NAME

B::Concise - Walk Perl syntax tree, printing concise info about ops

SYNOPSIS

```
perl -MO=Concise[,OPTIONS] foo.pl

use B::Concise qw(set_style add_callback);
```

DESCRIPTION

This compiler backend prints the internal OPs of a Perl program's syntax tree in one of several space-efficient text formats suitable for debugging the inner workings of perl or other compiler backends. It can print OPs in the order they appear in the OP tree, in the order they will execute, or in a text approximation to their tree structure, and the format of the information displayed is customizable. Its function is similar to that of perl's **-Dx** debugging flag or the **B::Terse** module, but it is more sophisticated and flexible.

EXAMPLE

Here's an example of 2 outputs (aka 'renderings'), using the -exec and -basic (i.e. default) formatting conventions on the same code snippet.

```
% perl -MO=Concise,-exec -e '$a = $b + 42'
1  <0> enter
2  <;> nextstate(main 1 -e:1) v
3  <#> gvsv[*b] s
4  <$> const[IV 42] s
*  5  <2> add[t3] sK/2
6  <#> gvsv[*a] s
7  <2> sassign vKS/2
8  <@> leave[1 ref] vKP/REFC
```

Each line corresponds to an opcode. The opcode marked with '*' is used in a few examples below.

The 1st column is the op's sequence number, starting at 1, and is displayed in base 36 by default. This rendering is in -exec (i.e. execution) order.

The symbol between angle brackets indicates the op's type, for example; <2> is a BINOP, <@> a LISTOP, and <#> is a PADOP, which is used in threaded perls. (see *OP class abbreviations*).

The opname, as in **'add[t1]'**, which may be followed by op-specific information in parentheses or brackets (ex **'[t1]'**).

The op-flags (ex **'sK/2'**) follow, and are described in (*OP flags abbreviations*).

```
% perl -MO=Concise -e '$a = $b + 42'
8  <@> leave[1 ref] vKP/REFC ->(end)
1     <0> enter ->2
2     <;> nextstate(main 1 -e:1) v ->3
7     <2> sassign vKS/2 ->8
*  5        <2> add[t1] sK/2 ->6
-           <1> ex-rv2sv sK/1 ->4
3              <$> gvsv(*b) s ->4
4           <$> const(IV 42) s ->5
-        <1> ex-rv2sv sKRM*/1 ->7
6           <$> gvsv(*a) s ->7
```

The default rendering is top-down, so they're not in execution order. This form reflects the way the

stack is used to parse and evaluate expressions; the add operates on the two terms below it in the tree.

Nullops appear as `ex-opname`, where *opname* is an op that has been optimized away by perl. They're displayed with a sequence-number of '-', because they are not executed (they don't appear in previous example), they're printed here because they reflect the parse.

The arrow points to the sequence number of the next op; they're not displayed in -exec mode, for obvious reasons.

Note that because this rendering was done on a non-threaded perl, the PADOPs in the previous examples are now SVOPs, and some (but not all) of the square brackets have been replaced by round ones. This is a subtle feature to provide some visual distinction between renderings on threaded and un-threaded perls.

OPTIONS

Arguments that don't start with a hyphen are taken to be the names of subroutines to print the OPs of; if no such functions are specified, the main body of the program (outside any subroutines, and not including use'd or require'd files) is rendered. Passing `BEGIN`, `CHECK`, `INIT`, or `END` will cause all of the corresponding special blocks to be printed.

Options affect how things are rendered (ie printed). They're presented here by their visual effect, 1st being strongest. They're grouped according to how they interrelate; within each group the options are mutually exclusive (unless otherwise stated).

Options for Opcode Ordering

These options control the 'vertical display' of opcodes. The display 'order' is also called 'mode' elsewhere in this document.

-basic

Print OPs in the order they appear in the OP tree (a preorder traversal, starting at the root). The indentation of each OP shows its level in the tree, and the '->' at the end of the line indicates the next opcode in execution order. This mode is the default, so the flag is included simply for completeness.

-exec

Print OPs in the order they would normally execute (for the majority of constructs this is a postorder traversal of the tree, ending at the root). In most cases the OP that usually follows a given OP will appear directly below it; alternate paths are shown by indentation. In cases like loops when control jumps out of a linear path, a 'goto' line is generated.

-tree

Print OPs in a text approximation of a tree, with the root of the tree at the left and 'left-to-right' order of children transformed into 'top-to-bottom'. Because this mode grows both to the right and down, it isn't suitable for large programs (unless you have a very wide terminal).

Options for Line-Style

These options select the line-style (or just style) used to render each opcode, and dictates what info is actually printed into each line.

-concise

Use the author's favorite set of formatting conventions. This is the default, of course.

-terse

Use formatting conventions that emulate the output of **B::Terse**. The basic mode is almost indistinguishable from the real **B::Terse**, and the exec mode looks very similar, but is in a more logical order and lacks curly brackets. **B::Terse** doesn't have a tree mode, so the tree

mode is only vaguely reminiscent of **B::Terse**.

-linenoise

Use formatting conventions in which the name of each OP, rather than being written out in full, is represented by a one- or two-character abbreviation. This is mainly a joke.

-debug

Use formatting conventions reminiscent of **B::Debug**; these aren't very concise at all.

-env

Use formatting conventions read from the environment variables `B_CONCISE_FORMAT`, `B_CONCISE_GOTO_FORMAT`, and `B_CONCISE_TREE_FORMAT`.

Options for tree-specific formatting

-compact

Use a tree format in which the minimum amount of space is used for the lines connecting nodes (one character in most cases). This squeezes out a few precious columns of screen real estate.

-loose

Use a tree format that uses longer edges to separate OP nodes. This format tends to look better than the compact one, especially in ASCII, and is the default.

-vt

Use tree connecting characters drawn from the VT100 line-drawing set. This looks better if your terminal supports it.

-ascii

Draw the tree with standard ASCII characters like + and `|`. These don't look as clean as the VT100 characters, but they'll work with almost any terminal (or the horizontal scrolling mode of less(1)) and are suitable for text documentation or email. This is the default.

These are pairwise exclusive, i.e. compact or loose, vt or ascii.

Options controlling sequence numbering

-base*n*

Print OP sequence numbers in base *n*. If *n* is greater than 10, the digit for 11 will be 'a', and so on. If *n* is greater than 36, the digit for 37 will be 'A', and so on until 62. Values greater than 62 are not currently supported. The default is 36.

-bigendian

Print sequence numbers with the most significant digit first. This is the usual convention for Arabic numerals, and the default.

-littleendian

Print seqence numbers with the least significant digit first. This is obviously mutually exclusive with bigendian.

Other options

These are pairwise exclusive.

-main

Include the main program in the output, even if subroutines were also specified. This rendering is normally suppressed when a subroutine name or reference is given.

-nomain

This restores the default behavior after you've changed it with '-main' (it's not normally needed). If no subroutine name/ref is given, main is rendered, regardless of this flag.

-nobanner

Renderings usually include a banner line identifying the function name or stringified subref. This suppresses the printing of the banner.

TBC: Remove the stringified coderef; while it provides a 'cookie' for each function rendered, the cookies used should be 1,2,3.. not a random hex-address. It also complicates string comparison of two different trees.

-banner

restores default banner behavior.

-banneris => subref

TBC: a hookpoint (and an option to set it) for a user-supplied function to produce a banner appropriate for users needs. It's not ideal, because the rendering-state variables, which are a natural candidate for use in concise.t, are unavailable to the user.

Option Stickiness

If you invoke Concise more than once in a program, you should know that the options are 'sticky'. This means that the options you provide in the first call will be remembered for the 2nd call, unless you re-specify or change them.

ABBREVIATIONS

The concise style uses symbols to convey maximum info with minimal clutter (like hex addresses). With just a little practice, you can start to see the flowers, not just the branches, in the trees.

OP class abbreviations

These symbols appear before the op-name, and indicate the B:: namespace that represents the ops in your Perl code.

```
0      OP (aka BASEOP)  An OP with no children
1      UNOP             An OP with one child
2      BINOP            An OP with two children
|      LOGOP            A control branch OP
@      LISTOP           An OP that could have lots of children
/      PMOP             An OP with a regular expression
$      SVOP             An OP with an SV
"      PVOP             An OP with a string
{      LOOP             An OP that holds pointers for a loop
;      COP              An OP that marks the start of a statement
#      PADOP            An OP with a GV on the pad
```

OP flags abbreviations

OP flags are either public or private. The public flags alter the behavior of each opcode in consistent ways, and are represented by 0 or more single characters.

```
v      OPf_WANT_VOID    Want nothing (void context)
s      OPf_WANT_SCALAR  Want single value (scalar context)
l      OPf_WANT_LIST    Want list of any length (list context)
                        Want is unknown
K      OPf_KIDS         There is a firstborn child.
P      OPf_PARENS       This operator was parenthesized.
                         (Or block needs explicit scope entry.)
R      OPf_REF          Certified reference.
                         (Return container, not containee).
```

```
M      OPf_MOD          Will modify (lvalue).
S      OPf_STACKED      Some arg is arriving on the stack.
*      OPf_SPECIAL      Do something weird for this op (see op.h)
```

Private flags, if any are set for an opcode, are displayed after a '/'

```
8  <@> leave[1 ref] vKP/REFC ->(end)
7     <2> sassign vKS/2 ->8
```

They're opcode specific, and occur less often than the public ones, so they're represented by short mnemonics instead of single-chars; see *op.h* for gory details, or try this quick 2-liner:

```
$> perl -MB::Concise -de 1
DB<1> |x \%B::Concise::priv
```

FORMATTING SPECIFICATIONS

For each line-style ('concise', 'terse', 'linenoise', etc.) there are 3 format-specs which control how OPs are rendered.

The first is the 'default' format, which is used in both basic and exec modes to print all opcodes. The 2nd, goto-format, is used in exec mode when branches are encountered. They're not real opcodes, and are inserted to look like a closing curly brace. The tree-format is tree specific.

When a line is rendered, the correct format-spec is copied and scanned for the following items; data is substituted in, and other manipulations like basic indenting are done, for each opcode rendered.

There are 3 kinds of items that may be populated; special patterns, #vars, and literal text, which is copied verbatim. (Yes, it's a set of s///g steps.)

Special Patterns

These items are the primitives used to perform indenting, and to select text from amongst alternatives.

(x(*exec_text***;***basic_text***)x)**

Generates *exec_text* in exec mode, or *basic_text* in basic mode.

(*(*text***)*)**

Generates one copy of *text* for each indentation level.

(*(*text1***;***text2***)*)**

Generates one fewer copies of *text1* than the indentation level, followed by one copy of *text2* if the indentation level is more than 0.

(?(*text1#varText2***)?)**

If the value of *var* is true (not empty or zero), generates the value of *var* surrounded by *text1* and *Text2*, otherwise nothing.

~

Any number of tildes and surrounding whitespace will be collapsed to a single space.

Variables

These #vars represent opcode properties that you may want as part of your rendering. The '#' is intended as a private sigil; a #var's value is interpolated into the style-line, much like "read $this".

These vars take 3 forms:

#var

A property named 'var' is assumed to exist for the opcodes, and is interpolated into the rendering.

#varN

Generates the value of *var*, left justified to fill *N* spaces. Note that this means while you can have properties 'foo' and 'foo2', you cannot render 'foo2', but you could with 'foo2a'. You would be wise not to rely on this behavior going forward ;-)

#Var

This ucfirst form of #var generates a tag-value form of itself for display; it converts '#Var' into a 'Var => #var' style, which is then handled as described above. (Imp-note: #Vars cannot be used for conditional-fills, because the => #var transform is done after the check for #Var's value).

The following variables are 'defined' by B::Concise; when they are used in a style, their respective values are plugged into the rendering of each opcode.

Only some of these are used by the standard styles, the others are provided for you to delve into optree mechanics, should you wish to add a new style (see *add_style* below) that uses them. You can also add new ones using *add_callback*.

#addr

The address of the OP, in hexadecimal.

#arg

The OP-specific information of the OP (such as the SV for an SVOP, the non-local exit pointers for a LOOP, etc.) enclosed in parentheses.

#class

The B-determined class of the OP, in all caps.

#classsym

A single symbol abbreviating the class of the OP.

#coplabel

The label of the statement or block the OP is the start of, if any.

#exname

The name of the OP, or 'ex-foo' if the OP is a null that used to be a foo.

#extarg

The target of the OP, or nothing for a nulled OP.

#firstaddr

The address of the OP's first child, in hexadecimal.

#flags

The OP's flags, abbreviated as a series of symbols.

#flagval

The numeric value of the OP's flags.

#hyphseq

The sequence number of the OP, or a hyphen if it doesn't have one.

#label

'NEXT', 'LAST', or 'REDO' if the OP is a target of one of those in exec mode, or empty

#lastaddr otherwise.

The address of the OP's last child, in hexadecimal.

#name

The OP's name.

#NAME

The OP's name, in all caps.

#next

The sequence number of the OP's next OP.

#nextaddr

The address of the OP's next OP, in hexadecimal.

#noise

A one- or two-character abbreviation for the OP's name.

#private

The OP's private flags, rendered with abbreviated names if possible.

#privval

The numeric value of the OP's private flags.

#seq

The sequence number of the OP. Note that this is a sequence number generated by B::Concise.

#seqnum

5.8.x and earlier only. 5.9 and later do not provide this.

The real sequence number of the OP, as a regular number and not adjusted to be relative to the start of the real program. (This will generally be a fairly large number because all of **B::Concise** is compiled before your program is).

#opt

Whether or not the op has been optimised by the peephole optimiser.

Only available in 5.9 and later.

#static

Whether or not the op is statically defined. This flag is used by the B::C compiler backend and indicates that the op should not be freed.

Only available in 5.9 and later.

#sibaddr

The address of the OP's next youngest sibling, in hexadecimal.

#svaddr

The address of the OP's SV, if it has an SV, in hexadecimal.

#svclass

The class of the OP's SV, if it has one, in all caps (e.g., 'IV').

#svval

The value of the OP's SV, if it has one, in a short human-readable format.

#targ

The numeric value of the OP's targ.

#targarg

The name of the variable the OP's targ refers to, if any, otherwise the letter t followed by the OP's targ in decimal.

#targarglife

Same as **#targarg**, but followed by the COP sequence numbers that delimit the variable's lifetime (or 'end' for a variable in an open scope) for a variable.

#typenum

The numeric value of the OP's type, in decimal.

Using B::Concise outside of the O framework

The common (and original) usage of B::Concise was for command-line renderings of simple code, as given in EXAMPLE. But you can also use **B::Concise** from your code, and call compile() directly, and repeatedly. By doing so, you can avoid the compile-time only operation of O.pm, and even use the debugger to step through B::Concise::compile() itself.

Once you're doing this, you may alter Concise output by adding new rendering styles, and by optionally adding callback routines which populate new variables, if such were referenced from those (just added) styles.

Example: Altering Concise Renderings

```
use B::Concise qw(set_style add_callback);
add_style($yourStyleName => $defaultfmt, $gotofmt, $treefmt);
add_callback
  ( sub {
        my ($h, $op, $format, $level, $stylename) = @_;
        $h->{variable} = some_func($op);
    });
$walker = B::Concise::compile(@options,@subnames,@subrefs);
$walker->();
```

set_style()

set_style accepts 3 arguments, and updates the three format-specs comprising a line-style (basic-exec, goto, tree). It has one minor drawback though; it doesn't register the style under a new name. This can become an issue if you render more than once and switch styles. Thus you may prefer to use add_style() and/or set_style_standard() instead.

set_style_standard($name)

This restores one of the standard line-styles: `terse`, `concise`, `linenoise`, `debug`, `env`, into effect. It also accepts style names previously defined with add_style().

add_style()

This subroutine accepts a new style name and three style arguments as above, and creates, registers, and selects the newly named style. It is an error to re-add a style; call set_style_standard() to switch between several styles.

add_callback()

If your newly minted styles refer to any new #variables, you'll need to define a callback subroutine that will populate (or modify) those variables. They are then available for use in the style you've chosen.

The callbacks are called for each opcode visited by Concise, in the same order as they are added. Each subroutine is passed five parameters.

1. A hashref, containing the variable names and values which are
   populated into the report-line for the op
2. the op, as a B<B::OP> object
3. a reference to the format string
4. the formatting (indent) level
5. the selected stylename

To define your own variables, simply add them to the hash, or change existing values if you need to. The level and format are passed in as references to scalars, but it is unlikely that they will need to be changed or even used.

Running B::Concise::compile()

compile accepts options as described above in *OPTIONS*, and arguments, which are either coderefs, or subroutine names.

It constructs and returns a $treewalker coderef, which when invoked, traverses, or walks, and renders the optrees of the given arguments to STDOUT. You can reuse this, and can change the rendering style used each time; thereafter the coderef renders in the new style.

walk_output lets you change the print destination from STDOUT to another open filehandle, or into a string passed as a ref (unless you've built perl with -Uuseperlio).

```
my $walker = B::Concise::compile('-terse','aFuncName', \&aSubRef);  # 1
walk_output(\my $buf);
$walker->();   # 1 renders -terse
set_style_standard('concise'); # 2
$walker->();   # 2 renders -concise
$walker->(@new);   # 3 renders whatever
print "3 different renderings: terse, concise, and @new: $buf\n";
```

When $walker is called, it traverses the subroutines supplied when it was created, and renders them using the current style. You can change the style afterwards in several different ways:

1. call C<compile>, altering style or mode/order
2. call C<set_style_standard>
3. call $walker, passing @new options

Passing new options to the $walker is the easiest way to change amongst any pre-defined styles (the ones you add are automatically recognized as options), and is the only way to alter rendering order without calling compile again. Note however that rendering state is still shared amongst multiple $walker objects, so they must still be used in a coordinated manner.

B::Concise::reset_sequence()

This function (not exported) lets you reset the sequence numbers (note that they're numbered arbitrarily, their goal being to be human readable). Its purpose is mostly to support testing, i.e. to compare the concise output from two identical anonymous subroutines (but different instances). Without the reset, B::Concise, seeing that they're separate optrees, generates different sequence numbers in the output.

Errors

Errors in rendering (non-existent function-name, non-existent coderef) are written to the STDOUT, or wherever you've set it via walk_output().

Errors using the various *style* calls, and bad args to walk_output(), result in die(). Use an eval if you wish to catch these errors and continue processing.

AUTHOR

Stephen McCamant, <smcc@CSUA.Berkeley.EDU>.