

# Mapping Op Trees

by Eric Maki  
eric@uc.org



This issue's cover illustration is a Perl self-portrait: a picture of picture-generating code. It is a graphic representation of the op trees that make up the program generating the graphic. Strictly speaking, the program itself only outputs a structured description of the op trees, using some of Perl's introspection modules. The heavy lifting of actually plotting digraphs was done by dot, part of the GraphViz package.

This article pokes into Perl internals a little, but from the perspective of the public introspection interface provided by core Perl modules. Results may vary considerably from one version of Perl to another. The details of compilation provided are extremely abstracted.

## ■ What are op trees?

Perl is not compiled to native machine code, but it is compiled. A compiled Perl program resides in memory as a sort of bytecode. This bytecode consists of trees of operations against the Perl stack. Like many "virtual machines," the Perl runtime engine is stack-based rather than register-based. These operations, called "PP ops" for "push/pop," generally pop items from the top of the stack, operate on them, and push results back on to the stack.

As you might expect, perl starts parsing this expression:

```
$a and $b or $c
```

by building a parse tree as shown in Figure 1.

Here the lowest precedent operator is at the top, and the highest at the bottom. The padsv opcodes simply fetch a lexical variable from the surrounding scope pad, and place it on the stack. The null opcode here is a relic of the parsing.

Once the parse tree has been completed, perl then plots an execution path through the tree by joining the opnodes' next pointers together. In the case of the and and or ops, which are logical operators (LOGOP), there is an additional other pointer for the alternate outcome of the operation. After linking, our expression from above looks like Figure 2 (next page).

The path that perl plots through the tree is generally an in-order traversal of the tree. The entry point for this evaluation is the leftmost leaf of the tree. I push \$a onto the stack, then evaluate the and. If the top item on the stack is false, I can skip up to the or, via the

next link. If \$a is true, then I need to evaluate \$b. Note that the null opcode is optimized away as a needless step. perl retains it in the tree, but it is never executed.

All of the parse tree links are retained in the final structure, but are ignored at runtime, yielding an execution tree that looks like Figure 3 (page 12).

Such trees quickly become complex. Figure 4 (page 13) shows an example of the tree constructed for this subroutine:

```
sub {  
    $a + $b - $c;  
}
```

The parse tree is relatively complex, but note that there is a single linear path through this code, making it simply a singly-linked list of instructions.

In this case, the variable retrieval and stack pushing ops are gvsv ops because the variables in this case are globals. Globals need to be fetched by name rather than resolved as lexicals. This graph also includes the "op class" (UNOP, BINOP, etc.). Different operations require different numbers of arguments: UNOP takes

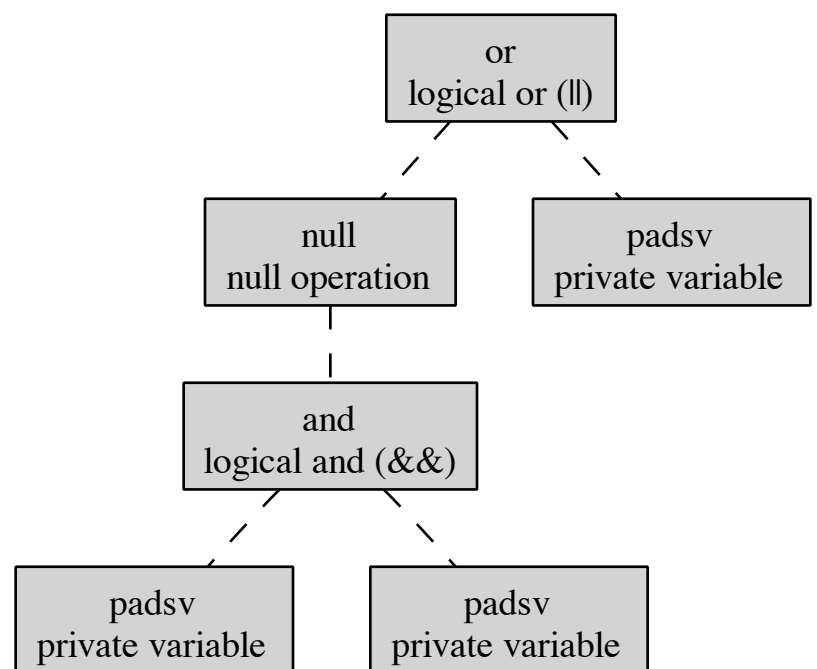


Figure 1: The parse tree for \$a and \$b or \$c.