xh

Spencer Tipping

April 19, 2014

Contents

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`

```
(defgrammar xh-parser
            rx" (?\$before $xh-ignored )
                (?\$v $xh-vector | $xh-list | $xh-map | $xh-atom )
                (?\$after $xh-ignored ) "

  xh-identifier rx" [^\s()\[\]{}$@\"']+ "
  xh-comment    rx" #(.*) "
  xh-ignored    rx" ($xh-comment | \s*)* "
  xh-atom       rx" $xh-hardstring | $xh-softstring
                  | $xh-bareword
                  | $xh-single-interpolation
                  | $xh-flat-interpolation "

  xh-hardstring rx" (?\$prefix $xh-identifier ?)
                    ' (?\$v ([^'\\]* | \\. )*) ' "

  xh-softstring rx" (?\$prefix $xh-identifier ?)
                    \" (?\$v [^\\\"\$\@]* | $xh-single-interpolation
                                          | $xh-flat-interpolation
                                          | $xh-interpolated-list
                                          | \\. ) \" "

  xh-bareword             rx" $xh-identifier "
  xh-single-interpolation rx" \$ ($xh-identifier | $xh-list) "
xh-flat-interpolation   rx" \@ ($xh-identifier | $xh-list) "
xh-vector               (xh-braced '[' ']')
xh-list                 (xh-braced '(' ')')
xh-map                  (xh-braced '{' '}')

(xh-braced $open $close) rx" (?\$prefix $xh-identifier ?)
                             $open (?\$xs $xh-parser *) $close ")
```

Chapter 3

Evaluation semantics

xh evaluation semantics can be defined in terms of parse trees. While we're at it, we also define the useful `unquote` (analogous to Clojure's `read-string`) and `quote` (analogous to Clojure's `pr-str`).

Listing 3.1 `modules/eval.xh`

```
(def (unquote $s) (grammar-apply $xh-parser $s)
     (quote   $s) (grammar-parse $xh-parser $s))
```

Chapter 4

Perl backend

As things stand, we have a perfectly good parser written in 2 and no way to execute it. This chapter doesn't really help; instead, we make the hole deeper and write an xh-to-Perl compiler in xh-script. There are good reasons to do it this way.

Listing 4.1 `modules/perl.xh`

```
# TODO
```

Part III

Bootstrap implementation

Chapter 5

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 5.1 boot/xh-header

```
#!/usr/bin/env perl
BEGIN {eval(our $xh_bootstrap = q{
# xh: the X shell | https://github.com/spencertipping/xh
# Copyright (C) 2014, Spencer Tipping
# Licensed under the terms of the MIT source code license

# For the benefit of HTML viewers (long story):
# <body style='display:none'>
# <script src='http://spencertipping.com/xh/page.js'></script>
use 5.010;
package xh;
our %modules;
our @module_ordering;
our %eval_numbers = (1 => '$xh_bootstrap');

sub with_eval_rewriting(&) {
  my @result = eval {$_[0]->(@_[1..$#_])};
  $@ =~ s/\(eval (\d+)\)/$eval_numbers{$1}/eg if $@;
  die $@ if $@;
  @result;
}

sub named_eval {
  my ($name, $code) = @_;
  $eval_numbers{$1 + 1} = $name if eval('__FILE__') =~ /\(eval (\d+)\)/;
  with_eval_rewriting {eval $code};
}

our %compilers = (pl => sub {
  my $package = $_[0] =~ s/\./::/gr;
  named_eval $_[0], "{package ::$package;\n$_[1]\n}";
  die "error compiling module $_[0]: $@" if $@;
});

sub defmodule {
  my ($name, $code, @args) = @_;
  chomp($modules{$name} = $code);
  push @module_ordering, $name;
  my ($base, $extension) = split /\.(\w+$)/, $name;
  die "undefined module extension '$extension' for $name"
    unless exists $compilers{$extension};
  $compilers{$extension}->($base, $code, @args);
}

chomp($modules{bootstrap} = $::xh_bootstrap);
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)

```
sub image {
  my @pieces = "#!/usr/bin/env perl";
  push @pieces, "BEGIN {eval(our \$xh_bootstrap = <<'_')}",
                $modules{bootstrap},
                '_';
  push @pieces, "BEGIN {xh::defmodule('$_', <<'_')}",
                $modules{$_},
                '_' for @module_ordering;
  push @pieces, "xh::main::main;\n__DATA__";
  join "\n", @pieces;
}
})}
```