

X shell

Spencer Tipping

March 6, 2014

Contents

1	Current issues	2
I	Language reference	4
2	Expansion syntax	5
II	Bootstrap implementation	6
3	Self-replication	7
4	Data structures	9
5	Evaluator	14
6	Globals	19
7	Bootstrap unit tests	26
8	REPL	31
III	xh standard library	33
9	Function functions	34
10	List functions	35
11	Math macro	36

Chapter 1

Current issues

Sorted by roughly descending priority.

1.1 Design problems

1. No support for scope extension/inheritance, so all sub-scopes with new variables end up polluting the parents. (Let's actually assume that everything having to do with scope referencing is totally horked.)
2. Interpolation for closures is really obnoxious because you have to escape everything you're not interpolating. It's also impossible to define some sort of identity where `$x` expands to `$x` for undefined variables, since some expansions are function calls rather than simple values.

It's probably possible to fix this using some kind of namespaced quote/unquote operators – but it's unclear what the syntax for that should be. This feature also wouldn't address the normal function-call case because it doesn't delay expansion for later statements or for macros; this is probably a by-product of the fact that interpolation is defined independently of evaluation.

There are a few ways to solve this:

- (a) Make argument interpolation a first-class idea (dangerous)
- (b) Remove macros altogether
- (c) Decouple macros from runtime state
- (d) Impose some kind of monotonicity on local bindings

Beyond referencing lexical closure data, there's the added issue of modifying it. Any substitution-based strategy makes it difficult to have mutable closure state.¹

¹TODO: do we want to support this?

3. Macros may be too powerful. They can see/change runtime state at expansion-time, and it isn't clear what the purpose of this is beyond just having the ability to inspect arguments without quoting. They also complicate the evaluation model.
4. It isn't obvious what kind of list should be used for what purpose.
It's not clear to me that this is a solvable problem. The degrees of freedom within most data formats are specifically designed to accommodate alternative layouts; whitespace-independence is generally considered a virtue.
Does it suffice to use word lists by default and treat everything else as a way to parse external data?
5. It also isn't obvious what kind of bracket should be used.
6. A lot of commonly-used interpolation constructs involve shift-key contention while typing.
7. There is no way to define expression-macros; e.g. `$0` to mean `[$_ @/0]`.
8. Calling a single-word anonymous function with no arguments causes arguments to be added spuriously.
9. Associative list retrieval is a hack in that it doesn't work across list levels. This probably gets back to 4.

Part I

Language reference

Chapter 2

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}      # transpose interpolation: {foo bar bif}

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 3

Self-replication

Listing 3.1 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 3.2 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 4

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 4.1 `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     # 5. We hit a backslash.
127
128     if ($sublist_depth < $interpolating_depth
129         or $sublist_depth == $interpolating_depth
130         and $piece eq '/' || $piece =~ /^\\s/) {
131         # No longer interpolating because of what we just saw, so emit
132         # current item and start a new constant piece.
133         push @result, $current_item if length $current_item;
134         $current_item = $piece;
135         $interpolating = 0;
136     } elsif ($sublist_depth == $interpolating_depth
137         && $closed_something) {
138         push @result, "$current_item$piece";
139         $current_item = '';
140         $interpolating = 0;
141     } elsif ($sublist_depth == $interpolating_depth && $piece =~ /^\\\/) {
142         push @result, $current_item if length $current_item;
143         $current_item = $piece;
144         $interpolating = 0;
145     } else {
146         # Still interpolating, so collect the piece.
147         $current_item .= $piece;
148     }
149 }
150 }
151
152 push @result, $current_item if length $current_item;
153 @result;
154 }
155
156 sub undo_backslash_escape {
157     return "\n" if $_[0] eq '\n';
158     return "\t" if $_[0] eq '\t';
159     return "\\" if $_[0] eq '\\\\';
160     substr $_[0], 1;
161 }
162
163 sub unbox {

```

```

164  my ($s) = @_;
165  my $depth      = 0;
166  my $last_depth = 1;
167  for my $piece (grep length, split /(\.|\[|\]|{|\})/, $s) {
168      $depth += $piece =~ /\[({)/;
169      $depth -= $piece =~ /\]|})/;
170      return $s if $last_depth <= 0;
171      $last_depth = $depth;
172  }
173  $s =~ s/^\s*\[({(.*?)\])\]\s*$/$1/sgr;
174  }
175  _

```

Chapter 5

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 5.1 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 = $initial =~ /^([^\]]+)$/sg;
70     $initial = call $calling_stack,
71                   (map {s/^\/\//r} $fns),

```



```

72             xh::v::parse_words $initial
73     for @fns;
74
75     return interpolate_wrap $prefix, $initial;
76 } elsif ($rhs =~ /\^\{\/) {
77     # Interpolated quotation, possibly under a different scope index.
78     my $index      = scope_index_for $prefix;
79     my $calling_stack = truncated_stack $binding_stack, $index;
80
81     return interpolate_wrap $prefix,
82         interpolate $calling_stack, xh::v::unbox $rhs;
83 } else {
84     # It's either a plain word or another $-term. Either way, go ahead and
85     # interpolate it so that it's ready for this operator.
86     $rhs = xh::v::unbox interpolate $binding_stack, $rhs;
87
88     my $index = scope_index_for $prefix;
89     interpolate_wrap $prefix,
90         $$binding_stack[$index]{$rhs}
91         // $$binding_stack[0]{$rhs}
92         // die "unbound var: [$rhs] (bound vars are ["
93             . join(' ', sort keys %{$$binding_stack[$index]})
94             . "] locally, ["
95             . join(' ', sort keys %{$$binding_stack[$index - 1]})
96             . " ] in parent stack, ["
97             . join(' ', sort keys %{$$binding_stack[0]})
98             . "] globally)";
99 }
100 }
101
102 sub interpolate {
103     my ($binding_stack, $x) = @_;
104     join '', map {$_ =~ /\^\$/ ? interpolate_dollar $binding_stack, $_
105         : $_ =~ /\^\$/ ? xh::v::undo_backslash_escape $_
106         : $_ } xh::v::split_by_interpolation $x;
107 }
108
109 sub call {
110     my ($binding_stack, $f, @args) = @_;
111     my $fn = xh::v::quote_as_word($f) =~ /\^\{\/ ? $f
112         : $$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 6

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 6.1 `modules/globals.pl`

```
1 BEGIN {xh::defmodule('xh::globals.pl', <<'_'')}
2 sub def {
3   my ($binding_stack, $n, %args) = @_;
4   $$binding_stack[-$n]{$_} = $args{$_} for keys %args;
5   join ' ', keys %args;
6 }
7
8 sub local_def {def $_[0], 1, @_[1..$#_]}
9
10 sub echo {
11   my ($binding_stack, @args) = @_;
12   join ' ', map xh::v::quote_as_word($_), @args;
13 }
14
15 sub comment      {' '}
16 sub print_from_xh {print STDERR join(' ', @_[1 .. $#_]), "\n"; ''}
17
18 sub perl_eval {
19   my $result = eval $_[1];
20   die "$@ while evaluating $_[1]" if $@;
21   $result;
22 }
23
24 sub assert_eq_macro {
25   my ($binding_stack, $lhs, $rhs) = @_;
26 }
```

```

27  # We should get the same result by evaluating the LHS and RHS; otherwise
28  # expand into a print statement describing the error.
29  my $expanded_lhs = xh::e::interpolate $binding_stack, $lhs;
30  my $expanded_rhs = xh::e::interpolate $binding_stack, $rhs;
31
32  $expanded_lhs eq $expanded_rhs
33  ? ''
34  : 'print ' . xh::v::quote_default("$lhs (-> $expanded_lhs)")
35    . ' != '
36    . xh::v::quote_default("$rhs (-> $expanded_rhs)");
37 }
38
39 sub xh_if {
40     my ($binding_stack, $cond, $then, $else) = @_;
41     xh::e::evaluate $binding_stack, length $cond ? $then : $else;
42 }
43
44 sub xh_while {
45     my ($binding_stack, $cond, $body) = @_;
46     my $result;
47     $result = xh::e::evaluate $binding_stack, $body
48         while length xh::e::evaluate $binding_stack, $cond;
49     $result;
50 }
51
52 sub xh_not {
53     my ($binding_stack, $v) = @_;
54     length $v ? '' : '{}';
55 }
56
57 sub xh_eq {
58     my ($binding_stack, $x, $y) = @_;
59     $x eq $y ? "{" . xh::v::quote_as_word($x) . "}" : '';
60 }
61
62 sub xh_matches {
63     # NOTE: leaky abstraction (real xh regexps won't support all of the perl
64     # extensions)
65     my ($binding_stack, $pattern, $s) = @_;
66     $s =~ /$pattern/ ? "{" . xh::v::quote_as_word($s) . "}" : '';
67 }
68
69 sub escalate {
70     my ($binding_stack, $levels, $body) = @_;
71     xh::e::evaluate xh::e::truncated_stack($binding_stack, -($levels + 1)),
72         $body;

```

```

73 }
74
75 # Create an interpreter instance that lets us interpret modules written in
76 # XH-script.
77 our $globals = [{def    => \&local_def,
78                    '^def' => \&def,
79                    '^'    => sub {escalate $_[0], 1, @_[1..$_]},
80                    '^'    => sub {escalate $_[0], 2, @_[1..$_]},
81                    '^'    => sub {escalate $_[0], 3, @_[1..$_]},
82                    echo   => \&echo,
83                    print  => \&print_from_xh,
84                    perl   => \&perl_eval,
85                    if     => \&xh_if,
86                    while  => \&xh_while,
87                    not    => \&xh_not,
88                    '=='   => \&xh_eq,
89                    '=~'   => \&xh_matches,
90                    '#'    => \&comment,
91                    '#=='  => \&assert_eq_macro}];
92
93 sub defglobals {
94     my %vals = @_;
95     $$globals[0]{$_} = $vals{$_} for keys %vals;
96 }
97
98 $xh::compilers{xh} = sub {
99     my ($module_name, $code) = @_;
100     eval {xh::e::evaluate $globals, $code};
101     die "error running $module_name: $@" if $@;
102 }
103 -

```

6.1 List accessors

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

Listing 6.2 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 outer_lines {dereference $_[1], [xh::v::split_lines $_[2]]}
78 sub outer_words {dereference $_[1], [xh::v::split_words $_[2]]}
79 sub outer_path {dereference $_[1], [xh::v::split_path $_[2]]}
80
81 sub update {
82     my ($subscript, $replacement, $join, $quote, $boxed_list) = @_;
83     my $expanded = expand_subscript $subscript, scalar @$boxed_list;
84
85     die "can't use list subscript for update: $subscript"
86     if ref $expanded eq 'ARRAY';
87
88     my $associative = $expanded =~ s/^\^//;
89
90     my @result;
91     for (my $i = 0; $i < @$boxed_list; ++$i) {
92         my ($k) = xh::v::parse_words $$boxed_list[$i];
93         push @result, ($associative ? $expanded eq $k : $expanded eq $i)
94             ? $replacement
95             : $$boxed_list[$i];
96     }
97
98     if ($expanded =~ /\^d+$/ and $expanded > @$boxed_list) {
99         # It could be that we need to add something to the end.

```



```

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

```

```

146         "\" => sub {map ord, split //, $_[0]});
147
148     for my $k1 (keys %parsers) {
149         for my $k2 (keys %parsers) {
150             next if $k1 eq $k2;
151             my $fn = list_to_list_fn($joins{$k2}, $quotes{$k2}, $parsers{$k1});
152             xh::globals::defglobals "$k1$k2" => $fn;
153         }
154     }
155
156     sub explode {xh::v::unbox $_[1]}
157     xh::globals::defglobals '!' => \&explode;
158     -

```

6.2 Double-precision math

These functions are low-level and are usually called by generated code rather than by hand. See [11](#) for a macro that does this.

Listing 6.3 modules/bootmath.pl

```

1 BEGIN {xh::defmodule('xh::math.pl', <<'_'')}
2 sub binary_to_nary {
3     my ($f, $zero) = @_;
4     sub {
5         my ($bindings, $x, @args) = @_;
6         return $zero unless defined $x;
7         return &$f($zero, $x) unless @args;
8         $x = &$f($x, $_) for @args;
9         $x;
10    };
11 }
12
13 xh::globals::defglobals
14     "math$_" => binary_to_nary(eval "sub {\$_[0] $_ \$_[1]}", /^[*\/]$/);
15 for qw[+ - * / & | ! < > << >>];
16 -

```

Chapter 7

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 7.1 modules/bootunit.xh

```
1 def test {
2   def perltime {perl {use Time::HiRes qw/time/; time}}
3   def start-time $(perltime)
4   '$[_ @/1!]
5   def end-time $(perltime)
6   print tested $[_ @/0] in $(math* 1000 $(math- $end-time $start-time)) ms
7 }
8
9 test everything {
10  # This is a comment and should work properly.
11  # {
12    This is a block comment and should also work.
13  }
14  #== 1 1
15
16  test basic-interpolation {
17    def foo bar
18    #== $@foo      bar
19    #== $@foo      {bar}
20    #== $@foo      (bar)
21    #== $@foo      [bar]
22    #== $foo       bar
23    #== $(echo $foo) bar
24    #== @(echo $foo) bar
25  }
26
```

```

27 test subroutines {
28     def greet {
29         echo hi there, $_
30     }
31     #== @(greet spencer)          {hi there, spencer}
32     #== @(greet spencer tipping) {hi there, spencer tipping}
33
34     # Also anonymous functions:
35     #== @($greet spencer)        {hi there, spencer}
36     #== @({echo hi $_} spencer) {hi spencer}
37 }
38
39 test scoping {
40     def newdef {
41         # Define stuff within the calling scope; should be equivalent to
42         # using def.
43         echo $^(def $_)
44     }
45     newdef x 5
46     #== $@x 5
47 }
48
49 test line-interpolation {
50     def x 5
51     def two-statements {
52         def x 10
53         echo $x
54     }
55     #== $@x 5
56     $'two-statements
57     #== $@x 10
58 }
59
60 test outer-interpolation {
61     def get-5-plus {
62         echo $(math+ [$ _ @/0] 5)
63     }
64     def inner {
65         echo $^(get-5-plus 10)
66     }
67     #== $(inner) 15
68 }
69
70 test list-accessors {
71     def xs (foo bar bif baz)
72     #== $@(@ 0 $xs) foo

```

```

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

```

```

119     foo bar
120     bif baz
121 }
122 #== $@[$associative '/^foo @(0 1)] {{foo bar}}
123 #== $@[$associative '/^foo @/1] bar
124 #== $@[$associative '/^bif @/1] baz
125 #== $@[$associative '/^bok] {}
126
127 #== $@[$associative '/#] 4
128 #== $@[$associative @/#] 4
129
130 #== $@[$associative '=/^foo[FOO BAR] '/^FOO @/1] BAR
131 }
132
133 test byte-lists {
134     #== $@[abcd "/0] 97
135     #== $@[abcd "/1:3] {{98 99 100}}
136 }
137
138 test path-lists {
139     #== $@[/usr/bin/bash :(^/bin)] /bin
140     #== $@[./... :/^..] ..
141 }
142
143 test macro-definition {
144     def #-> {echo #== \${@($@[$_ @/0])} ${$_ @/1}}
145     #-> {echo hi} hi
146 }
147
148 test equality-comparison {
149     def x 10
150     #== $(== $x 10) {{10}}
151     #== $(== {} {}) {{{}}
152     #== $(== $x 9) {}
153
154     #== $(not $(== $x 9)) {{{}}
155     #== $(not $(== $x 10)) {}
156 }
157
158 test conditions {
159     def x 5
160     if $(== $x 5) {def x 6} {def x 7}
161     #== $x 6
162     if $(== $x 5) {def x 8} {def x 9}
163     #== $x 9
164 }

```

```

165
166 test iteration {
167     def i      0
168     def count 0
169     while {not $(== $i 10)} {
170         def i      $(math+ $i      1)
171         def count $(math+ $count 1)
172     }
173     #== $count 10
174     #== $i      10
175 }
176
177 test float-math {
178     #== $(math+ 0 1) 1
179     #== $(math/ 4 8) 0.5
180     #== $(math< 3 4) 1
181     #== $(math< 5 4) {}
182 }
183 }

```

Chapter 8

REPL

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

Listing 8.1 `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 }
```


Part III

xh standard library

Chapter 9

Function functions

Listing 9.1 modules/fn.xh

```
1 def defn {
2   def fname [$ _ @/0]
3   def args  [$ _ @/1]
4   def body  [$ _ @/2]
5
6   def i 0
7   def argdefs {}
8   while {math< $i [$args @/#]} {
9     def argdefs ['$argdefs \n def [$args @/$i] \[$[_ @/$i]]
10    def i $(math+ $i 1)
11  }
12
13  ^ ${def $fname ${'$argdefs \n $'body}}
14 }
```

Chapter 10

List functions

The usual suspects.

Listing 10.1 modules/list.xh

```
1 defn @each [v f xs] {
2   def i 0
3   def n [$xs @/#]
4   def r {}
5   while {math< $i $n} {
6     def r [$^(def $v [$xs @/$i] \n $'f)]
7     def i $(math+ $i 1)
8   }
9   echo $@r
10 }
11
12 defn @m [f xs] {
13   def ys {}
14   @each x {def ys [$@ys $(f $x)] \n echo $ys} $xs
15 }
16
17 defn @r [f zero xs] {
18   @each x {def zero $(f $zero $x) \n echo $zero} $xs
19 }
20
21 #== $[[1 2 3] @m{math+ 1 $_} @/:] {{2 3 4}}
22 #== $[[1 2 3] @r/math+/0] 6
```

Chapter 11

Math macro

The math macro converts infix math to low-level prefix instructions.

Listing 11.1 modules/math.xh

```
1 def operator-precedence {</4 >/4 +/3 -/3 */2 {/}/2}  
2  
3 #== [$operator-precedence @/^< :/1] 4  
4 #== [$operator-precedence @/^{/} :/1] 4  
5  
6 def #m {  
7   # TODO  
8 }
```