NAME

perlinterp - An overview of the Perl interpreter

DESCRIPTION

This document provides an overview of how the Perl interpreter works at the level of C code, along with pointers to the relevant C source code files.

ELEMENTS OF THE INTERPRETER

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. "Compiled code" in perlguts explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

Startup

The action begins in *perlmain.c*. (or *miniperlmain.c* for miniperl) This is very high-level code, enough to fit on a single screen, and it resembles the code found in perlembed; most of the real action takes place in *perl.c*

perlmain.c is generated by "ExtUtils::Miniperl" from *miniperlmain.c* at make time, so you should make perl to follow this along.

First, *perlmain.c* allocates some memory and constructs a Perl interpreter, along these lines:

```
1 PERL_SYS_INIT3(&argc,&argv,&env);
2
3 if (!PL_do_undump) {
4     my_perl = perl_alloc();
5     if (!my_perl)
6         exit(1);
7     perl_construct(my_perl);
8     PL_perl_destruct_level = 0;
9 }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references "PL_do_undump", a global variable - all global variables in Perl start with "PL_". This tells you whether the current running program was created with the "-u" flag to perl and then *undump*, which means it's going to be false in any sane context.

Line 4 calls a function in *perl.c* to allocate memory for a Perl interpreter. It's quite a simple function,

and the guts of it looks like this:

```
 my_perl = (PerlInterpreter*)PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: "PerlMem_malloc" is either your system's "malloc", or Perl's own "malloc" as defined in *malloc.c* if you selected that option at configure time.

Next, in line 7, we construct the interpreter using perl_construct, also in *perl.c*; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```
 if (!perl_parse(my_perl, xs_init, argc, argv, (char **)NULL))
     perl_run(my_perl);

 exitstatus = perl_destruct(my_perl);

 perl_free(my_perl);
```

"perl_parse" is actually a wrapper around "S_parse_body", as defined in *perl.c*, which processes the command line options, sets up any statically linked XS modules, opens the program and calls "yyparse" to parse it.

Parsing

The aim of this stage is to take the Perl source, and turn it into an op tree. We'll see what one of those looks like later. Strictly speaking, there's three things going on here.

"yyparse", the parser, lives in *perly.c*, although you're better off reading the original YACC input in *perly.y*. (Yes, Virginia, there **is** a YACC grammar for Perl!) The job of the parser is to take your code and "understand" it, splitting it into sentences, deciding which operands go with which operators and so on.

The parser is nobly assisted by the lexer, which chunks up your input into tokens, and decides what type of thing each token is: a variable name, an operator, a bareword, a subroutine, a core function, and so on. The main point of entry to the lexer is "yylex", and that and its associated routines can be found in *toke.c*. Perl isn't much like other computer languages; it's highly context sensitive at times, it can be tricky to work out what sort of token something is, or where a token ends. As such, there's a lot of interplay between the tokeniser and the parser, which can get pretty frightening if you're not used to it.

As the parser understands a Perl program, it builds up a tree of operations for the interpreter to perform during execution. The routines which construct and link together the various operations are to be found in *op.c*, and will be examined later.

Optimization

Now the parsing stage is complete, and the finished tree represents the operations that the Perl interpreter needs to perform to execute our program. Next, Perl does a dry run over the tree looking for optimisations: constant expressions such as "3 + 4" will be computed now, and the optimizer will also see if any multiple operations can be replaced with a single one. For instance, to fetch the variable $foo, instead of grabbing the glob *foo and looking at the scalar component, the optimizer fiddles the op tree to use a function which directly looks up the scalar in question. The main optimizer is "peep" in *op.c*, and many ops have their own optimizing functions.

Running

Now we're finally ready to go: we have compiled Perl byte code, and all that's left to do is run it. The actual execution is done by the "runops_standard" function in *run.c*; more specifically, it's done by these three innocent looking lines:

```
while ((PL_op = PL_op->op_ppaddr(aTHX))) {
   PERL_ASYNC_CHECK();
}
```

You may be more comfortable with the Perl version of that:

```
PERL_ASYNC_CHECK() while $Perl::op = &{$Perl::op->{function}};
```

Well, maybe not. Anyway, each op contains a function pointer, which stipulates the function which will actually carry out the operation. This function will return the next op in the sequence - this allows for things like "if" which choose the next op dynamically at run time. The "PERL_ASYNC_CHECK" makes sure that things like signals interrupt execution if required.

The actual functions called are known as PP code, and they're spread between four files: *pp_hot.c* contains the "hot" code, which is most often used and highly optimized, *pp_sys.c* contains all the system-specific functions, *pp_ctl.c* contains the functions which implement control structures ("if", "while" and the like) and *pp.c* contains everything else. These are, if you like, the C code for Perl's built-in functions and operators.

Note that each "pp_" function is expected to return a pointer to the next op. Calls to perl subs (and eval blocks) are handled within the same runops loop, and do not consume extra space on the C stack. For example, "pp_entersub" and "pp_entertry" just push a "CxSUB" or "CxEVAL" block struct onto the

context stack which contain the address of the op following the sub call or eval. They then return the first op of that sub or eval block, and so execution continues of that sub or block. Later, a "pp_leavesub" or "pp_leavetry" op pops the "CxSUB" or "CxEVAL", retrieves the return op from it, and returns it.

Exception handing

Perl's exception handing (i.e. "die" etc.) is built on top of the low-level "setjmp()"/"longjmp()" C-library functions. These basically provide a way to capture the current PC and SP registers and later restore them; i.e. a "longjmp()" continues at the point in code where a previous "setjmp()" was done, with anything further up on the C stack being lost. This is why code should always save values using "SAVE_*FOO*" rather than in auto variables.

The perl core wraps "setjmp()" etc in the macros "JMPENV_PUSH" and "JMPENV_JUMP". The basic rule of perl exceptions is that "exit", and "die" (in the absence of "eval") perform a JMPENV_JUMP(2), while "die" within "eval" does a JMPENV_JUMP(3).

At entry points to perl, such as "perl_parse()", "perl_run()" and "call_sv(cv, G_EVAL)" each does a "JMPENV_PUSH", then enter a runops loop or whatever, and handle possible exception returns. For a 2 return, final cleanup is performed, such as popping stacks and calling "CHECK" or "END" blocks. Amongst other things, this is how scope cleanup still occurs during an "exit".

If a "die" can find a "CxEVAL" block on the context stack, then the stack is popped to that level and the return op in that block is assigned to "PL_restartop"; then a JMPENV_JUMP(3) is performed. This normally passes control back to the guard. In the case of "perl_run" and "call_sv", a non-null "PL_restartop" triggers re-entry to the runops loop. The is the normal way that "die" or "croak" is handled within an "eval".

Sometimes ops are executed within an inner runops loop, such as tie, sort or overload code. In this case, something like

```
sub FETCH { eval { die } }
```

would cause a longjmp right back to the guard in "perl_run", popping both runops loops, which is clearly incorrect. One way to avoid this is for the tie code to do a "JMPENV_PUSH" before executing "FETCH" in the inner runops loop, but for efficiency reasons, perl in fact just sets a flag, using "CATCH_SET(TRUE)". The "pp_require", "pp_entereval" and "pp_entertry" ops check this flag, and if true, they call "docatch", which does a "JMPENV_PUSH" and starts a new runops level to execute the code, rather than doing it on the current loop.

As a further optimisation, on exit from the eval block in the "FETCH", execution of the code following

the block is still carried on in the inner loop. When an exception is raised, "docatch" compares the "JMPENV" level of the "CxEVAL" with "PL_top_env" and if they differ, just re-throws the exception. In this way any inner loops get popped.

Here’s an example.

```
1: eval { tie @a, ’A’ };
2: sub A::TIEARRAY {
3:     eval { die };
4:     die;
5: }
```

To run this code, "perl_run" is called, which does a "JMPENV_PUSH" then enters a runops loop. This loop executes the eval and tie ops on line 1, with the eval pushing a "CxEVAL" onto the context stack.

The "pp_tie" does a "CATCH_SET(TRUE)", then starts a second runops loop to execute the body of "TIEARRAY". When it executes the entertry op on line 3, "CATCH_GET" is true, so "pp_entertry" calls "docatch" which does a "JMPENV_PUSH" and starts a third runops loop, which then executes the die op. At this point the C call stack looks like this:

```
Perl_pp_die
Perl_runops     # third loop
S_docatch_body
S_docatch
Perl_pp_entertry
Perl_runops     # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops     # first loop
S_run_body
perl_run
main
```

and the context and data stacks, as shown by "-Dstv", look like:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
```

```
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
  CX 1: EVAL   => *
retop=nextstate
```

The die pops the first "CxEVAL" off the context stack, sets "PL_restartop" from it, does a JMPENV_JUMP(3), and control returns to the top "docatch". This then starts another third-level runops level, which executes the nextstate, pushmark and die ops on line 4. At the point that the second "pp_die" is called, the C call stack looks exactly like that above, even though we are no longer within an inner eval; this is because of the optimization mentioned earlier. However, the context stack now looks like this, ie with the top CxEVAL popped:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
```

The die on line 4 pops the context stack back down to the CxEVAL, leaving it as:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
```

As usual, "PL_restartop" is extracted from the "CxEVAL", and a JMPENV_JUMP(3) done, which pops the C stack back to the docatch:

```
S_docatch
Perl_pp_entertry
Perl_runops     # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops     # first loop
S_run_body
perl_run
main
```

In this case, because the "JMPENV" level recorded in the "CxEVAL" differs from the current one, "docatch" just does a JMPENV_JUMP(3) and the C stack unwinds to:

```
perl_run
main
```

Because "PL_restartop" is non-null, "run_body" starts a new runops loop and execution continues.

INTERNAL VARIABLE TYPES

You should by now have had a look at perlguts, which tells you about Perl's internal variable types: SVs, HVs, AVs and the rest. If not, do that now.

These variables are used not only to represent Perl-space variables, but also any constants in the code, as well as some structures completely internal to Perl. The symbol table, for instance, is an ordinary Perl hash. Your code is represented by an SV as it's read into the parser; any program files you call are opened via ordinary Perl filehandles, and so on.

The core Devel::Peek module lets us examine SVs from a Perl program. Let's see, for instance, how Perl treats the constant "hello".

```
  % perl -MDevel::Peek -e 'Dump("hello")'
1 SV = PV(0xa041450) at 0xa04ecbc
2   REFCNT = 1
3   FLAGS = (POK,READONLY,pPOK)
4   PV = 0xa0484e0 "hello"\0
5   CUR = 5
6   LEN = 6
```

Reading "Devel::Peek" output takes a bit of practise, so let's go through it line by line.

Line 1 tells us we're looking at an SV which lives at 0xa04ecbc in memory. SVs themselves are very simple structures, but they contain a pointer to a more complex structure. In this case, it's a PV, a structure which holds a string value, at location 0xa041450. Line 2 is the reference count; there are no other references to this data, so it's 1.

Line 3 are the flags for this SV - it's OK to use it as a PV, it's a read-only SV (because it's a constant) and the data is a PV internally. Next we've got the contents of the string, starting at location 0xa0484e0.

Line 5 gives us the current length of the string - note that this does **not** include the null terminator. Line

6 is not the length of the string, but the length of the currently allocated buffer; as the string grows, Perl automatically extends the available storage via a routine called "SvGROW".

You can get at any of these quantities from C very easily; just add "Sv" to the name of the field shown in the snippet, and you've got a macro which will return the value: "SvCUR(sv)" returns the current length of the string, "SvREFCOUNT(sv)" returns the reference count, "SvPV(sv, len)" returns the string itself with its length, and so on. More macros to manipulate these properties can be found in perlguts.

Let's take an example of manipulating a PV, from "sv_catpvn", in *sv.c*

```
 1  void
 2  Perl_sv_catpvn(pTHX_ SV *sv, const char *ptr, STRLEN len)
 3  {
 4      STRLEN tlen;
 5      char *junk;

 6      junk = SvPV_force(sv, tlen);
 7      SvGROW(sv, tlen + len + 1);
 8      if (ptr == junk)
 9          ptr = SvPVX(sv);
10      Move(ptr,SvPVX(sv)+tlen,len,char);
11      SvCUR(sv) += len;
12      *SvEND(sv) = '\0';
13      (void)SvPOK_only_UTF8(sv);          /* validate pointer */
14      SvTAINT(sv);
15  }
```

This is a function which adds a string, "ptr", of length "len" onto the end of the PV stored in "sv". The first thing we do in line 6 is make sure that the SV **has** a valid PV, by calling the "SvPV_force" macro to force a PV. As a side effect, "tlen" gets set to the current value of the PV, and the PV itself is returned to "junk".

In line 7, we make sure that the SV will have enough room to accommodate the old string, the new string and the null terminator. If "LEN" isn't big enough, "SvGROW" will reallocate space for us.

Now, if "junk" is the same as the string we're trying to add, we can grab the string directly from the SV; "SvPVX" is the address of the PV in the SV.

Line 10 does the actual catenation: the "Move" macro moves a chunk of memory around: we move the

string "ptr" to the end of the PV - that's the start of the PV plus its current length. We're moving "len" bytes of type "char". After doing so, we need to tell Perl we've extended the string, by altering "CUR" to reflect the new length. "SvEND" is a macro which gives us the end of the string, so that needs to be a "\0".

Line 13 manipulates the flags; since we've changed the PV, any IV or NV values will no longer be valid: if we have "$a=10; $a.="6";" we don't want to use the old IV of 10. "SvPOK_only_utf8" is a special UTF-8-aware version of "SvPOK_only", a macro which turns off the IOK and NOK flags and turns on POK. The final "SvTAINT" is a macro which launders tainted data if taint mode is turned on.

AVs and HVs are more complicated, but SVs are by far the most common variable type being thrown around. Having seen something of how we manipulate these, let's go on and look at how the op tree is constructed.

OP TREES

First, what is the op tree, anyway? The op tree is the parsed representation of your program, as we saw in our section on parsing, and it's the sequence of operations that Perl goes through to execute your program, as we saw in "Running".

An op is a fundamental operation that Perl can perform: all the built-in functions and operators are ops, and there are a series of ops which deal with concepts the interpreter needs internally - entering and leaving a block, ending a statement, fetching a variable, and so on.

The op tree is connected in two ways: you can imagine that there are two "routes" through it, two orders in which you can traverse the tree. First, parse order reflects how the parser understood the code, and secondly, execution order tells perl what order to perform the operations in.

The easiest way to examine the op tree is to stop Perl after it has finished parsing, and get it to dump out the tree. This is exactly what the compiler backends B::Terse, B::Concise and CPAN module <B::Debug do.

Let's have a look at how Perl sees "$a = $b + $c":

```
% perl -MO=Terse -e '$a=$b+$c'
1  LISTOP (0x8179888) leave
2      OP (0x81798b0) enter
3      COP (0x8179850) nextstate
4      BINOP (0x8179828) sassign
5          BINOP (0x8179800) add [1]
6              UNOP (0x81796e0) null [15]
```

```
 7           SVOP (0x80fafe0) gvsv  GV (0x80fa4cc) *b
 8         UNOP (0x81797e0) null [15]
 9           SVOP (0x8179700) gvsv  GV (0x80efeb0) *c
10       UNOP (0x816b4f0) null [15]
11         SVOP (0x816dcf0) gvsv  GV (0x80fa460) *a
```

Let’s start in the middle, at line 4. This is a BINOP, a binary operator, which is at location 0x8179828. The specific operator in question is "sassign" - scalar assignment - and you can find the code which implements it in the function "pp_sassign" in *pp_hot.c*. As a binary operator, it has two children: the add operator, providing the result of "$b+$c", is uppermost on line 5, and the left hand side is on line 10.

Line 10 is the null op: this does exactly nothing. What is that doing there? If you see the null op, it’s a sign that something has been optimized away after parsing. As we mentioned in "Optimization", the optimization stage sometimes converts two operations into one, for example when fetching a scalar variable. When this happens, instead of rewriting the op tree and cleaning up the dangling pointers, it’s easier just to replace the redundant operation with the null op. Originally, the tree would have looked like this:

```
10       SVOP (0x816b4f0) rv2sv [15]
11         SVOP (0x816dcf0) gv  GV (0x80fa460) *a
```

That is, fetch the "a" entry from the main symbol table, and then look at the scalar component of it: "gvsv" ("pp_gvsv" in *pp_hot.c*) happens to do both these things.

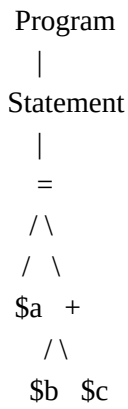
The right hand side, starting at line 5 is similar to what we’ve just seen: we have the "add" op ("pp_add", also in *pp_hot.c*) add together two "gvsv"s.

Now, what’s this about?

```
1  LISTOP (0x8179888) leave
2    OP (0x81798b0) enter
3    COP (0x8179850) nextstate
```

"enter" and "leave" are scoping ops, and their job is to perform any housekeeping every time you enter and leave a block: lexical variables are tidied up, unreferenced variables are destroyed, and so on. Every program will have those first three lines: "leave" is a list, and its children are all the statements in the block. Statements are delimited by "nextstate", so a block is a collection of "nextstate" ops, with the ops to be performed for each statement being the children of "nextstate". "enter" is a single op which functions as a marker.

That’s how Perl parsed the program, from top to bottom:



However, it’s impossible to **perform** the operations in this order: you have to find the values of $b and $c before you add them together, for instance. So, the other thread that runs through the op tree is the execution order: each op has a field "op_next" which points to the next op to be run, so following these pointers tells us how perl executes the code. We can traverse the tree in this order using the "exec" option to "B::Terse":

```
% perl -MO=Terse,exec -e ’$a=$b+$c’
1  OP (0x8179928) enter
2  COP (0x81798c8) nextstate
3  SVOP (0x81796c8) gvsv  GV (0x80fa4d4) *b
4  SVOP (0x8179798) gvsv  GV (0x80efeb0) *c
5  BINOP (0x8179878) add [1]
6  SVOP (0x816dd38) gvsv  GV (0x80fa468) *a
7  BINOP (0x81798a0) sassign
8  LISTOP (0x8179900) leave
```

This probably makes more sense for a human: enter a block, start a statement. Get the values of $b and $c, and add them together. Find $a, and assign one to the other. Then leave.

The way Perl builds up these op trees in the parsing process can be unravelled by examining *toke.c*, the lexer, and *perly.y*, the YACC grammar. Let’s look at the code that constructs the tree for "$a = $b + $c".

First, we’ll look at the "Perl_yylex" function in the lexer. We want to look for "case ’x’", where x is the first character of the operator. (Incidentally, when looking for the code that handles a keyword, you’ll want to search for "KEY_foo" where "foo" is the keyword.) Here is the code that handles assignment (there are quite a few operators beginning with "=", so most of it is omitted for brevity):

```
1    case '=':
2        s++;
         ... code that handles == => etc. and pod ...
3        pl_yylval.ival = 0;
4        OPERATOR(ASSIGNOP);
```

We can see on line 4 that our token type is "ASSIGNOP" ("OPERATOR" is a macro, defined in *toke.c*, that returns the token type, among other things). And "+":

```
1    case '+':
2        {
3            const char tmp = *s++;
             ... code for ++ ...
4            if (PL_expect == XOPERATOR) {
                 ...
5                Aop(OP_ADD);
6            }
             ...
7        }
```

Line 4 checks what type of token we are expecting. "Aop" returns a token. If you search for "Aop" elsewhere in *toke.c*, you will see that it returns an "ADDOP" token.

Now that we know the two token types we want to look for in the parser, let's take the piece of *perly.y* we need to construct the tree for "$a = $b + $c"

```
1 term    :   term ASSIGNOP term
2             { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
3         |   term ADDOP term
4             { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

If you're not used to reading BNF grammars, this is how it works: You're fed certain things by the tokeniser, which generally end up in upper case. "ADDOP" and "ASSIGNOP" are examples of "terminal symbols", because you can't get any simpler than them.

The grammar, lines one and three of the snippet above, tells you how to build up more complex forms. These complex forms, "non-terminal symbols" are generally placed in lower case. "term" here is a non-terminal symbol, representing a single expression.

The grammar gives you the following rule: you can make the thing on the left of the colon if you see all

the things on the right in sequence. This is called a "reduction", and the aim of parsing is to completely reduce the input. There are several different ways you can perform a reduction, separated by vertical bars: so, "term" followed by "=" followed by "term" makes a "term", and "term" followed by "+" followed by "term" can also make a "term".

So, if you see two terms with an "=" or "+", between them, you can turn them into a single expression. When you do this, you execute the code in the block on the next line: if you see "=", you'll do the code in line 2. If you see "+", you'll do the code in line 4. It's this code which contributes to the op tree.

```
|   term ADDOP term
{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

What this does is creates a new binary op, and feeds it a number of variables. The variables refer to the tokens: $1 is the first token in the input, $2 the second, and so on - think regular expression backreferences. $$ is the op returned from this reduction. So, we call "newBINOP" to create a new binary operator. The first parameter to "newBINOP", a function in *op.c*, is the op type. It's an addition operator, so we want the type to be "ADDOP". We could specify this directly, but it's right there as the second token in the input, so we use $2. The second parameter is the op's flags: 0 means "nothing special". Then the things to add: the left and right hand side of our expression, in scalar context.

The functions that create ops, which have names like "newUNOP" and "newBINOP", call a "check" function associated with each op type, before returning the op. The check functions can mangle the op as they see fit, and even replace it with an entirely new one. These functions are defined in *op.c*, and have a "Perl_ck_" prefix. You can find out which check function is used for a particular op type by looking in *regen/opcodes*. Take "OP_ADD", for example. ("OP_ADD" is the token value from the "Aop(OP_ADD)" in *toke.c* which the parser passes to "newBINOP" as its first argument.) Here is the relevant line:

```
add             addition (+)            ck_null         IfsT2   S S
```

The check function in this case is "Perl_ck_null", which does nothing. Let's look at a more interesting case:

```
readline        <HANDLE>                ck_readline     t%      F?
```

And here is the function from *op.c*:

```
1 OP *
2 Perl_ck_readline(pTHX_ OP *o)
3 {
```

```
 4    PERL_ARGS_ASSERT_CK_READLINE;
 5
 6    if (o->op_flags & OPf_KIDS) {
 7        OP *kid = cLISTOPo->op_first;
 8        if (kid->op_type == OP_RV2GV)
 9            kid->op_private |= OPpALLOW_FAKE;
10    }
11    else {
12        OP * const newop
13            = newUNOP(OP_READLINE, 0, newGVOP(OP_GV, 0,
14                                              PL_argvgv));
15        op_free(o);
16        return newop;
17    }
18    return o;
19 }
```

One particularly interesting aspect is that if the op has no kids (i.e., "readline()" or "<>") the op is freed and replaced with an entirely new one that references *ARGV (lines 12-16).

STACKS

When perl executes something like "addop", how does it pass on its results to the next op? The answer is, through the use of stacks. Perl has a number of stacks to store things it’s currently working on, and we’ll look at the three most important ones here.

Argument stack

Arguments are passed to PP code and returned from PP code using the argument stack, "ST". The typical way to handle arguments is to pop them off the stack, deal with them how you wish, and then push the result back onto the stack. This is how, for instance, the cosine operator works:

```
NV value;
value = POPn;
value = Perl_cos(value);
XPUSHn(value);
```

We’ll see a more tricky example of this when we consider Perl’s macros below. "POPn" gives you the NV (floating point value) of the top SV on the stack: the $x in "cos($x)". Then we compute the cosine, and push the result back as an NV. The "X" in "XPUSHn" means that the stack should be extended if necessary - it can’t be necessary here, because we know there’s room for one more item on the stack, since we’ve just removed one! The "XPUSH*" macros at least guarantee safety.

Alternatively, you can fiddle with the stack directly: "SP" gives you the first element in your portion of the stack, and "TOP*" gives you the top SV/IV/NV/etc. on the stack. So, for instance, to do unary negation of an integer:

```
SETi(-TOPi);
```

Just set the integer value of the top stack entry to its negation.

Argument stack manipulation in the core is exactly the same as it is in XSUBs - see perlxstut, perlxs and perlguts for a longer description of the macros used in stack manipulation.

Mark stack

I say "your portion of the stack" above because PP code doesn't necessarily get the whole stack to itself: if your function calls another function, you'll only want to expose the arguments aimed for the called function, and not (necessarily) let it get at your own data. The way we do this is to have a "virtual" bottom-of-stack, exposed to each function. The mark stack keeps bookmarks to locations in the argument stack usable by each function. For instance, when dealing with a tied variable, (internally, something with "P" magic) Perl has to call methods for accesses to the tied variables. However, we need to separate the arguments exposed to the method to the argument exposed to the original function - the store or fetch or whatever it may be. Here's roughly how the tied "push" is implemented; see "av_push" in *av.c*:

```
1  PUSHMARK(SP);
2  EXTEND(SP,2);
3  PUSHs(SvTIED_obj((SV*)av, mg));
4  PUSHs(val);
5  PUTBACK;
6  ENTER;
7  call_method("PUSH", G_SCALAR|G_DISCARD);
8  LEAVE;
```

Let's examine the whole implementation, for practice:

```
1  PUSHMARK(SP);
```

Push the current state of the stack pointer onto the mark stack. This is so that when we've finished adding items to the argument stack, Perl knows how many things we've added recently.

```
2  EXTEND(SP,2);
3  PUSHs(SvTIED_obj((SV*)av, mg));
```

```
4  PUSHs(val);
```

We're going to add two more items onto the argument stack: when you have a tied array, the "PUSH" subroutine receives the object and the value to be pushed, and that's exactly what we have here - the tied object, retrieved with "SvTIED_obj", and the value, the SV "val".

```
5  PUTBACK;
```

Next we tell Perl to update the global stack pointer from our internal variable: "dSP" only gave us a local copy, not a reference to the global.

```
6  ENTER;
7  call_method("PUSH", G_SCALAR|G_DISCARD);
8  LEAVE;
```

"ENTER" and "LEAVE" localise a block of code - they make sure that all variables are tidied up, everything that has been localised gets its previous value returned, and so on. Think of them as the "{" and "}" of a Perl block.

To actually do the magic method call, we have to call a subroutine in Perl space: "call_method" takes care of that, and it's described in perlcall. We call the "PUSH" method in scalar context, and we're going to discard its return value. The **call_method()** function removes the top element of the mark stack, so there is nothing for the caller to clean up.

Save stack

C doesn't have a concept of local scope, so perl provides one. We've seen that "ENTER" and "LEAVE" are used as scoping braces; the save stack implements the C equivalent of, for example:

```
{
   local $foo = 42;
   ...
}
```

See "Localizing changes" in perlguts for how to use the save stack.

MILLIONS OF MACROS

One thing you'll notice about the Perl source is that it's full of macros. Some have called the pervasive use of macros the hardest thing to understand, others find it adds to clarity. Let's take an example, a stripped-down version the code which implements the addition operator:

```
 1  PP(pp_add)
 2  {
 3      dSP; dATARGET;
 4      tryAMAGICbin_MG(add_amg, AMGf_assign|AMGf_numeric);
 5      {
 6        dPOPTOPnnrl_ul;
 7        SETn( left + right );
 8        RETURN;
 9      }
10  }
```

Every line here (apart from the braces, of course) contains a macro. The first line sets up the function declaration as Perl expects for PP code; line 3 sets up variable declarations for the argument stack and the target, the return value of the operation. Line 4 tries to see if the addition operation is overloaded; if so, the appropriate subroutine is called.

Line 6 is another variable declaration - all variable declarations start with "d" - which pops from the top of the argument stack two NVs (hence "nn") and puts them into the variables "right" and "left", hence the "rl". These are the two operands to the addition operator. Next, we call "SETn" to set the NV of the return value to the result of adding the two values. This done, we return - the "RETURN" macro makes sure that our return value is properly handled, and we pass the next operator to run back to the main run loop.

Most of these macros are explained in perlapi, and some of the more important ones are explained in perlxs as well. Pay special attention to "Background and PERL_IMPLICIT_CONTEXT" in perlguts for information on the "[pad]THX_?" macros.

FURTHER READING

For more information on the Perl internals, please see the documents listed at "Internals and C Language Interface" in perl.