X shell

Spencer Tipping

February 22, 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);
}
```

```
30
31 chomp($modules{bootstrap} = $::xh_bootstrap);
32 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_quoted {
  my ($events_to_split, $split_sublists, $s) = @_;
  my @result;
  my $current_item  = '';
  my $sublist_depth = 0;

  for my $piece (split /(\v+|\s+|\/|\\.|[\[\](){}])/, $s) {
    next unless length $piece;
    my $depth_before_piece = $sublist_depth;
    $sublist_depth += $piece =~ /^[\[({]$/;
    $sublist_depth -= $piece =~ /^[\])}]$/;

    if ($split_sublists && !$sublist_depth != !$depth_before_piece) {
      # Two possibilities. One is that we just closed an item, in which
      # case we take the piece, concatenate it to the item, and continue.
      # The other is that we just opened one, in which case we emit what we
      # have and start a new item with the piece.
      if ($sublist_depth) {
        # Just opened one; kick out current item and start a new one.
        push @result, $current_item if length $current_item;
        $current_item = $piece;
      } else {
        # Just closed a list; concat and kick out the full item.
        push @result, "$current_item$piece";
```

```
        $current_item = '';
      }
    } elsif (!$sublist_depth && $piece =~ /$events_to_split/) {
      # If the match produces a group, then treat it as a part of the next
      # item. Otherwise throw it away.
      push @result, $current_item if length $current_item;
      $current_item = $1;
    } else {
      $current_item .= $piece;
    }
  }

  push @result, $current_item if length $current_item;
  @result;
}

sub parse_lines {parse_with_quoted '\v+', 0, @_}
sub parse_words {parse_with_quoted '\s+', 0, @_}
sub parse_path  {parse_with_quoted '(/)', 1, @_}

sub brace_balance {my $without_escapes = $_[0] =~ s/\\.//gr;
                   length($without_escapes =~ s/[^\[({]//gr) -
                   length($without_escapes =~ s/[^\])}]//gr)}

sub escape_braces_in {$_[0] =~ s/([\\\[\](){}])/\\$1/gr}

sub brace_wrap {
  "{" . (brace_balance($_[0]) ? escape_braces_in($_[0]) : $_[0]) . "}"
}

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

sub split_by_interpolation {
  # Splits a value into constant and interpolated pieces, where
  # interpolated pieces always begin with $. Adjacent constant pieces may
  # be split across items. Any active backslash-escapes will be placed on
  # their own.

  my @result;
  my $current_item        = '';
  my $sublist_depth       = 0;
  my $blocker_count       = 0;      # number of open-braces
  my $interpolating       = 0;
  my $interpolating_depth = 0;
```

```
72
73    for my $piece (split /([\[\](){}]|\\.|\/[!@#]|\/|\$|\s+)/, $_[0]) {
74      $sublist_depth += $piece =~ /^[\[({]$/;
75      $sublist_depth -= $piece =~ /^[\])}]$/;
76      $blocker_count += $piece eq '{';
77      $blocker_count -= $piece eq '}';
78
79      if (!$interpolating) {
80        # Not yet interpolating, but see if we can find a reason to change
81        # that.
82        if (!$blocker_count && $piece eq '$') {
83          # Emit current item and start interpolating.
84          push @result, $current_item if length $current_item;
85          $current_item = $piece;
86          $interpolating = 1;
87          $interpolating_depth = $sublist_depth;
88        } elsif (!$blocker_count && $piece =~ /^\//) {
89          # The backslash should be interpreted, so emit it as its own piece.
90          push @result, $current_item if length $current_item;
91          push @result, $piece;
92          $current_item = '';
93        } else {
94          # Collect the piece and continue.
95          $current_item .= $piece;
96        }
97      } else {
98        # We're inside an interpolated quantity, so scan forwards collecting
99        # pieces until one of a few things happens:
100       #
101       # 1. We close the list in which the interpolation is happening.
102       # 2. We hit a / not immediately followed by an interpolation sigil.
103       # 3. We hit whitespace not inside a sublist.
104       #
105       # Cases (2) and (3) apply only if we're not inside a sublist.
106
107       if ($sublist_depth < $interpolating_depth
108           or $sublist_depth == $interpolating_depth
109              and $piece eq '/' || $piece =~ /^\s/) {
110         # No longer interpolating because of what we just saw, so emit
111         # current item and start a new constant piece.
112         push @result, $current_item if length $current_item;
113         $current_item = $piece;
114         $interpolating = 0;
115       } else {
116         # Still interpolating, so collect piece.
117         $current_item .= $piece;
```

```
      }
    }
  }

  push @result, $current_item if length $current_item;
  @result;
}

sub undo_backslash_escape {
  return "\n" if $_[0] eq '\n';
  return "\t" if $_[0] eq '\t';
  return "\\" if $_[0] eq '\\\\';
  substr $_[0], 1;
}
_
```