xh

Spencer Tipping

March 23, 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 `[]` indicate the xh prompt (neither needs to be typed).

```
bash
$ echo hi
$ foo=bar
$ echo $foo
$ echo "$foo"
$ echo "$(eval $foo)"
$ echo $(eval $foo)
$ find . -name '*.txt'
$ ls name\ with\ spaces
$ rm x && touch x
$ for f in $files; do
>   rm "$f" && touch "$f"
```

```
xh
[ . hi ]
[ =d foo bar ]
[ . @foo ]
[ . $foo ]
[ . !foo ]
[ . @!foo ]
[ find . -name '*.txt' ]
[ ls name\ with\ spaces ]
[ rm x && touch x ]
[ =m f[rm $_ && touch $_] $files ]
```

```
> done
$ if [[ -x foo ]]; then          [ -x foo && ./foo arg1 arg2 @_ ]
>   ./foo arg1 arg2 "$@"
> fi
$ # this is a comment            [ # this is a comment ]
$ ls | wc -l                     [ ls | wc -l ]
$ ls | while read f; do          [ ls | =f -S ]
>   [[ -S $f ]] && echo $f
> done
```

xh also shares some design elements with Haskell:

```
haskell                          xh
> f x | x == 0    = 1            [ =d [f 0]  1
      | otherwise = x * f (x-1)        [f $n] [* $n [f.-1 $n]] ]

> nats = 1 : map (+ 1) nats      [ =d nats {1 @nats=m:+1} ]
> take 5 nats                    [ =i $nats 0+5 ]
> f x = y * 2 where y = x + 1    [ =d [f $x] [*2 $y %w y [+1 $x]] ]
> let y = 10 in y + 1            [ +1 $y %w y 10 ]
```

And with Prolog:[1]

```
prolog                           xh
:- f(a, b).                      [ =d [f a] b ]
f(a, X) :- g(b, X).              [ =d [f a] [g b] ]
?- f(a, X).                      [ =b $x [f a] ]
?- f(X, b).                      [ =b [f $x] b ]
?- f(X, Y).                      # no direct equivalent
```

1.2 Special characters

```
!         expand without quoting
@         expand without singularizing
#         quote
$         expand
%         invoke macro
[]        expand the result of a function call
=         not a special character, just the prefix for most xh builtins
""        string with interpolation (like in bash)
''        string without interpolation
{}        string with interpolation, used as a list or map
```

[1] There's a lot more in common than is evident here, but I'm not familiar enough with Prolog syntax to list better analogies.

Part II

Self-hosting implementation

Chapter 2

xh-script parser

Defined in terms of structural equivalence between quoted and unquoted forms by specifying the behavior of the quote function, written in xh as `=q`.

Listing 2.1 `modules/parse.xh`

```
[=d [=q [@xs]] "\[@[=m =q @xs]\]"
    [=q {@xs}] "{@[=m =q @xs]}"
    [=q ""]]
# TODO
```

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

```
#!/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 5.014;
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 3.2 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;
}
})}
```