X shell

Spencer Tipping

February 21, 2014

Contents

Part I

Bootstrap implementation

Chapter 1

Self-replication

Listing 1.1 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 %compilers = (pl => sub {
  my $package = $_[0] =~ s/\./::/gr;
  eval "{package ::$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 1.2 `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;
}
})}
```

Chapter 2

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 2.1 modules/v.pl

```
BEGIN {xh::defmodule('xh::v.pl', <<'_')}
sub parse_with_brackets {
  my ($regexp, $filler, $x) = @_;
  $regexp = qr/$regexp/;
  my @initial_split = split /$regexp/, $x;

  @initial_split = grep length, @initial_split if $regexp =~ /\(/;
  my $item;
  my @result;
  my $bracket_count = 0;

  for my $data (@initial_split) {
    $bracket_count += length($data =~ s/\\.|[^\[({]//gr);
    $bracket_count -= length($data =~ s/\\.|[^\])}]//gr);
    $item = length($item) ? "$item$filler$data" : $data;
    unless ($bracket_count) {
      push @result, $item;
      $item = '';
    }
  }

  push @result, $item if $item;
  @result;
}

```

```
sub parse_lines {parse_with_brackets '\v',                 "\n", @_}
sub parse_words {parse_with_brackets '\s',                 " ",  @_}
sub parse_path  {parse_with_brackets '(/[^\[\](){}\s/]*)', "",   @_}

sub quote_as_line {parse_lines(@_) > 1 ? "{$_[0]}" : $_[0]}
sub quote_as_word {parse_words(@_) > 1 ? "{$_[0]}" : $_[0]}
sub quote_as_path {parse_path(@_)  > 1 ? "{$_[0]}" : $_[0]}

sub to_hash {
  my %result;
  for my $line (parse_lines $_[0]) {
    my ($key, @value) = parse_words $line;
    $result{$key} = [@value];
  }
  \%result;
}
_
```

Chapter 3

Perl compiler

This is the dumbest thing we can possibly do to make xh runnable. The compiler here is designed to operate on each function you define, and it has no support for syntax macros beyond 'def'. It also generates terrible code.

Listing 3.1 modules/compile.pl

```
BEGIN {xh::defmodule('xh::compile.pl', <<'_')}
our %shorthands = ("'["  => 'xh::compile::line_index',
                   "'s{" => 'xh::compile::line_transform',
                   "'#"  => 'xh::compile::line_count',
                   '@['  => 'xh::compile::word_index',
                   '@s{' => 'xh::compile::word_transform',
                   '@#'  => 'xh::compile::word_count',
                   ':['  => 'xh::compile::path_index',
                   ':s{' => 'xh::compile::path_transform',
                   ':#'  => 'xh::compile::path_count',
                   '"['  => 'xh::compile::byte_index',
                   '"s{' => 'xh::compile::byte_transform',
                   '"#'  => 'xh::compile::byte_count');

sub compile_statement;
sub compile_expression;
sub expand_shorthands;
sub unquote_list;
sub to_perl_ident;
sub to_perl_string;

sub compile_function {
  join "\n", "(sub {",
             'local $_ = $_[0];',
             (map {compile_statement $_} xh::v::parse_lines(@_)),
             "})";
```

```
}

sub compile_statement {
  my ($fn, @args) = xh::v::parse_words $_[0];
  if ($fn =~ /^\[/) {
    join ' . ', compile_expression($fn),
                map compile_expression($_), @args;
  } elsif ($fn eq 'def') {
    my @result;
    for (my $i = 0; $i < @args; $i += 2) {
      # No first-class names here.
      my $safe_name = to_perl_ident      $args[$i];
      my $expr      = compile_expression $args[$i + 1];
      push @result, "my \$$safe_name = $expr;";
    }
    @result;
  } else {
    # Everything else is just a function call.
    my $fn_name       = compile_expression $fn;
    my @compiled_args = map compile_expression($_), @args;
    "\${$fn_name}->(" . join(', ', @compiled_args) . ");";
  }
}

sub compile_expression {
  # Two cases that require interpretation here. One is when the word begins
  # with a $, in which case we expand shorthands into real function calls.
  # The other is when the word is quoted, in which case we unquote it by a
  # layer.
  my ($word) = @_;
  return expand_shorthands substr($word, 1) if $word =~ /^\$/;
  return unquote_list      $word            if $word =~ /^[{(]/;
  return to_perl_string    $word;
}

sub expand_shorthands {
  my ($initial, @operators) = xh::v::parse_path @_;
  my $result;
  if ($initial =~ /^[^\[\]{}]/) {
    # A regular word, so start by referencing a Perl variable.
    $result = "(\$" . to_perl_ident($initial) . ")";
  } elsif ($initial =~ /^\{/) {
    # A quoted constant; unquote by a layer and use a Perl string.
    $result = "(" . to_perl_string(unquote_list($initial)) . ")";
  } elsif ($initial =~ /^\(/) {
    # Substituting in the value from a command.
    $result = "((sub {local \$_ = \$_[0];"
            . compile_statement($initial)
            . "})->())";
  } elsif ($initial =~ /^\[/) {
    # A quoted vector; leave as a vector, parse as words, and evaluate each
    # element (TODO)
    die "need to implement shorthand-from-vector case";
  } else {
    die "expand_shorthands: got $initial";
  }

  # Now compile shorthands into function calls.
  for (my $i = 0; $i < @operators; ++$i) {
    my $op        = $operators[$i];
    my $arg_count = 0;

    die "$op does not begin with a slash" unless $op =~ s/^\///;

    while ($operators[$i + $arg_count + 1] =~ /^([\[({])/) {
      ++$arg_count;
      $op .= $1;
    }
    die "undefined shorthand operator: $op" unless exists $shorthands{$op};
    $result = "$shorthands{$op}("
            . join(",", $result, map compile_expression($_),
                                    @operators[$i + 1 .. $i + $arg_count])
            . ")";
    $i += $arg_count;
  }
  $result;
}

sub unquote_list {
  my ($l) = @_;

  # Simple case: literal expansion of {}
  return to_perl_string substr($l, 1, -1) if $l =~ /^\{/;

  # Any other list is subject to in-place interpolation (TODO: fix this to
  # preserve whitespace).
  my @elements = xh::v::parse_words substr($l, 1, -2);
  my $compiled = 'join(" "'
               . join(', ', map compile_expression($_), @elements)
               . ')';

  $l =~ /^\[/ ? "'[' . $compiled . ']'" : $compiled;
}

sub to_perl_ident {
  # Mangle names by replacing every non-alpha character with its char-code.
  $_[0] =~ s/(\W)/"_x".ord($1)/egr;
}

sub to_perl_string {
  # Quote the value by escaping any single-quotes and backslashes.
  "'" . ($_[0] =~ s/[\\']/\\$1/gr) . "'";
}
_
```