 my $expanded_rhs = xh::e::interpolate $binding_stack, $rhs;
29
30 $expanded_lhs eq $expanded_rhs
31 ? ''
32 : 'print ' . xh::v::quote_default("$lhs (-> $expanded_lhs)")
33   . ' != '
34   . xh::v::quote_default("$rhs (-> $expanded_rhs)");
35 }
36
37 # Create an interpreter instance that lets us interpret modules written in
38 # XH-script.
39 our $globals = [{def => \&def,
40                  echo => \&echo,
41                  print => \&print_from_xh,
42                  perl => \&perl_eval,
43                  '#' => \&comment,
44                  '#==' => \&assert_eq_macro}];
45
46 sub defglobals {
47     my %vals = @_;
48     $$globals[0]{$_} = $vals{$_} for keys %vals;
49 }
50
51 $xh::compilers{xh} = sub {
52     my ($module_name, $code) = @_;
53     eval {xh::e::evaluate $globals, $code};
54     die "error running $module_name: $@" if $@;
55 }
56 -

```

5.1 List accessors

List elements are accessed using single-character functions, one for each type of list.

Listing 5.2 modules/bootlist.pl

```

1 BEGIN {xh::defmodule('xh::bootlist.pl', <<'_'')}
2 sub wrap_negative {
3     my ($i, $n) = @_;
4     return undef unless length $i;
5     return $n + $i if $i < 0;
6     $i;
7 }
8

```

```

9  sub flexible_range {
10     my ($lower, $upper) = @_;
11     return reverse $upper .. $lower if $upper < $lower;
12     $lower .. $upper;
13 }
14
15 sub expand_subscript;
16 sub expand_subscript {
17     my ($subscript, $n) = @_;
18
19     return [map expand_subscript($_, $n),
20             xh::v::split_words xh::v::unbox $subscript]
21     if $subscript =~ /\{ /;
22
23     return [flexible_range wrap_negative($1, $n) // 0,
24             wrap_negative($2, $n) // $n - 1]
25     if $subscript =~ /^(~\d*):(-?\d*)$/;
26
27     return wrap_negative $subscript, $n if $subscript =~ /^- /;
28     $subscript;
29 }
30
31 sub dereference_one;
32 sub dereference_one {
33     my ($subscript, $boxed_list) = @_;
34
35     # List homomorphism of subscripts
36     return xh::v::quote_default
37         join ' ', map dereference_one($_, $boxed_list),
38         @$subscript if ref $subscript eq 'ARRAY';
39
40     # Normal numeric lookup, with empty string for out-of-bounds
41     return '' if $subscript =~ /^- /;
42     return $$boxed_list[$subscript] // '' if $subscript =~ /\d+ /;
43
44     if ($subscript =~ s/^\/ /) {
45         # In this case the boxed list should contain at least words, and
46         # probably whole lines. We word-parse each entry looking for the
47         # first subscript hit.
48         $subscript = xh::v::unbox $subscript;
49         for my $x (@$boxed_list) {
50             my @words = xh::v::parse_words $x;
51             return $x if $words[0] eq $subscript;
52         }
53         '';
54     } elsif ($subscript eq '#') {

```

```

55     scalar @$boxed_list;
56 } else {
57     die "unrecognized subscript form: $subscript";
58 }
59 }
60
61 sub dereference;
62 sub dereference {
63     my ($subscript, $boxed_list) = @_;
64     $subscript = xh::v::quote_as_word $subscript;
65     dereference_one expand_subscript($subscript, scalar(@$boxed_list)),
66                     $boxed_list;
67 }
68
69 sub index_lines {dereference $_[1], [xh::v::parse_lines $_[2]]}
70 sub index_words {dereference $_[1], [xh::v::parse_words $_[2]]}
71 sub index_path {dereference $_[1], [xh::v::parse_path $_[2]]}
72 sub index_bytes {dereference $_[1], [map ord, split //, $_[2]]}
73
74 sub update {
75     my ($subscript, $replacement, $join, $quote, $boxed_list) = @_;
76     my $expanded = expand_subscript $subscript, scalar @$boxed_list;
77
78     die "can't use list subscript for update: $subscript"
79     if ref $expanded eq 'ARRAY';
80
81     my @result;
82     for (my $i = 0; $i < @$boxed_list; ++$i) {
83         my ($k) = xh::v::parse_words $$boxed_list[$i];
84         push @result, $subscript eq $i || $subscript eq $k
85             ? $replacement
86             : $$boxed_list[$i];
87     }
88
89     if ($subscript =~ /\^d+$/ and $subscript > @$boxed_list) {
90         # It could be that we need to add something to the end.
91         for (my $i = @$boxed_list; $i < $subscript; ++$i) {
92             push @result, '';
93         }
94         push @result, $replacement;
95     }
96
97     join $join, map xh::v::quote_default($_), @result;
98 }
99
100 sub update_lines {update $_[1, 2], "\n", \&xh::v::quote_as_line,

```

```

101         [xh::v::parse_lines $_[3]]}
102
103     sub update_words {update @_[1, 2], ' ', \&xh::v::quote_as_word,
104                     [xh::v::parse_words $_[3]]}
105
106     sub update_path {update @_[1, 2], '', \&xh::v::quote_as_path,
107                     [xh::v::parse_path $_[3]]}
108
109     sub update_byte {update @_[1, 2], '', sub {$_[0]},
110                     [map ord, split //, $_[3]]}
111
112     sub quoted_fn {
113         my ($f) = @_;
114         sub {xh::v::quote_default &$f(@_)};
115     }
116
117     # TODO: fix duplication between quoted and unquoted functions. There
118     # should be some kind of sensible default that just works.
119     xh::globals::defglobals "" => \&index_lines, "'=" => \&update_lines,
120                             "@" => \&index_words, "@=" => \&update_words,
121                             ":" => \&index_path, ":=" => \&update_path,
122                             "\" => \&index_bytes, "\"=" => \&update_byte,
123
124                             "'!" => quoted_fn(\&index_lines),
125                             "@!" => quoted_fn(\&index_words),
126                             ":!" => quoted_fn(\&index_path),
127                             "\"!" => quoted_fn(\&index_bytes),
128
129                             "'="! => quoted_fn(\&update_lines),
130                             "@="! => quoted_fn(\&update_words),
131                             ":"="! => quoted_fn(\&update_path),
132                             "\"="! => quoted_fn(\&update_byte);
133
134     # Conversions between list forms.
135     sub list_to_list_fn {
136         my ($join, $quote, $parse) = @_;
137         sub {join $join, map &$quote($_), map &$parse($_), @_[1 .. $#_]};
138     }
139
140     my %joins = ("'" => "\"n", "@" => ' ', ":" => '/', "\"" => '');
141     my %quotes = ("'" => \&xh::v::quote_as_line,
142                  "@" => \&xh::v::quote_as_word,
143                  ":" => \&xh::v::quote_as_path,
144                  "\"" => sub {chr $_[0]});
145
146     my %parsers = ("'" => \&xh::v::parse_lines,

```

```

147         "@" => \&xh::v::parse_words,
148         ":" => \&xh::v::parse_path,
149         "\" => sub {map ord, split //, $_[0]}});
150
151     for my $k1 (keys %parsers) {
152         for my $k2 (keys %parsers) {
153             next if $k1 eq $k2;
154             my $fn = list_to_list_fn($joins{$k2}, $quotes{$k2}, $parsers{$k1});
155             xh::globals::defglobals "$k1$k2" => $fn,
156                                     "$k1$k2!" => quoted_fn $fn;
157         }
158     }
159
160     # Arglist collectors
161     for (keys %parsers) {
162         xh::globals::defglobals "_$_!" => quoted_fn
163                                     list_to_list_fn($joins{$_},
164                                                         $quotes{$_},
165                                                         sub {$_[0]});
166     }
167     -

```

Chapter 6

Bootstrap unit tests

This is our first layer of sanity checking for the interpreter. A failure here won't stop `xh` from running, but it will print a diagnostic message so we know something is up.

Listing 6.1 `modules/bootunit.xh`

```
1 # This is a comment and should work properly.
2 # {
3   This is a block comment and should also work.
4 }
5 #== 1 1
6
7 def test {
8   print testing $[_ @/0]
9   $'[_ @/1]
10 }
11
12 test basic-interpolation {
13   def foo bar
14     #== $@foo      bar
15     #== $@foo      {bar}
16     #== $@foo      (bar)
17     #== $@foo      [bar]
18     #== $foo        {{bar}}
19     #== $(echo $foo) {{bar}}
20     #== $@(echo $foo) bar
21   }
22
23 test subroutines {
24   def greet {
25     echo hi there, $@_
26   }
```



```

27     #== @(greet spencer)           {hi there, spencer}
28     #== @(greet spencer tipping) {hi there, spencer tipping}
29 }
30
31 test scoping {
32     def newdef {
33         # Define stuff within the calling scope; should be equivalent to using
34         # def.
35         echo $^(def $@_)
36     }
37     newdef x 5
38     #== $@x 5
39 }
40
41 test line-interpolation {
42     def x 5
43     def two-statements {
44         def x 10
45         echo $x
46     }
47     #== $@x 5
48     '$two-statements
49     #== $@x 10
50 }
51
52 test list-accessors {
53     def xs (foo bar bif baz)
54     #== $@(@ 0 $xs) foo
55     #== $@(@ 1 $xs) bar
56     #== $@(@ 2 $xs) bif
57     #== $@(@ 3 $xs) baz
58     #== $@(@ ^foo $xs) foo
59
60     test {$[]-expansion} {
61         #== $@[there echo/hi]           {hi there}
62         #== $@[spencer echo/there echo/hi] {hi there spencer}
63
64         #== $@[$^xs @/0]   foo
65         #== $@[$^xs @/-1]  baz
66         #== $@[$^xs @/-2]  bif
67         #== $@[$^xs @/:]   {foo bar bif baz}
68         #== $@[$^xs @/1:]  {bar bif baz}
69         #== $@[$^xs @/:1]  {foo bar}
70         #== $@[$^xs @/:-2] {foo bar bif}
71         #== $@[$^xs @/3:1] {baz bif bar}
72

```

```

73     #== $@[$^xs @/^bar] bar
74     #== $@[$^xs @/^bif] bif
75     #== $@[$^xs @/^notfound] {}
76
77     #== $@[$^xs @{0 2}]      {foo bif}
78     #== $@[$^xs @{0 2:}]    {foo {bif baz}}
79     #== $@[$^xs @{0 {2:}}]  {foo {{bif baz}}}
80 }
81 }
82
83 test associative-maps {
84     def associative {
85         foo bar
86         bif baz
87     }
88     #== $@[$associative '!'/^foo] {{      foo bar}}
89     #== $@[$associative '!'/^foo @/1] bar
90     #== $@[$associative '!'/^bif @/1] baz
91     #== $@[$associative '!'/^bok] {{{}}
92     #== $@[$associative '/^bok] {}
93
94     #== $@[$associative '/#] 4
95     #== $@[$associative @/#] 4
96 }
97
98 test byte-lists {
99     #== $@[abcd "/0] 97
100    #== $@[abcd "/1:3] {{98 99 100}}
101 }
102
103 test path-lists {
104     #== $@[/usr/bin/bash :(^/bin)] /bin
105     #== $@[../.. :/^..] ..
106 }
107
108 test macro-definition {
109     def #-> {echo #== \${@($@[$_ @/0])} [$$_ @/1]}
110     #-> {echo hi} hi
111 }

```

Chapter 7

REPL

A totally cheesy bootstrap repl for now. Later on this will be implemented in `xh-script`.

Listing 7.1 `modules/main.pl`

```
1 BEGIN {xh::defmodule('xh::main.pl', <<'_'>)}
2 sub main {
3   # This keeps xh from blocking on stdin when we ask it to compile itself.
4   /^--recompile$/ and return 0 for @ARGV;
5
6   my $list_depth = 0;
7   my $expression = '';
8   my $binding_stack = $xh::globals::globals;
9
10  print "xh\$ ";
11  while (my $line = <STDIN>) {
12    if (!($list_depth += xh::v::brace_balance $line)) {
13      # Collect the line and evaluate everything we have.
14      $expression .= $line;
15
16      my $result = eval {xh::e::evaluate $binding_stack, $expression};
17      print "error: $@\n" if length $@;
18      print "$result\n" if length $result;
19
20      $expression = '';
21      print "xh\$ ";
22    } else {
23      $expression .= $line;
24      print '> ' . ' ' x $list_depth;
25    }
26  }
27 }
```

