

X shell

Spencer Tipping

March 20, 2014

Contents

I	Language reference	2
1	Similarities to TCL	3
2	Similarities to Lisp	5
3	Dissimilarities from everything else I know of	7
4	Functions	9
II	Bootstrap implementation	10
5	Self-replication	11
6	Perl-hosted evaluator	13

Part I

Language reference

Chapter 1

Similarities to TCL

Every xh value is a string. This includes functions, closures, lazy expressions, scope chains, call stacks, and heaps. Asserting string equivalence makes it possible to serialize any value losslessly, including a running xh process.¹

Although the string equivalence is available, most operations have higher-level structure. For example, the `$` operator, which performs string interpolation, interpolates values in such a way that two things are true:

1. No interpolated value will be further interpolated (idempotence).
2. The interpolated value will be read as a single list element.

For example:

```
(def bar bif)
(def foo "hi there \${bar}")
(def baz $foo)                # no quoting necessary here by (2)
(echo $baz)
hi there $bar!                # $bar unevaluated by (1)
()
```

This interpolation structure can be overridden by using one of three alternative forms of `$`:

```
(def bar bif)
(def foo "hi there \${bar}")
(echo ${!foo})                # allow re-interpolation
hi there bif!
(count [$foo])                # single element
1
(count [@$foo])               # multiple elements
```

¹Note that things like active socket connections and external processes will be proxied, however; xh can't migrate system-native things.

```

3
(nth [$@!foo] 2)          # multiple and re-interpolation
bif!
()

```

All string values in xh programs are lifted into reader-safe quotations. This causes any “active” characters such as \$ to be prefixed with backslashes, a transformation you can mostly undo by using \$@!. The only thing you can’t undo is bracket balancing, which if undone would wreak havoc on your programs. You can see the effect of balancing by doing something like this:

```

(def foo "[[ [")
(def bar [$@!foo])
(echo $bar)
[\[\[ \[\[
()

```

We can’t get xh to create an unbalanced list through any series of rewriting operations, since the contract is that any active list characters are either positive and balanced, or escaped.

Chapter 2

Similarities to Lisp

xh is strongly based on the Lisp family of languages, most visibly in its homiconicity. Any string wrapped in curly braces is a list of lines with the following contract (TODO: rewrite this section):

```
$ def foo {bar bif
>      baz
>      bok quux}
$ count $foo
3
$ nth $foo 0
bar bif
$ nth $foo 2
bok quux
$ def foo {
>  bar bif
>  baz
>  bok quux
> }
$ count $foo
3
$ nth $foo 0
bar bif
$
```

Any string wrapped in [] or () is interpreted as a list of words, just as it is in Clojure. Also as in Lisp in general, () interpolates its result into the surrounding context:

```
$ def foo 'hi there'
$ echo $foo
hi there
$ echo (echo $foo)           # similar to bash's $()
```

```
hi there
$
```

Any `()` list can be prefixed with `@` and/or `!` with effects analogous to `$`;
e.g. `echo !@(echo hi there)`.

Chapter 3

Dissimilarities from everything else I know of

xh evaluates expressions outside-in:

1. Variable shadowing is not generally possible.
2. Expansion is idempotent for any set of bindings.
3. Unbound variables expand to active versions of themselves (a corollary of 2).
4. Laziness is implemented by referring to unbound quantities.
5. Bindings can be arbitrary list expressions, not just names (a partial corollary of 4).
6. No errors are ever thrown; all expressions that cannot be evaluated become (error) clauses that most functions consider to be opaque.
7. xh has no support for syntax macros.

Unbound names are treated as though they might at some point exist. For example:

```
$ echo $x
$x
$ def x $y
$ echo $x
$y
$ def y 10
$ echo $x
10
$
```


You can also bind expressions of things to express partial knowledge:

```
$ echo (count $str)
(count $str)
$ def (count $str) 10
$ echo $str
$str
$ echo (count $str)
10
$
```

This is the mechanism by which xh implements lazy evaluation, and it's also the reason you can serialize partially-computed lazy values.

Chapter 4

Functions

xh supports two equivalent ways to write function-like relations:

```
$ def (foo $x) {echo hi there, $x!}  
$ foo spencer  
hi there, spencer!  
$
```

This is named definition by destructuring, which works great for most cases. When you're writing an anonymous function, however, you'll need to describe the mappings individually:

```
$ reduce {[$total +$x] + $total $x  
          [$total *$x] * $total $x} \  
0 \  
[+1 +2 *5 +1]  
16  
$
```

Part II

Bootstrap implementation

Chapter 5

Self-replication

Listing 5.1 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 5.2 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 6

Perl-hosted evaluator

xh is self-hosting, but to get there we need to implement an interpreter in Perl. This interpreter is mostly semantically correct but slow and shouldn't be used for anything besides bootstrapping the real compiler.

Listing 6.1 modules/interpreter.pl

```
1 BEGIN {xh::defmodule('xh::interpreter.pl', <<'_'')}
2 use Memoize qw/memoize/;
3 use List::Util qw/max/;
4
5 sub active_regions {
6     # Returns a series of numbers that describes, in pre-order, regions of
7     # the given string that should be interpolated. The numeric list has the
8     # following format:
9     #
10    # (offset << 32 | len), (offset << 32 | len) ...
11
12    my @pieces = split /(\.|\$@?!?\w+|\$@?!?\{[^\}]+\}|@?!?\(|['\]))/s, $_[0];
13    my $offset = 0;
14    my @result;
15    my @quote_offsets;
16
17    for (@pieces) {
18        if (@quote_offsets && substr($_[0], $quote_offsets[-1], 1) eq '"') {
19            # We're inside a hard-quote, so ignore everything except for the next
20            # hard-quote.
21            pop @quote_offsets if /^'/;
22        } else {
23            if (/^'/ || /^@?!?\(/) {
24                push @quote_offsets, $offset;
25            } elsif (/^\$/ ) {
26                push @result, $offset << 32 | length;
```

```

27         } elsif (/^\)/) {
28             my $start = pop @quote_offsets;
29             push @result, $start << 32 | $offset + 1 - $start;
30         }
31     }
32     $offset += length;
33 }
34
35 sort {$a <=> $b} @result;
36 }
37
38 memoize 'active_regions';
39
40 our $whitespace = qr/\s+/;
41 our $newlines   = qr/\n(?:\s*\n)*/;
42 our %closers    = ('(' => ')', '[' => ']', '{' => '}');
43
44 sub element_regions {
45     # Returns integer-encoded regions describing the positions of list
46     # elements. The list passed into this function should be unwrapped; that
47     # is, it should have no braces.
48     my ($is_vertical, $xs) = @_;
49     my $split_on           = $is_vertical ? $newlines : $whitespace;
50     my $offset             = 0;
51     my @pieces             = split / ( "(?:\\.|[^\"])*"
52                               | "(?:\\.|[^\'])*"
53                               | \\.
54                               | [({\[\]\}])
55                               | $split_on ) /xs, $_[0];
56     my @paren_offsets;
57     my @parens;
58     my @result;
59     my $item_start = -1;
60
61     for (@pieces) {
62         unless (@paren_offsets) {
63             if (/ $split_on / || /^[\]\}]/) {
64                 # End any item if we have one.
65                 push @result, $item_start << 32 | $offset - $item_start
66                 if $item_start >= 0;
67                 $item_start = -1;
68             } else {
69                 # Start an item unless we've already done so.
70                 $item_start = $offset if $item_start < 0;
71             }
72         }

```

```

73
74     # Update bracket tracking.
75     if ($_ eq $closers{$parens[-1]}) {
76         if (@parens) {
77             pop @paren_offsets;
78             pop @parens;
79         } else {
80             die 'illegal closing brace: ... '
81                 . substr($xs, max(0, $offset - 10), 20)
82                 . ' ...'
83                 . "\n(whole string is $xs)";
84         }
85     } elsif (/^\([\{\}]/) {
86         push @paren_offsets, $offset;
87         push @parens, $_;
88     }
89
90     $offset += length;
91 }
92
93 push @result, $item_start << 32 | $offset if $item_start >= 0;
94 @result;
95 }
96
97 memoize 'element_regions';
98
99 sub xh_list_box {
100     $_[0] !~ /\[({\[}/ && element_regions(0, $_[0]) > 1
101     ? "[$_[0]"
102     : $_[0];
103 }
104
105 sub xh_list_unbox {
106     return $1 if $_[0] =~ /\[({(.*)\})$/
107         || $_[0] =~ /\[({(.*)\})$/
108         || $_[0] =~ /\[({(.*)\})$/;
109     $_[0];
110 }
111
112 sub parse_list {
113     my $unboxed = xh_list_unbox $_[0];
114     map xh_list_box(substr $unboxed, $_ >> 32, $_ & 0xffffffff),
115         element_regions 0, $unboxed;
116 }
117
118 sub parse_block {

```



```

119     my $unboxed = xh_list_unbox $_[0];
120     map xh_list_box(substr $unboxed, $_ >> 32, $_ & 0xffffffff),
121         element_regions 1, $unboxed;
122 }
123
124 sub into_list {'(' . join(' ', map xh_list_box($_), @_) . ')'}
125 sub into_vec {'[' . join(' ', map xh_list_box($_), @_) . ']'}
126 sub into_block {'{' . join("\n", @_) . '}' }
127
128 sub xh_vecp {$_[0] =~ /\^[.]*$/}
129 sub xh_listp {$_[0] =~ /\^(.)*$/}
130 sub xh_blockp {$_[0] =~ /\^{.}*$/}
131 sub xh_varp {$_[0] =~ /\$$/}
132
133 sub xh_count {
134     scalar element_regions 0, xh_list_unbox $_[0];
135 }
136
137 sub xh_nth {(parse_list $_[0])[$_[1]]}
138
139 sub xh_nth_eq {
140     # FIXME
141     my ($copy, $i, $v) = @_;
142     my @regions = element_regions 0, $copy;
143     my $r = $regions[$i];
144     substr($copy, $r >> 32, $r & 0xffffffff) = $v;
145     $copy;
146 }
147
148 sub xh_vcount {
149     scalar element_regions 1, xh_list_unbox $_[0];
150 }
151
152 sub xh_vnth {
153     my @regions = element_regions 1, $_[0];
154     my $r = $regions[$_[1]];
155     xh_list_box substr $_[0], $r >> 32, $r & 0xffffffff;
156 }
157
158 sub xh_vnth_eq {
159     my ($copy, $i, $v) = @_;
160     my @regions = element_regions 1, $copy;
161     my $r = $regions[$i];
162     substr($copy, $r >> 32, $r & 0xffffffff) = $v;
163     $copy;
164 }

```

```

165
166 sub destructuring_bind;
167 sub destructuring_bind {
168     # Both $pattern and $v should be quoted; that is, the string character [
169     # should be encoded as \[.
170     my ($pattern, $v) = @_;
171     my @pattern_elements = element_regions 0, $pattern;
172     my @v_elements      = element_regions 0, $v;
173     my %bindings;
174
175     # NOTE: no $@ matching
176     return undef unless @v_elements == @pattern_elements;
177
178     # NOTE: no foo$bar matching (partial constants)
179     for (my $i = 0; $i < @pattern_elements; ++$i) {
180         my $pi = xh_nth $pattern, $i;
181         my $vi = xh_nth $v,      $i;
182
183         return undef if $pi !~ /^$/ && $pi ne $vi;
184
185         my @pattern_regions = element_regions 0, $pi;
186         my @v_regions      = element_regions 0, $vi;
187         return undef unless @pattern_regions == 1 && $pi =~ /^$/
188             || @pattern_regions == @v_regions;
189
190         if (xh_vecp $pi) {
191             my $sub_bind = destructuring_bind $pi, $vi;
192             return undef unless ref $sub_bind;
193             my %sub_bindings = %$sub_bind;
194             for (keys %sub_bindings) {
195                 return undef if exists $bindings{$_}
196                     && $bindings{$_} ne $sub_bindings{$_};
197                 $bindings{$_} = $sub_bindings{$_};
198             }
199         } elsif (xh_listp $pi) {
200             die "TODO: implement list binding for $pi";
201         } elsif ($pi =~ /^$\{?(w+)\}\}$/) {
202             return undef if exists $bindings{$1} && $bindings{$1} ne $vi;
203             $bindings{$1} = $vi;
204         } elsif ($pi =~ /^$/ ) {
205             die "illegal binding form: $pi";
206         } else {
207             return undef unless $pi eq $vi;
208         }
209     }
210

```

```

211     {%bindings};
212 }
213
214 sub invoke;
215 sub interpolate;
216 sub interpolate {
217     # Takes a string and a compiled binding hash and interpolates all
218     # applicable substrings outside-in. This process may involve full
219     # evaluation if () subexpressions are present, and is in general
220     # quadratic or worse in the length of the string.
221     my $bindings          = $_[0];
222     my @interpolation_regions = active_regions $_[1];
223     my @result_pieces;
224
225     for (@interpolation_regions) {
226         my $slice = substr $_[0], $_ >> 32, $_ & 0xffffffff;
227
228         # NOTE: no support for complex ${} expressions
229         if ($slice =~ /\^$(@?!?)\{?(\\w+)\}\\?$/ ) {
230             # Expand a named variable that may or may not be defined yet.
231             push @result_pieces,
232                 exists ${$bindings}{$2} ?
233                     $1 eq '' ? xh_listquote(xh_deactivate $bindings->{$2})
234                     : $1 eq '@' ? xh_deactivate($bindings->{$2})
235                     : $1 eq '!' ? xh_listquote($bindings->{$2})
236                     :
237                     $bindings->{$2}
238                 : "\\$$slice";
239         } elsif ($slice =~ /\^\\((.*)\\)$/s) {
240             push @result_pieces, invoke $bindings, parse_list interpolate $1;
241         } else {
242             push @result_pieces, $slice;
243         }
244     }
245
246     join '', @result_pieces;
247 }
248
249 sub xh_function_cases {
250     my @result;
251     my @so_far;
252     for (parse_vlist $_[0]) {
253         my ($command, @args) = parse_list $_;
254         if (xh_vecp $command) {
255             push @result, into_block @so_far if @so_far;
256             @so_far = ($command, into_list @args);
257         }
258     }
259 }

```

```

257     }
258     push @result, into_block @so_far if @so_far;
259     @result;
260 }
261
262 sub evaluate;
263 sub invoke {
264     # NOTE: no support for (foo bar $x)-style conditional destructuring;
265     # these are all rewritten into lambda forms
266     my ($bindings, $f, @args) = @_;
267     my $args = into_vec @args;
268
269     # Resolve f into a lambda form if it's still in word form.
270     $f = $bindings->{$f} if exists $bindings->{$f};
271
272     # Escape into perl
273     return $f->($bindings, @args) if ref $f eq 'CODE';
274
275     my %nested_bindings = %$bindings;
276     for (xh_function_cases $f) {
277         my ($formals, @body) = parse_block $_;
278         if (my $maybe_bindings = destructuring_bind $formals, $args) {
279             $nested_bindings{$_} = %$maybe_bindings{$_}
280             for keys %$maybe_bindings;
281             return evaluate {%nested_bindings}, into_block @body;
282         }
283     }
284
285     return into_list $f, @args;
286 }
287
288 sub evaluate {
289     my ($bindings, $block) = @_;
290     my @statements = parse_block $block;
291     my $result;
292
293     # NOTE: this function updates $bindings in place.
294     for (@statements) {
295         # Each statement is an invocation, which for now we assume all to be
296         # functions.
297         #
298         # NOTE: this is semantically incomplete as we don't consider
299         # macro-bindings.
300         $result = invoke $bindings, parse_list interpolate $bindings, $_;
301     }
302     $result;

```

303 }
304 _