X shell

Spencer Tipping

February 25, 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 $foo                # quote result into a word
xh$ echo $'foo               # flatten into multiple lines (be careful!)
xh$ echo $@foo               # flatten into multiple words (one line)
xh$ echo $:foo               # one path component
xh$ echo $"foo               # one braced list

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', <<'_')}
use Memoize qw/memoize/;

sub unbox;

sub parse_with_quoted {
  my ($events_to_split, $split_sublists, $take_zero_width, $s) = @_;
  my @result;
  my $current_item  = '';
  my $sublist_depth = 0;

  for my $piece (split /(\v|\s+|\/|\\.|[\[\]{}()])/, $s) {
    next if !$take_zero_width and !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 $take_zero_width or
```

```
                                       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 $take_zero_width or
                                     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, 1, @_}
sub split_words {parse_with_quoted '\s+', 0, 0, @_}
sub split_path  {parse_with_quoted '(/)', 1, 0, @_}

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

memoize $_ for qw/parse_lines parse_words parse_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];
}

memoize 'quote_as_multiple_lines';

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.

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

  my $closed_something    = 0;
  my $opened_something    = 0;

  for my $piece (split /([\[\]()\{}]|\\.|\/|\$|\s+)/, $_[0]) {
    $sublist_depth += $opened_something = $piece =~ /^[\[({]$/;
    $sublist_depth -= $closed_something = $piece =~ /^[\])}]$/;
    $blocker_count += $piece eq '{';
    $blocker_count -= $piece eq '}';

    if (!$interpolating) {
      # Not yet interpolating, but see if we can find a reason to change
      # that.
      if (!$blocker_count && $piece eq '$') {
        # Emit current item and start interpolating.
        push @result, $current_item if length $current_item;
        $current_item = $piece;
        $interpolating = 1;
        $interpolating_depth = $sublist_depth;
      } elsif (!$blocker_count && $piece =~ /^\\/) {
        # The backslash should be interpreted, so emit it as its own piece.
        push @result, $current_item if length $current_item;
        push @result, $piece;
        $current_item = '';
      } else {
        # Collect the piece and continue.
        $current_item .= $piece;
      }
    } else {
      # Grab everything until:
      #
      # 1. We close the list in which the interpolation occurred.
      # 2. We close a list to get back out to the interpolation depth.
      # 3. We observe whitespace.
      # 4. We observe a path separator.
      # 5. We hit a backslash.

      if ($sublist_depth < $interpolating_depth
          or $sublist_depth == $interpolating_depth
             and $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;
        $interpolating = 0;
      } elsif ($sublist_depth == $interpolating_depth
               && $closed_something) {
        push @result, "$current_item$piece";
        $current_item  = '';
        $interpolating = 0;
      } elsif ($sublist_depth == $interpolating_depth && $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;
  my $last_depth = 1;
  for my $piece (grep length, split /(\\.|[\[\](){}])/, $s) {
    $depth += $piece =~ /^[\[({]/;
    $depth -= $piece =~ /^[\])}]/;
    return $s if $last_depth <= 0;
    $last_depth = $depth;
  }
  $s =~ s/^\s*[\[({](.*)[\])}]\s*$/$1/sgr;
}
_
```

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_path           $unquoted if $prefix =~ /:$/;
  return xh::v::quote_default           $unquoted if $prefix =~ /"$/;
  xh::v::quote_as_word $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 =~ /^\{/) {
    # Interpolated quotation, possibly under a different scope index.
    my $index         = scope_index_for $prefix;
    my $calling_stack = truncated_stack($binding_stack, $index);

    return interpolate_wrap $prefix,
      interpolate $calling_stack, 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;

    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[$index - 1]})
             . " ] in parent stack, ["
             . 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 $fn = xh::v::quote_as_word($f) =~ /^\{/ ? $f
         : $$binding_stack[-1]{$f}
        // $$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 @$binding_stack,
       {_ => join ' ', map xh::v::quote_as_word($_), @args};

  my $result = eval {evaluate $binding_stack, $fn};
  my $error  = "$@ in $f "
             . join(' ', map xh::v::quote_as_word($_), @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 @words = xh::v::parse_words $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 $binding_stack, @words};
      die "$@ 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, $n, %args) = @_;
  $$binding_stack[-$n]{$_} = $args{$_} for keys %args;
  join ' ', keys %args;
}

sub local_def {def $_[0], 1, @_[1..$#_]}

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) = @_;

```

```perl
  # 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)");
}

sub xh_if {
  my ($binding_stack, $cond, $then, $else) = @_;
  xh::e::evaluate $binding_stack, length $cond ? $then : $else;
}

sub xh_while {
  my ($binding_stack, $cond, $body) = @_;
  my $result;
  $result = xh::e::evaluate $binding_stack, $body
    while length xh::e::evaluate $binding_stack, $cond;
  $result;
}

sub xh_not {
  my ($binding_stack, $v) = @_;
  length $v ? '' : '{}';
}

sub xh_eq {
  my ($binding_stack, $x, $y) = @_;
  $x eq $y ? "{" . xh::v::quote_as_word($x) . "}" : '';
}

sub xh_matches {
  # NOTE: leaky abstraction (real xh regexps won't support all of the perl
  # extensions)
  my ($binding_stack, $pattern, $s) = @_;
  $s =~ /$pattern/ ? "{" . xh::v::quote_as_word($s) . "}" : '';
}

sub escalate {
  my ($binding_stack, $levels, $body) = @_;
  xh::e::evaluate xh::e::truncated_stack($binding_stack, -($levels + 1)),
                  $body;
```

```
}

# Create an interpreter instance that lets us interpret modules written in
# XH-script.
our $globals = [{def    => \&local_def,
                 '^def' => \&def,
                 '^'    => \&escalate,
                 echo   => \&echo,
                 print  => \&print_from_xh,
                 perl   => \&perl_eval,
                 if     => \&xh_if,
                 while  => \&xh_while,
                 not    => \&xh_not,
                 '=='   => \&xh_eq,
                 '=~'   => \&xh_matches,
                 '#'    => \&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 xh::v::quote_as_word '' if $subscript =~ /^-/;
  return $$boxed_list[$1] // ''  if $subscript =~ /^(\d+)!$/;

  return xh::v::quote_as_word $$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 $subscript;
    }
    '';
  } elsif ($subscript eq '#') {
    scalar @$boxed_list;
  } 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]]}

sub outer_lines {dereference $_[1], [xh::v::split_lines $_[2]]}
sub outer_words {dereference $_[1], [xh::v::split_words $_[2]]}
sub outer_path  {dereference $_[1], [xh::v::split_path  $_[2]]}

sub update {
  my ($subscript, $replacement, $join, $quote, $boxed_list) = @_;
  my $expanded = expand_subscript $subscript, scalar @$boxed_list;

  die "can't use list subscript for update: $subscript"
  if ref $expanded eq 'ARRAY';

  my $associative = $expanded =~ s/^\^//;

  my @result;
  for (my $i = 0; $i < @$boxed_list; ++$i) {
    my ($k) = xh::v::parse_words $$boxed_list[$i];
    push @result, ($associative ? $expanded eq $k : $expanded eq $i)
                  ? $replacement
                  : $$boxed_list[$i];
  }

  if ($expanded =~ /^\d+$/ and $expanded > @$boxed_list) {
    # It could be that we need to add something to the end.
    for (my $i = @$boxed_list; $i < $expanded; ++$i) {
      push @result, '';
    }
    push @result, $replacement;
  }

  xh::v::quote_as_word join $join, map &$quote($_), @result;
}

sub update_lines {update @_[1, 2], "\n", \&xh::v::quote_as_line,
                         [xh::v::parse_lines $_[3]]}

sub update_words {update @_[1, 2], ' ',  \&xh::v::quote_as_word,
                         [xh::v::parse_words $_[3]]}

sub update_path  {update @_[1, 2], '',   \&xh::v::quote_as_path,
                         [xh::v::parse_path  $_[3]]}

sub update_byte  {update @_[1, 2], '',   sub {$_[0]},
                         [map ord, split //, $_[3]]}

xh::globals::defglobals "'"  => \&index_lines,  "'="  => \&update_lines,
                        "@"  => \&index_words,  "@="  => \&update_words,
                        ":"  => \&index_path,   ":="  => \&update_path,
                        "\"" => \&index_bytes,  "\"=" => \&update_byte,

                        "'%" => \&outer_lines,
                        "@%" => \&outer_words,
                        ":%" => \&outer_path;

# Conversions between list types.
sub list_to_list_fn {
  my ($join, $quote, $parse) = @_;
  sub {xh::v::quote_as_word
       join $join, map &$quote($_), map &$parse($_), @_[1 .. $#_]};
}

my %joins   = ("'" => "\n", "@" => ' ', ":" => '/', "\"" => '');
my %quotes  = ("'"  => \&xh::v::quote_as_line,
               "@"  => \&xh::v::quote_as_word,
               ":"  => \&xh::v::quote_as_path,
               "\"" => sub {chr $_[0]});

my %parsers = ("'"  => \&xh::v::parse_lines,
               "@"  => \&xh::v::parse_words,
               ":"  => \&xh::v::parse_path,
               "\"" => sub {map ord, split //, $_[0]});

```

```
for my $k1 (keys %parsers) {
  for my $k2 (keys %parsers) {
    next if $k1 eq $k2;
    my $fn = list_to_list_fn($joins{$k2}, $quotes{$k2}, $parsers{$k1});
    xh::globals::defglobals "$k1$k2" => $fn;
  }
}

sub explode {xh::v::unbox $_[1]}
xh::globals::defglobals '!' => \&explode;
_
```

5.2 Double-precision math

These functions are low-level and are usually called by generated code rather than by hand. See 10 for a macro that does this.

Listing 5.3 modules/bootmath.pl

```
BEGIN {xh::defmodule('xh::math.pl', <<'_')}
sub binary_to_nary {
  my ($f, $zero) = @_;
  sub {
    my ($bindings, $x, @args) = @_;
    return $zero unless defined $x;
    return &$f($zero, $x) unless @args;
    $x = &$f($x, $_) for @args;
    $x;
  };
}

xh::globals::defglobals
  "math$_" => binary_to_nary(eval "sub {\$_[0] $_ \$_[1]}", /^[*\/]$/)
for qw[+ - * / & | ! < > << >>];
_
```

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

```
def test {
  def perltime {perl {use Time::HiRes qw/time/; time}}
  def start-time $(perltime)
  $'[$_ @/1!]
  def end-time $(perltime)
  print tested $[$_ @/0] in $(math* 1000 $(math- $end-time $start-time)) ms
}

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

  test basic-interpolation {
    def foo bar
    #== $@foo         bar
    #== $@foo         {bar}
    #== $@foo         (bar)
    #== $@foo         [bar]
    #== $foo          bar
    #== $(echo $foo)  bar
    #== $@(echo $foo) bar
  }

```

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

  # Also anonymous functions:
  #== $@($greet spencer)       {hi there, spencer}
  #== $@({echo hi $_} spencer) {hi spencer}
}

test scoping {
  def newdef {
    # Define stuff within the calling scope; should be equivalent to
    # using def.
    echo $^(def $@_)
  }
  newdef x 5
  #== $@x 5
}

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

test outer-interpolation {
  def get-5-plus {
    echo $(math+ $[$_ @/0] 5)
  }
  def inner {
    echo $^(get-5-plus 10)
  }
  #== $(inner) 15
}

test list-accessors {
  def xs (foo bar bif baz)
  #== $@(@ 0 $xs) foo
```

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

    def ys ({foo} {bar bif} [baz] (bok))
    #== $@(@% 0 $ys) {{foo}}
    #== $@(@  0 $ys) foo
    #== $@(@% 1 $ys) {{bar bif}}
    #== $@(@  1 $ys) {{bar bif}}
    #== $@(@% 2 $ys) {[baz]}
    #== $@(@  2 $ys) baz
    #== $@(@% 3 $ys) {(bok)}
    #== $@(@  3 $ys) bok

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

      #== $@[$^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}}}}
    }
  }

  test list-updaters {
    def xs (a b c d)
    #== $@[$xs @=/0/b  !] {b b c d}
    #== $@[$xs @=/-1/a !] {a b c a}
  }

  test associative-maps {
    def associative {
```

```
    foo bar
    bif baz
  }
  #== $@[$associative '/^foo @(0 1)] {{foo bar}}
  #== $@[$associative '/^foo @/1] bar
  #== $@[$associative '/^bif @/1] baz
  #== $@[$associative '/^bok] {}

  #== $@[$associative '/#] 4
  #== $@[$associative @/#] 4

  #== $@[$associative '=/^foo[FOO BAR] '/^FOO @/1] BAR
}

test byte-lists {
  #== $@[abcd "/0] 97
  #== $@[abcd "/1:3] {{98 99 100}}
}

test path-lists {
  #== $@[/usr/bin/bash :(^/bin)] /bin
  #== $@[../.. :/^..] ..
}

test macro-definition {
  def #-> {echo #== \$@($@[$_ @/0]) $[$_ @/1]}
  #-> {echo hi} hi
}

test equality-comparison {
  def x 10
  #== $(== $x 10) {{10}}
  #== $(== {} {}) {{}}
  #== $(== $x 9)  {}

  #== $(not $(== $x 9))  {{}}
  #== $(not $(== $x 10)) {}
}

test conditions {
  def x 5
  if $(== $x 5) {def x 6} {def x 7}
  #== $x 6
  if $(== $x 5) {def x 8} {def x 9}
  #== $x 9
}
```

```

  test iteration {
    def i     0
    def count 0
    while {not $(== $i 10)} {
      def i     $(math+ $i     1)
      def count $(math+ $count 1)
    }
    #== $count 10
    #== $i     10
  }

  test float-math {
    #== $(math+ 0 1) 1
    #== $(math/ 4 8) 0.5
    #== $(math< 3 4) 1
    #== $(math< 5 4) {}
  }
}
```

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 _

Part III

xh standard library

Chapter 8

Function functions

Listing 8.1 modules/fn.xh

```
def defn {
  def fname $[$_ @/0]
  def args  $[$_ @/1]
  def body  $[$_ @/2]

  def i 0
  def argdefs {}
  while {math< $i $[$args @/#]} {
    def argdefs [$'argdefs \n def $[$args @/$i] \$[\$_ @/$i]]
    def i $(math+ $i 1)
  }

  ^def 2 $fname ${$'argdefs \n $'body}
}
```

Chapter 9

List functions

The usual suspects.

Listing 9.1 modules/list.xh

```
defn @each [v f xs] {
  def i 0
  def n $[$xs @/#]
  while {math< $i $n} {
    ^ 1 ${def $v $[$xs @/$i] \n $'f}
    def i $(math+ $i 1)
  }
}

defn @m [f xs] {
  def ys {}
  @each x {def ys ($@ys $(f $@x))} $xs
  echo $ys
}

#== $(@m {math+ 1 $_} [1 2 3]) {{2 3 4}}
```

Chapter 10

Math macro

The math macro converts infix math to low-level prefix instructions.