X shell

Spencer Tipping

February 23, 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 $'foo within current scope
xh$ $'foo                   # this works too
xh$ echo $^($'foo)          # result of running $'foo within calling scope
```

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, $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 split_lines {parse_with_quoted '\v+', 0, @_}
sub split_words {parse_with_quoted '\s+', 0, @_}
sub split_path  {parse_with_quoted '(/)', 1, @_}

sub parse_lines {map unbox($_), split_lines @_}
sub parse_words {map unbox($_), split_words @_}
sub parse_path  {map unbox($_), split_path  @_}

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 {
```

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

```
118
119       if ($sublist_depth < $interpolating_depth
120           or $sublist_depth == $interpolating_depth
121              and $piece eq '/' || $piece =~ /^\s/) {
122         # No longer interpolating because of what we just saw, so emit
123         # current item and start a new constant piece.
124         push @result, $current_item if length $current_item;
125         $current_item  = $piece;
126         $interpolating = 0;
127       } elsif ($sublist_depth == $interpolating_depth
128                && $closed_something) {
129         push @result, "$current_item$piece";
130         $current_item  = '';
131         $interpolating = 0;
132       } else {
133         # Still interpolating, so collect the piece.
134         $current_item .= $piece;
135       }
136     }
137   }
138
139   push @result, $current_item if length $current_item;
140   @result;
141 }
142
143 sub undo_backslash_escape {
144   return "\n" if $_[0] eq '\n';
145   return "\t" if $_[0] eq '\t';
146   return "\\" if $_[0] eq '\\\\';
147   substr $_[0], 1;
148 }
149
150 sub unbox {
151   my ($s) = @_;
152   my $depth      = 0;
153   my $last_depth = 1;
154   for my $piece (grep length, split /(\\.|[\[\](){}])/, $s) {
155     $depth += $piece =~ /^[\[({]/;
156     $depth -= $piece =~ /^[\])}]/;
157     return $s if $last_depth <= 0;
158     $last_depth = $depth;
159   }
160   $s =~ s/^\s*[\[({](.*)[\])}]\s*$/$1/sgr;
161 }
162 _
```

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 call;

sub interpolate_wrap {
  my ($prefix, $unquoted) = @_;
  return xh::v::quote_as_multiple_lines $unquoted if $prefix =~ /'$/;
  return xh::v::quote_as_line           $unquoted if $prefix =~ /\@$/;
  return xh::v::quote_as_word           $unquoted if $prefix =~ /:$/;
  return xh::v::quote_as_path           $unquoted if $prefix =~ /"$/;
  xh::v::quote_default $unquoted;
}

sub scope_index_for {
  my ($carets) = $_[0] =~ /^\$(\^*)/g;
  -(1 + length $carets);
}

sub truncated_stack {
  my ($stack, $index) = @_;
  return $stack if $index == -1;
  [@$stack[0 .. @$stack + $index]];
}

```

```
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) = $term =~ /^(\$\^*[@"':]?)(.*)$/sg;

  # Do we have a compound form? If so, then we need to treat the whole
  # thing as a unit.
  if ($rhs =~ /^\(/) {
    # The exact semantics here are a little subtle. Because the RHS is just
    # ()-boxed, it should be expanded within the current scope. The actual
    # evaluation, however, might be happening within a parent scope; we'll
    # know by looking at the $prefix to check for ^s.

    my $interpolated_rhs = interpolate $binding_stack, xh::v::unbox $rhs;
    my $index            = scope_index_for $prefix;
    my $new_stack        = truncated_stack $binding_stack, $index;

    return interpolate_wrap $prefix,
                            evaluate $new_stack, $interpolated_rhs;
  } elsif ($rhs =~ /^\[/) {
    # $[] is a way to call a series of functions on a value, just like
    # Clojure's (-> x y z). Like $(), we always interpolate the terms of
    # the [] list in the current environment; but any ^s you use (e.g.
    # $^[]) cause the inner functions to be called from a parent scope.
    # This can be relevant in certain pathological cases that you should
    # probably never use.

    my ($initial, @fns) = map {interpolate $binding_stack, $_}
                               xh::v::parse_words xh::v::unbox $rhs;
    my $index           = scope_index_for $prefix;
    my $calling_stack   = truncated_stack $binding_stack, $index;

    # You can use paths as a curried notation within $[] interpolation. For
    # example:
    #
    # > echo $[foo echo/hi]
    # hi foo
    #
    # Lists also work, but there is no difference between () and [], which
    # is a horrible oversight that should probably be addressed at some
    # point.
    $initial = call $calling_stack,
                    (map {s/^\///r} xh::v::parse_path($_)),
                    xh::v::parse_words $initial
```

```
    for @fns;

    return interpolate_wrap $prefix, $initial;
  } elsif ($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 = xh::v::unbox interpolate $binding_stack, $rhs;
  }

  # Try to unwrap any layers around the RHS. Any braces at this point mean
  # that it's artificially quoted, or that the RHS is unusable.
  while ($rhs =~ /^\{/) {
    my $new_rhs = xh::v::unbox $rhs;
    die "illegal interpolation: $rhs" if $new_rhs eq $rhs;
    $rhs = $new_rhs;
  }

  my $index = scope_index_for $prefix;
  interpolate_wrap $prefix,
    $$binding_stack[$index]{$rhs}
    // $$binding_stack[0]{$rhs}
    // die "unbound var: $rhs (bound vars are ["
           . join(' ', sort keys %{$$binding_stack[$index]})
           . "] locally, ["
           . join(' ', sort keys %{$$binding_stack[0]})
           . "] globally)";
}

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

sub call {
  my ($binding_stack, $f, @args) = @_;
  my $fn = $$binding_stack[-1]{$f}
        // $$binding_stack[0]{$f}
        // die "unbound function: $f";

  # Special case: if it's a builtin Perl sub, then just call that directly.
  return &$fn($binding_stack, @args) if ref $fn eq 'CODE';

```

```
  # Otherwise use xh calling convention.
  push @$binding_stack,
       {_ => join ' ', map xh::v::quote_default($_), @args};

  my $result = eval {evaluate $binding_stack, $fn};
  my $error  = "$@ in $f "
             . join(' ', map xh::v::quote_default($_), @args)
             . ' at calling stack depth ' . @$binding_stack
             . " with locals:\n"
             . join("\n", map "  $_ -> $$binding_stack[-1]{$_}",
                              sort keys %{$$binding_stack[-1]}) if $@;
  pop @$binding_stack;
  die $error if $error;
  $result;
}

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

  for my $s (@statements) {
    my $original = $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)
    my @words;
    while ((@words = xh::v::parse_words $s)[0] =~ /^#/) {
      $s = eval {call $binding_stack, @words};
      die "$@ in @words (while macroexpanding $original)" if $@;
    }

    # Step 2: Interpolate the whole command once. Note that we can't wrap
    # each word at this point, since that would block interpolation
    # altogether.
    my $new_s = eval {interpolate $binding_stack, $s};
    die "$@ in $s (while interpolating from $original)" if $@;
    $s = $new_s;

    # If that killed our value, then we have nothing to do.
    next unless length $s;

    # 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.
      $result = eval {call $binding_stack, xh::v::parse_words $s};
      die "$@ in $s (while evaluating $original)" if $@;
    }
  }
  $result;
}
_
```

Chapter 5

Globals

At this point we have the evaluator logic, but xh code can't do anything because it has no way to create variable bindings. This is solved by defining the `def` function and list/hash accessors.

Listing 5.1 modules/globals.pl

```
BEGIN {xh::defmodule('xh::globals.pl', <<'_')}
sub def {
  my ($binding_stack, %args) = @_;
  $$binding_stack[-1]{$_} = $args{$_} for keys %args;
  join ' ', keys %args;
}

sub echo {
  my ($binding_stack, @args) = @_;
  join ' ', @args;
}

sub comment       {''}
sub print_from_xh {print STDERR join(' ', @_[1 .. $#_]), "\n"}

sub perl_eval {
  my $result = eval $_[1];
  die "$@ while evaluating $_[1]" if $@;
  $result;
}

sub assert_eq_macro {
  my ($binding_stack, $lhs, $rhs) = @_;

  # We should get the same result by evaluating the LHS and RHS; otherwise
  # expand into a print statement describing the error.
```

```
  my $expanded_lhs = xh::e::interpolate $binding_stack, $lhs;
  my $expanded_rhs = xh::e::interpolate $binding_stack, $rhs;

  $expanded_lhs eq $expanded_rhs
    ? ''
    : 'print ' . xh::v::quote_default("$lhs (-> $expanded_lhs)")
               . ' != '
               . xh::v::quote_default("$rhs (-> $expanded_rhs)");
}

# Create an interpreter instance that lets us interpret modules written in
# XH-script.
our $globals = [{def   => \&def,
                 echo  => \&echo,
                 print => \&print_from_xh,
                 perl  => \&perl_eval,
                 '#'   => \&comment,
                 '#==' => \&assert_eq_macro}];

sub defglobals {
  my %vals = @_;
  $$globals[0]{$_} = $vals{$_} for keys %vals;
}

$xh::compilers{xh} = sub {
  my ($module_name, $code) = @_;
  eval {xh::e::evaluate $globals, $code};
  die "error running $module_name: $@" if $@;
}
_
```

5.1 List accessors

List elements are accessed using single-character functions, one for each type of list.

Listing 5.2 modules/bootlist.pl

```
BEGIN {xh::defmodule('xh::bootlist.pl', <<'_')}
sub wrap_negative {
  my ($i, $n) = @_;
  return undef unless length $i;
  return $n + $i if $i < 0;
  $i;
}

```

```
sub flexible_range {
  my ($lower, $upper) = @_;
  return reverse $upper .. $lower if $upper < $lower;
  $lower .. $upper;
}

sub expand_subscript;
sub expand_subscript {
  my ($subscript, $n) = @_;

  return [map expand_subscript($_, $n),
              xh::v::split_words xh::v::unbox $subscript]
  if $subscript =~ /^\{/;

  return [flexible_range wrap_negative($1, $n) // 0,
                         wrap_negative($2, $n) // $n - 1]
  if $subscript =~ /^(-?\d*):(-?\d*)$/;

  return wrap_negative $subscript, $n if $subscript =~ /^-/;
  $subscript;
}

sub dereference_one;
sub dereference_one {
  my ($subscript, $boxed_list) = @_;

  # List homomorphism of subscripts
  return xh::v::quote_default
         join ' ', map dereference_one($_, $boxed_list),
                       @$subscript if ref $subscript eq 'ARRAY';

  # Normal numeric lookup, with empty string for out-of-bounds
  return ''                               if $subscript =~ /^-/;
  return $$boxed_list[$subscript] // '' if $subscript =~ /^\d+/;

  if ($subscript =~ s/^\^//) {
    # In this case the boxed list should contain at least words, and
    # probably whole lines. We word-parse each entry looking for the
    # first subscript hit.
    $subscript = xh::v::unbox $subscript;
    for my $x (@$boxed_list) {
      my @words = xh::v::parse_words $x;
      return xh::v::quote_as_word $x if $words[0] eq $subscript;
    }
    '';
  } else {
```

```
    die "unrecognized subscript form: $subscript";
  }
}

sub dereference;
sub dereference {
  my ($subscript, $boxed_list) = @_;
  $subscript = xh::v::quote_as_word $subscript;
  dereference_one expand_subscript($subscript, scalar(@$boxed_list)),
                  $boxed_list;
}

sub index_lines {dereference $_[1], [xh::v::parse_lines $_[2]]}
sub index_words {dereference $_[1], [xh::v::parse_words $_[2]]}
sub index_path  {dereference $_[1], [xh::v::parse_path  $_[2]]}
sub index_bytes {dereference $_[1], [map ord, split //, $_[2]]}

xh::globals::defglobals "'"  => \&index_lines,
                        "@"  => \&index_words,
                        ":"  => \&index_path,
                        "\"" => \&index_bytes;
_
```

Chapter 6

Bootstrap unit tests

This is our first layer of sanity checking for the interpreter. A failure here won't stop xh from running, but it will print a diagnostic message so we know something is up.

Listing 6.1 modules/bootunit.xh

```
# This is a comment and should work properly.
# {
  This is a block comment and should also work.
}
#== 1 1

def foo bar
#== $@foo         bar
#== $@foo         {bar}
#== $@foo         (bar)
#== $@foo         [bar]
#== $foo          {{bar}}
#== $(echo $foo)  {{bar}}
#== $@(echo $foo) bar

def greet {
  echo hi there, $@_
}
#== $@(greet spencer)         {hi there, spencer}
#== $@(greet spencer tipping) {hi there, spencer tipping}

def newdef {
  # Define stuff within the calling scope; should be equivalent to using
  # def.
  echo $^(def $@_)
}
```

```
newdef x 5
#== $@x 5

def two-statements {
  def x 10
  echo $x
}
#== $@x 5
$'two-statements
#== $@x 10

#== $@[there echo/hi]              {hi there}
#== $@[spencer echo/there echo/hi] {hi there spencer}

def xs (foo bar bif baz)
#== $@(@ 0 $xs) foo
#== $@(@ 1 $xs) bar
#== $@(@ 2 $xs) bif
#== $@(@ 3 $xs) baz
#== $@(@ ^foo $xs) foo

#== $@[$xs @/0]   foo
#== $@[$xs @/-1]  baz
#== $@[$xs @/-2]  bif
#== $@[$xs @/:]   {{foo bar bif baz}}
#== $@[$xs @/1:]  {{bar bif baz}}
#== $@[$xs @/:1]  {{foo bar}}
#== $@[$xs @/:-2] {{foo bar bif}}
#== $@[$xs @/3:1] {{baz bif bar}}

#== $@[$xs @/^bar] bar
#== $@[$xs @/^bif] bif
#== $@[$xs @/^notfound] {}

#== $@[$xs @{0 2}]    {{foo bif}}
#== $@[$xs @{0 2:}]   {{foo {bif baz}}}
#== $@[$xs @{0 {2:}}] {{foo {{bif baz}}}}

def associative {
  foo bar
  bif baz
}
#== $@[$associative '/^foo] {{  foo bar}}
#== $@[$associative '/^foo @/1] bar
#== $@[$associative '/^bif @/1] baz
#== $@[$associative '/^bok] {}
```

```
73
74 #== $@[abcd "/0] 97
75 #== $@[abcd "/1:3] {{98 99 100}}
76
77 #== $@[/usr/bin/bash :(^/bin)] /bin
78 #== $@[../.. :/^..] ..
79
80 def #-> {echo #== \$@($@[$_ @/0]) $[$_ @/1]}
81 #-> {echo hi} hi
```

Chapter 7

REPL

A totally cheesy bootstrap repl for now. Later on this will be implemented in xh-script.

Listing 7.1 modules/main.pl

```
BEGIN {xh::defmodule('xh::main.pl', <<'_')}
sub main {
  # This keeps xh from blocking on stdin when we ask it to compile itself.
  /^--recompile$/ and return 0 for @ARGV;

  my $list_depth    = 0;
  my $expression    = '';
  my $binding_stack = $xh::globals::globals;

  print "xh\$ ";
  while (my $line = <STDIN>) {
    if (!($list_depth += xh::v::brace_balance $line)) {
      # Collect the line and evaluate everything we have.
      $expression .= $line;

      my $result = eval {xh::e::evaluate $binding_stack, "$expression"};
      print "error: $@\n" if length $@;
      print "$result\n"   if length $result;

      $expression = '';
      print "xh\$ ";
    } else {
      $expression .= $line;
      print '>   ' . '  ' x $list_depth;
    }
  }
}
```

28 _