X shell

Spencer Tipping

February 22, 2014

Contents

Part I

Language reference

Chapter 1

Expansion syntax

```
xh$ echo $foo               # simple variable expansion
xh$ echo $(echo hi)         # command output expansion
xh$ echo $[$foo '0 @#]      # #words in first line of val of var foo
xh$ echo $[{foo bar} "#]    # number of bytes in quoted string 'foo bar'

xh$ echo $foo[0 1]          # reserved for future use (don't write this)
xh$ echo $foo$bar           # reserved for future use (use ${foo}$bar)

xh$ echo $foo               # quote result with braces
xh$ echo $'foo              # flatten into multiple lines (be careful!)
xh$ echo $@foo              # flatten into multiple words (one line)
xh$ echo $:foo              # multiple path components (one word)
xh$ echo $"foo              # multiple bytes (one path component)

xh$ echo ${foo}             # same as $foo
xh$ echo ${foo bar bif}     # reserved for future use

xh$ echo $@{asdf asdf}      # expands into asdf adsf

xh$ echo $$foo              # $ is right-associative
xh$ echo $^$foo             # expand $foo within calling context
xh$ echo $($'foo)           # result of running $'foo
xh$ $'foo                   # this works too
```

Part II

Bootstrap implementation

Chapter 2

Self-replication

Listing 2.1 boot/xh-header

```
#!/usr/bin/env perl
BEGIN {
print STDERR q{
NOTE: Development image

If you see this note after installing the shell, it's probably because
you're running a version that has not yet rebuilt itself (maybe you got the
wrong file from the Git repo?). You can do this, but it will be really
slow and may use a lot of memory. There are two ways to fix this:

1. Download the standard image from http://spencertipping.com/xh
2. Have this image recompile itself by running xh.recompile-in-place (this
   will take some time because it stress-tests your Perl runtime)

Note also that bootstrapping requires Perl 5.14 or later, whereas running a
compiled image just requires Perl 5.10.

};
}

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 2.2 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 3

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 3.1 modules/v.pl

```
BEGIN {xh::defmodule('xh::v.pl', <<'_')}
sub unbox;

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, unbox $current_item if length $current_item;
        $current_item = $piece;
      } else {
```

```
        # Just closed a list; concat and kick out the full item.
        push @result, unbox "$current_item$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, unbox $current_item if length $current_item;
      $current_item = $1;
    } else {
      $current_item .= $piece;
    }
  }

  push @result, unbox $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 quote_as_multiple_lines {
  return escape_braces_in $_[0] if brace_balance $_[0];
  $_[0];
}

sub brace_wrap {"{" . quote_as_multiple_lines($_[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 quote_default {brace_wrap $_[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.
```

```
72
73  my @result;
74  my $current_item        = '';
75  my $sublist_depth       = 0;
76  my $blocker_count       = 0;      # number of open-braces
77  my $interpolating       = 0;
78  my $interpolating_depth = 0;
79
80  my $closed_something    = 0;
81  my $opened_something    = 0;
82
83  for my $piece (split /([\[\](){}]|\\.|\/|\$|\s+)/, $_[0]) {
84    $sublist_depth += $opened_something = $piece =~ /^[\[({]$/;
85    $sublist_depth -= $closed_something = $piece =~ /^[\])}]$/;
86    $blocker_count += $piece eq '{';
87    $blocker_count -= $piece eq '}';
88
89    if (!$interpolating) {
90      # Not yet interpolating, but see if we can find a reason to change
91      # that.
92      if (!$blocker_count && $piece eq '$') {
93        # Emit current item and start interpolating.
94        push @result, $current_item if length $current_item;
95        $current_item = $piece;
96        $interpolating = 1;
97        $interpolating_depth = $sublist_depth;
98      } elsif (!$blocker_count && $piece =~ /^\\/) {
99        # The backslash should be interpreted, so emit it as its own piece.
100       push @result, $current_item if length $current_item;
101       push @result, $piece;
102       $current_item = '';
103     } else {
104       # Collect the piece and continue.
105       $current_item .= $piece;
106     }
107   } else {
108     # Grab everything until:
109     #
110     # 1. We close the list in which the interpolation occurred.
111     # 2. We close a list to get back out to the interpolation depth.
112     # 3. We observe whitespace.
113     # 4. We observe a path separator.
114
115     if ($sublist_depth < $interpolating_depth
116         or $sublist_depth == $interpolating_depth
117            and $closed_something || $piece eq '/' || $piece =~ /^\s/) {
```

```
        # No longer interpolating because of what we just saw, so emit
        # current item and start a new constant piece.
        push @result, $current_item if length $current_item;
        $current_item = $piece;
        $interpolating = 0;
      } else {
        # Still interpolating, so collect the piece.
        $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;
}

sub unbox {
  my ($s) = @_;
  my $depth  = 0;
  for my $piece (split /(\\.|[\[\](){}])/, $s) {
    $depth += $piece =~ /^[\[({]/;
    $depth -= $piece =~ /^[\])}]/;
    return $s if $depth <= 0;
  }
  substr $s, 1, -1;
}
_
```

Chapter 4

Evaluator

This bootstrap evaluator is totally cheesy, using Perl's stack and lots of recursion; beyond this, it is slow, allocates a lot of memory, and has absolutely no support for lazy values. Its only redeeming virtue is that it supports macroexpansion.

Listing 4.1 modules/e.pl

```
BEGIN {xh::defmodule('xh::e.pl', <<'_')}
sub evaluate;
sub interpolate;

sub interpolate_dollar {
  my ($binding_stack, $term) = @_;

  # First things first: strip off any prefix operator, then interpolate the
  # result. We do this because $ is right-associative.
  my ($prefix, $rhs) = $term =~ /^(\$\^*[@"':]?)?(.*)/g;

  # Do we have a compound form? If so, then we need to treat the whole
  # thing as a unit.
  if ($rhs =~ /^\(/) {
    # RHS is a command, so grab the result of executing the inside.
    return evaluate $binding_stack, substr($rhs, 1, -1);
  } elsif ($rhs =~ /^\[/) {
    # TODO: handle this case. Right now we count on the macro preprocessor
    # to do it for us.
    die 'unhandled interpolate case: $[]';
  } elsif ($rhs =~ /^\{/) {
    $rhs = xh::v::unbox $rhs;
  } else {
    # It's either a plain word or another $-term. Either way, go ahead and
    # interpolate it so that it's ready for this operator.
```

```
    $rhs = interpolate $binding_stack, $rhs;
  }

  # At this point we have a direct form we can use on the right: either a
  # quoted expression (in which case we unbox), or a word, in which case we
  # dereference.
  my $layer = length $rhs =~ /^\$(\^*)/ || 0;
  my $unquoted =
    $rhs =~ /^\{/ ? xh::v::unbox $rhs
                  : $$binding_stack[-($layer + 1)]{$rhs}    # local scope
                    // $$binding_stack[0]{$rhs}             # global scope
                    // die "unbound var: $rhs";

  # Now select how to quote the result based on the prefix.
  return xh::v::quote_as_multiple_lines $unquoted if $prefix eq "\$'";
  return xh::v::quote_as_line           $unquoted if $prefix eq "\$@";
  return xh::v::quote_as_word           $unquoted if $prefix eq "\$:";
  return xh::v::quote_as_path           $unquoted if $prefix eq "\$\"";
  xh::v::quote_default $unquoted;
}

sub interpolate {
  my ($binding_stack, $x) = @_;
  join '', map {$_ =~ /^\$/ ? interpolate_dollar $binding_stack, $_
               : $_ =~ /^\\/ ? xh::v::undo_backslash_escape $_
               : $_ } xh::split_by_interpolation $x;
}

sub call {
  my ($binding_stack, $fn, @args) = @_;
  push @$binding_stack,
       {_ => join ' ', map xh::v::quote_default($_), @args};
  my $result = evaluate $binding_stack, $fn;
  pop @$binding_stack;
  $result;
}

sub evaluate {
  my ($binding_stack, $body) = @_;
  my @statements = xh::v::parse_lines $body;
  my $result = '';

  for my $s (@statements) {
    my @words = xh::v::parse_words $s;

    # Step 1: Do we have a macro? If so, macroexpand before calling
```

```
    # anything. (NOTE: technically incorrect; macros should receive their
    # arguments with whitespace intact)
    @words = macroexpand $binding_stack, @words
    while is_a_macro $binding_stack, $words[0];

    # Step 2: Interpolate the whole command once.
    $s = interpolate $binding_stack,
                     join ' ', map xh::v::quote_default($_), @words;

    # Step 3: See if the interpolation produced multiple lines. If so, we
    # need to re-expand. Otherwise we can do a single function call.
    if (xh::v::parse_lines($s) > 1) {
      $result = evaluate $binding_stack, $s;
    } else {
      # Just one line, so continue normally. At this point we look up the
      # function and call it. If it's Perl native, then we're set; we just
      # call that on the newly-parsed arg list. Otherwise delegate to
      # create a new call frame and locals.
      my ($f, @args) = xh::v::parse_words $s;
      my $fn = $$binding_stack[-1]{$f}
            // $$binding_stack[0]{$f}
            // die "unbound function: $f";

      $result = ref $fn eq 'CODE' ? $fn->(@args)
                                  : call($binding_stack, $fn, @args);
    }
  }
  $result;
}
_
```