

xh

Spencer Tipping

April 13, 2014

Contents

I	Language reference	2
1	Introduction	3
II	Self-hosting implementation	5
2	xh-script parser	6
III	Bootstrap implementation	8
3	Self-replication	9

Part I

Language reference

Chapter 1

Introduction

As a programming language, xh gives you two fairly uncommon invariants:

1. Every value is fully expressible as a string, and behaves as such.
2. Every computation can be expressed as a series of string-transformation rules.

xh's string transformations are all about expansion, which corresponds roughly to the kind of interpolation found in shell script or TCL. Unlike those languages, however, xh string interpolation itself has invariants, some of which you can disable. The semantics of xh are all defined in terms of the string representations of values, though xh is at liberty to use any representation that convincingly maintains the illusion that your values function as strings.

1.1 Examples

In these examples, \$ indicates the bash prompt and the outermost () indicate the xh prompt (neither needs to be typed).

bash	xh
\$ echo hi	(. hi)
\$ foo=bar	(def foo bar)
\$ echo \$foo	(. @foo)
\$ echo "\$foo"	(. \$foo)
\$ echo "\$(eval \$foo)"	(. !foo)
\$ echo \$(eval \$foo)	(. @!foo)
\$ find . -name '*.txt'	(find . -name '*.txt')
\$ ls name\ with\ spaces	(ls name\ with\ spaces)
\$ rm x && touch x	(rm x && touch x)
\$ ls wc -l	(ls wc -l)
\$ cat foo > bar	(cat foo > bar)

\$ for f in \$files; do	(map fn(rm \$_ && touch \$_) \$files)
> rm "\$f" && touch "\$f"	
> done	
\$ if [[-x foo]]; then	(if (-x foo) (./foo arg1 arg2 @_))
> ./foo arg1 arg2 "\$@"	
> fi	
\$ ls while read f; do	(ls filter -S)
> [[-S \$f]] && echo \$f	
> done	

Some xh features have no analog in bash, for instance data structures:

clojure	xh
(def m {})	(def m {})
(assoc m :foo 5)	{foo 5 @m}
(assoc m :foo 5)	(assoc \$m foo 5)
(dissoc m :foo :bar)	(dissoc \$m foo bar)
(:foo m)	(\$m foo)
(get m :foo 0)	(\$m foo 0)
(map? m)	(map? \$m)
(contains? m :foo)	(contains? \$m foo)
(def v [])	(def v [])
(conj v 1 2 3)	[@v 1 2 3]
(conj v 1 2 3)	(push \$v 1 2 3)
(def s #{})	(def s s[])
(contains? s :foo)	(\$s foo)
(contains? s :foo)	(contains? \$s foo)
(fn [x] (inc x))	fn(inc \$_)
(fn [x] (inc x))	(fn [\$x] (inc \$x))
(fn ([x] (inc x))	(fn [\$x] (inc \$x)
([x y] (+ x y)))	[\$x \$y] (+ \$x \$y))
(comp f g h)	(comp f g h)
(partial f x)	(partial f x)

Part II

Self-hosting implementation

Chapter 2

xh-script parser

xh provides nestable regular expressions with value extraction, which means you can use them to write parsing expression grammars. It isn't as powerful as Perl 6, which also supports assertions and backtracking annotations, but it does let you invert the regexp match to produce the original (or a modified) string again.

Listing 2.1 modules/parse.xh

```
1 # Bidirectional quote/unquote relation
2 (def xh-parser
3   (grammar rx" (?\${before} $xh-ignored )
4               (?\${v} $xh-vector | $xh-list | $xh-map | $xh-atom )
5               (?\${after} $xh-ignored ) "
6
7   xh-identifier rx" [^\s()\[\]\{\}\$@\"']+ "
8   xh-comment    rx" #(.*) "
9   xh-ignored    rx" ($xh-comment | \s*)* "
10  xh-atom       rx" $xh-hardstring | $xh-softstring
11                | $xh-bareword
12                | $xh-single-interpolation
13                | $xh-flat-interpolation "
14
15  xh-hardstring rx" (?\${prefix} $xh-identifier ?)
16                ' (?\${v} ([^'\\]* | \\\. )*) ' "
17
18  xh-softstring rx" (?\${prefix} $xh-identifier ?)
19                \" (?\${v} [^\\"\\\"$@]* | $xh-single-interpolation
20                  | $xh-flat-interpolation
21                  | $xh-interpolated-list
22                  | \\\. ) \" "
23
24  xh-bareword    rx" $xh-identifier "
```

```

25     xh-single-interpolation rx" \$ ($xh-identifier | $xh-list) "
26     xh-flat-interpolation  rx" \@ ($xh-identifier | $xh-list) "
27     xh-vector              (xh-braced '[' ''])
28     xh-list                (xh-braced '(' ''))
29     xh-map                  (xh-braced '{' '})
30     ^where
31     (xh-braced $open $close) rx" (?\$prefix $xh-identifier ?)
32                               $open (?\$xs $xh-parser *) $close ")"
33
34 # Evaluation semantics
35 (def (unquote (inv xh-parser $x)) (eval $x)
36   ^where
37   (eval xh-bareword[$x])    xh-string-val[$x]
38   (eval xh-hardstring[@xs]) xh-string-val[(join "" (map eval $xs))]
39   (eval xh-softstring[@xs]) xh-string-val[(join "" (map eval $xs))]
40
41   (eval xh-vector{prefix $p xs $xs})
42   xh-seq{prefix $p
43     type    vector
44     xs      (flatmap eval-single $xs)}
45
46   (eval xh-map{prefix $p xs $xs})
47   xh-seq{prefix $p
48     type    map
49     xs      (flatmap eval-single $xs)}
50
51   (eval xh-list{prefix fn xs $xs}) xh-fn[$xs]
52   # TODO
53   )

```


Part III

Bootstrap implementation

Chapter 3

Self-replication

Note: This implementation requires Perl 5.14 or later, but the self-compiled xh image will run on anything back to 5.10. For this and other reasons, mostly performance-related, you should always use the xh-compiled image rather than bootstrapping in production.

Listing 3.1 boot/xh-header

```
1  #!/usr/bin/env perl
2  BEGIN {eval(our $xh_bootstrap = q{
3  # xh: the X shell | https://github.com/spencertipping/xh
4  # Copyright (C) 2014, Spencer Tipping
5  # Licensed under the terms of the MIT source code license
6
7  # For the benefit of HTML viewers (long story):
8  # <body style='display:none'>
9  # <script src='http://spencertipping.com/xh/page.js'></script>
10 use 5.010;
11 package xh;
12 our %modules;
13 our @module_ordering;
14 our %eval_numbers = (1 => '$xh_bootstrap');
15
16 sub with_eval_rewriting(&) {
17     my @result = eval {$_[0]->(@_[1..$_#])};
18     $_ =~ s/\(eval (\d+)\)/$eval_numbers{$1}/eg if $_;
19     die $_ if $_;
20     @result;
21 }
22
23 sub named_eval {
24     my ($name, $code) = @_;
25     $eval_numbers{$1 + 1} = $name if eval('__FILE__') =~ /\(eval (\d+)\)/;
```

```

26   with_eval_rewriting {eval $code};
27 }
28
29 our %compilers = (pl => sub {
30   my $package = $_[0] =~ s/\./::/gr;
31   named_eval $_[0], "{package ::$package;\n$_[1]\n}";
32   die "error compiling module $_[0]: $_" if $_;
33 });
34
35 sub defmodule {
36   my ($name, $code, @args) = @_;
37   chomp($modules{$name} = $code);
38   push @module_ordering, $name;
39   my ($base, $extension) = split /\.(?w+$/), $name;
40   die "undefined module extension '$extension' for $name"
41     unless exists $compilers{$extension};
42   $compilers{$extension}->($base, $code, @args);
43 }
44
45 chomp($modules{bootstrap} = $::xh_bootstrap);
46 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 }}

```