

# X shell

Spencer Tipping

February 23, 2014

# Contents

<b>I</b>	<b>Language reference</b>	<b>2</b>
1	Expansion syntax	3
<b>II</b>	<b>Bootstrap implementation</b>	<b>4</b>
2	Self-replication	5
3	Data structures	7
4	Evaluator	12
5	Globals	17
6	Bootstrap unit tests	23
7	REPL	27

## **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 $foo                # quote result into a word
xh$ echo $('foo              # flatten into multiple lines (be careful!)
xh$ echo @$foo               # flatten into multiple words (one line)
xh$ echo $:foo               # one path component
xh$ echo $"foo               # one braced list

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

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

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



```

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

```

```

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

```

```

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

```

```

164     $depth -= $piece =~ /\[\]\}]/;
165     return $s if $last_depth <= 0;
166     $last_depth = $depth;
167 }
168 $s =~ s/\s*\[\[{\}(.*)[\]\}]\s*$/1/sgr;
169 }
170 -

```

# 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_path            $unquoted if $prefix =~ '/:$/;
11    return xh::v::quote_default            $unquoted if $prefix =~ /"$/;
12    xh::v::quote_as_word $unquoted;
13 }
14
15 sub scope_index_for {
16     my ($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 $fn = xh::v::quote_as_word($f) =~ /\^\{\/ ? $f
112         : $$binding_stack[-1]{$f}
113         // $$binding_stack[0]{$f}
114         // die "unbound function: $f";
115
116     # Special case: if it's a builtin Perl sub, then just call that directly.
117     return &$fn($binding_stack, @args) if ref $fn eq 'CODE';

```

```

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

```



```

164     if (xh::v::parse_lines($s) > 1) {
165         $result = evaluate $binding_stack, $s;
166     } else {
167         # Just one line, so continue normally. At this point we look up the
168         # function and call it. If it's Perl native, then we're set; we just
169         # call that on the newly-parsed arg list. Otherwise delegate to
170         # create a new call frame and locals.
171         $result = eval {call $binding_stack, @words};
172         die "$@ in $s (while evaluating $original)" if $@;
173     }
174 }
175 $result;
176 }
177 -

```

## 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 xh::v::quote_as_word '' if $subscript =~ /^- /;
42     return $$boxed_list[$1] // '' if $subscript =~ /^(\d+)!$/;
43
44     return xh::v::quote_as_word $$boxed_list[$subscript] // ''
45     if $subscript =~ /^~\d+$/;
46
47     if ($subscript =~ s/^~ /) {
48         # In this case the boxed list should contain at least words, and
49         # probably whole lines. We word-parse each entry looking for the
50         # first subscript hit.
51         $subscript = xh::v::unbox $subscript;
52         for my $x (@$boxed_list) {
53             my @words = xh::v::parse_words $x;
54             return xh::v::quote_as_word $x if $words[0] eq $subscript;

```

```

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

```

```

101
102     xh::v::quote_as_word join $join, map &$quote($_), @result;
103 }
104
105 sub update_lines {update @_ [1, 2], "\n", \&xh::v::quote_as_line,
106                     [xh::v::parse_lines $_[3]]}
107
108 sub update_words {update @_ [1, 2], ' ', \&xh::v::quote_as_word,
109                     [xh::v::parse_words $_[3]]}
110
111 sub update_path {update @_ [1, 2], '', \&xh::v::quote_as_path,
112                     [xh::v::parse_path $_[3]]}
113
114 sub update_byte {update @_ [1, 2], '', sub {$_[0]},
115                     [map ord, split //, $_[3]]}
116
117 xh::globals::defglobals "" => \&index_lines, "'=" => \&update_lines,
118                        "@" => \&index_words, "@=" => \&update_words,
119                        ":" => \&index_path, ":=" => \&update_path,
120                        "\"" => \&index_bytes, "\"=" => \&update_byte;
121
122 # Conversions between list types.
123 sub list_to_list_fn {
124     my ($join, $quote, $parse) = @_;
125     sub {xh::v::quote_as_word
126         join $join, map &$quote($_), map &$parse($_), @_ [1 .. $#_]};
127 }
128
129 my %joins = ("'" => "\n", "@" => ' ', ":" => '/', "\"" => '');
130 my %quotes = ("'" => \&xh::v::quote_as_line,
131               "@" => \&xh::v::quote_as_word,
132               ":" => \&xh::v::quote_as_path,
133               "\"" => sub {chr $_[0]});
134
135 my %parsers = ("'" => \&xh::v::parse_lines,
136               "@" => \&xh::v::parse_words,
137               ":" => \&xh::v::parse_path,
138               "\"" => sub {map ord, split //, $_[0]});
139
140 for my $k1 (keys %parsers) {
141     for my $k2 (keys %parsers) {
142         next if $k1 eq $k2;
143         my $fn = list_to_list_fn($joins{$k2}, $quotes{$k2}, $parsers{$k1});
144         xh::globals::defglobals "$k1$k2" => $fn;
145     }
146 }

```

```
147
148 sub explode {xh::v::unbox $_[1]}
149 xh::globals::defglobals '!' => \&explode;
150 _
```

## 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   def perftime {perl {use Time::HiRes qw/time/; time}}
9   def start-time $(perftime)
10  '$[_ @/1!]
11  def end-time $(perftime)
12  print tested $[_ @/0] in $(perl (($end-time - $start-time) * 1000)) ms
13 }
14
15 test basic-interpolation {
16   def foo bar
17     #== $@foo      bar
18     #== $@foo      {bar}
19     #== $@foo      (bar)
20     #== $@foo      [bar]
21     #== $foo       bar
22     #== $(echo $foo) bar
23     #== $@(echo $foo) bar
24   }
25
26 test subroutines {
```



```

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

```

```

73     #== $@[$^xs @/-2]  bif
74     #== $@[$^xs @/:]   {{foo bar bif baz}}
75     #== $@[$^xs @/1:]  {{bar bif baz}}
76     #== $@[$^xs @/:1]  {{foo bar}}
77     #== $@[$^xs @/-2]  {{foo bar bif}}
78     #== $@[$^xs @/3:1] {{baz bif bar}}
79
80     #== $@[$^xs @/^bar] bar
81     #== $@[$^xs @/^bif] bif
82     #== $@[$^xs @/^notfound] {}
83
84     #== $@[$^xs @{0 2}]   {{foo bif}}
85     #== $@[$^xs @{0 2:}] {{foo {bif baz}}}
86     #== $@[$^xs @{0 {2:}}] {{foo {{bif baz}}}}
87   }
88 }
89
90 test list-updaters {
91   def xs (a b c d)
92     #== $@[$xs @=/0/b !] {b b c d}
93     #== $@[$xs @=-1/a !] {a b c a}
94 }
95
96 test associative-maps {
97   def associative {
98     foo bar
99     bif baz
100  }
101   #== $@[$associative '/^foo] {{    foo bar}}
102   #== $@[$associative '/^foo @/1] bar
103   #== $@[$associative '/^bif @/1] baz
104   #== $@[$associative '/^bok] {}
105
106   #== $@[$associative '/#] 4
107   #== $@[$associative @/#] 4
108
109   #== $@[$associative '=/^foo[FOO BAR] '/^FOO @/1] BAR
110 }
111
112 test byte-lists {
113   #== $@[abcd "/0] 97
114   #== $@[abcd "/1:3] {{98 99 100}}
115 }
116
117 test path-lists {
118   #== $@[/usr/bin/bash :(^/bin)] /bin

```

```
119   #== $@[../* :/^..] ..
120 }
121
122 test macro-definition {
123   def #-> {echo #== \${$@[$_ @/0]} ${$_ @/1}}
124   #-> {echo hi} hi
125 }
```

# 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 }
```

